RT-CUDA: A SOFTWARE TOOL FOR CUDA CODE RESTRUCTURING

BY

AYAZ UL HASSAN KHAN

A Dissertation Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

In

COMPUTER SCIENCE AND ENGINEERING

DECEMBER, 2014

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **AYAZ UL HASSAN KHAN** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE AND ENGINEERING**.

Dissertation Committee

[illegible]
Dr. Mayez Abdullah Al-Mouhamed (Adviser)

[illegible]
Dr. Mohammad Saleh Al-Mulhem (Co-adviser)

[illegible]
Dr. Moataz Ahmed (Member)

[illegible]
Dr. Mahmood Niazi (Member)

Tarek Helmy
Dr. Tarek Helmy El-Basuny (Member)

[illegible] Dec 22, 14
Dr. Umar Al-Turki
Department Chairman

[illegible]
Dr. Salam A. Zummo
Dean of Graduate Studies

29/12/14
Date

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

I thank Allah (SWT) to give me strength to complete this work.

ACKNOWLEDGMENTS

I, first of all, thank my adviser Dr. Mayez Abdullah Al-Mouhamed for giving me full support so that I could complete my dissertation. Without his guidance and continuous support, it may be very difficult to me to choose the right research direction and to set appropriate goals. I am also thankful to Dr. Mohammad Saleh Al-Mulhem and Dr. Adel Fadhl Noor Ahmed to provide his full support in the implementation of the code transformations. I also want to thank to my other committee members including Dr. Moataz Ahmed, Dr. Mahmood Niazi, and Dr. Tarek Helmy Al-Basuny to critically evaluate my dissertation work and support in setting the right directions to get success in the intended work. In addition to these, I also want to include the names of my friends at KFUPM who help in different tasks of my disseration to work together in analysing the problem and its solution. These includes Anas Al-Mousa (PhD Student), Ayham Zaza (PhD Student), Allam Fatayer (PhD Student). At last but not the least, I am very much thankful to KFUPM and KAUST (King Abdullah University of Science and Technology) to provide the required computing resources that are needed in my dissertation specially Dr. David Keyes (Professor at KAUST) and Dr. Hatem Ltaief (Computational Scientist at KAUST)

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

NAME:	Ayaz ul Hassan Khan
TITLE OF STUDY:	RT-CUDA: A Software Tool for CUDA Code Restructuring
MAJOR FIELD:	Computer Science and Engineering
DATE OF DEGREE:	December, 2014

Recent development in Graphic Processing Units (GPUs) has opened a new challenge in harnessing their computing power as a new general-purpose computing paradigm with its CUDA parallel programming. However, porting applications to CUDA remains a challenge to average programmers. In this thesis work we have developed a restructuring software compiler (RT-CUDA) with best possible kernel optimizations to bridge the gap between high-level languages and the machine dependent CUDA environment. RT-CUDA is based upon a set of compiler optimizations. RT-CUDA takes a C-like program and convert it into an optimized CUDA kernel with user directives in a configuration file for guiding the compiler. While the invocation of external libraries is not possible with OpenACC commercial compiler, RT-CUDA allows transparent invocation of the most optimized external math libraries like cuSparse and cuBLAS. For this, RT-CUDA uses interfacing APIs, error handling interpretation, and user transparent programming. This enables efficient design of linear algebra solvers (LAS). Evaluation of RT-CUDA has been performed on Tesla K20c GPU with a variety of basic linear algebra operators (M+, MM, MV, VV, etc.) as well as the programming of solvers of systems of linear equations like Jacobi and Conjugate Gradient. We obtained significant speedup over other compilers like OpenACC and GPGPU compilers. RT-CUDA facilitates the design of efficient parallel software for developing parallel simulators (reservoir simulators, molecular dynamics, etc.) which are critical for Oil & Gas industry in KSA. We expect RT-CUDA to be needed by many KSA industries dealing with science and engineering simulation on massively parallel computers like NVIDIA GPUs.

ملخص الرسالة

الاسم الكامل: اياز الحسن خان

عنوان الرسالة: RT-CUDA، أداة برمجية لإعادة هيكلة الشيفرة البرمجية الخاصة بلغة CUDA

التخصص: علوم وهندسة الحاسب الآلي

تاريخ الدرجة العلمية: ديسمبر 2014

التطورات الأخيرة في وحدات معالجة الرسوميات قد فتحت تحديا جديدا في تسخير قوة الحوسبة كنموذج للحواسب ذو التطبيقات العامة باستخدام لغة البرمجة المتوازية الخاصة بها CUDA. لكن ، كتابة البرامج باستخدام CUDA يبقى تحديا للمبرمج العادي. في هذه الرسالة ، طورنا برمجية مفسرة معاد تأهيلها (RT-CUDA) مع أفضل التحسينات الممكنة للنواة لسد الفجوة بين لغات البرمجة عالية المستوى وبيئة CUDA التي تعتمد على طبيعة الآلة التي تعمل عليها. يعتمد RT-CUDA على مجموعة من التحسينات في المفسر. يقوم RT-CUDA بتحويل البرامج العادية والمكتوبة مثلا بلغة C إلى نواة CUDA محسنة مع الأخذ بالاعتبار توجيهات المستخدم لإرشاد المفسر .وبينما يعتبر استدعاء libraries مستحيلا مع OpenACC ، يسمح RT-CUDA باستدعاء واستخدام أفضل الـ librariesالخارجية للرياضيات مثل cuSparse و cuBLAS .لأجل ذلك ، RT-CUDA يستخدم بعض API's للتواصل ، معالجة الأخطاء ويسمح أيضا ببرمجة واضحة وشفافة للمستخدم والذي بدوره يمكّن من تصميم أدوات لحل الجبر الخطي بشكل فعال . تم تقييم RT-CUDA باستخدام معالج الرسوميات Tesla K20 مع مجموعة متنوعة من عمليات الجبر الخطي مثل (M+, MM, MV, VV) وغيرها وأيضا برمجة أدوات لحل نظام من المعادلات الخطية مثل Jacobi و .Conjugate Gradient وفي الحقيقة لقد حصلنا على زيادة في السرعة بشكل كبير جدا مقارنة بمفسر OpenACC ومفسرGPGPU أيضا. يقوم RT-CUDAبتسهيل تصميم البرمجيات المتوازية لتطوير برامج محاكاة متوازية مثل (محاكاة آبار النفط ، الديناميكية الجزيئية وغيرها والتي تعتبر تطبيقات مهمة جدا في صناعة النفط والغاز في المملكة العربية السعودية .نحن نتوقع أن يكون هناك طلب كبير على RT-CUDAمن مجالات الصناعة في السعودية التي تتعامل مع محاكاة التطبيقات العملية والهندسية على أجهزة الحاسب المتوازية الكثيفة مثل معالجات الرسوميات الخاص بـ NVIDIA.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Massively parallel computing has obtained prominence through advances in implementing massive multithreading and recent improvements in its programming. Recent development in Graphic Processing Units (GPUs) has opened a new challenge in harnessing their computing power as a new general purpose computing paradigm. Strong implications are expected on computational science and engineering, especially in the area of discrete numerical simulation.

Modern GPUs use multiple streaming multiprocessors (SMs) with potentially hundreds of cores, fast context switching, and high memory bandwidth to tolerate ever-increasing latencies to main memory by overlapping long-latency loads in stalled threads with useful computation in other threads. The Compute Unified Device Architecture (CUDA) is a simple C-like interface proposed for programming NVIDIA GPUs. However, porting applications to CUDA remains a challenge to average programmers. CUDA places on the programmer the burden of packaging GPU code in separate functions, of explicitly managing data transfer between the host and GPU memories, and of manually optimizing the utilization of the GPU memory.

In this research, we proposed a design and implementation of a software tool for restructuring a C-like program into an optimized CUDA program. Such a restructuring tool greatly simplifies programming for the best performance of GPU which contributes to the spreading of the use of GPU supercomputing applications, scientific computing, and more generally the applications of information technology. This research builds sufficient know-how and state-of-the-art tools for the efficient programming of GPUs that stimulate a long-term interest in the research and development of programming massively parallel computers and their applications especially in the Oil and Gas industry. Specifically, the research outcomes serve the graduate research program and the industry in the kingdom of Saudi Arabia.

1.2 Thesis Contribution

This thesis makes the following contribution:

- We have explored the GPU architecture and CUDA programming framework to utilize GPU devices for general purpose computing
- We have presented a review of several numerical algorithm implementations, code transformations to enhance CUDA kernel performance, CUDA

kernel optimizations, performance models, auto-tuning frameworks, micro-benchmarking of GPU devices

- We have presented a detailed study about the execution model, programming, and synchronization mechanisms of latest GPU architectures including Fermi and Kepler

- We have presented an analysis of the existing GPGPU frameworks/compilers including CUDA-lite, hiCUDA, OpenMPC, PGI, OpenACC, HMPP, R-Stream, and CUDA-CHiLL

- We have Proposed a Restructuring Tool Algorithm (RTA-CUDA) to generate an optimized CUDA parallel program from a given sequential C program based on the identified GPU constraints for maximum performance such that the memory usage (global memory and shared memory), number of blocks, and number of threads per block

- We have developed a Parameter Tuning Algorithm to find an optimal set of CUDA kernel parameters generated by RTA-CUDA to establish the relationships between the influencing parameters

- We have presented the design and implementation of a Restructuring Tool (RT-CUDA) based on RTA-CUDA with an additional set of API functions to call highly optimized library routines for dense and sparse matrices (cuBLAS and cuSPARSE) and synchronization primitives for inter-block synchronization

- We have presented a performance evaluation of the tool using basic linear algebra operations including a set of routines from Lapack BLAS benchmark suite, Jacobi iterative solver with different inter-block synchronization primitives, and various dense/sparse matrix operations

1.3 Thesis Organization

The rest of the dissertation is organized as follows: Chapter 2 presents a review of several numerical algorithms implementations, code transformations to enhance CUDA kernel performance, CUDA kernel optimizations, performance models, auto-tuning frameworks, micro-benchmarking of GPU devices. Chapter 3 presents a detailed study about the execution model, programming, and synchronization mechanisms of latest GPU architectures including Fermi and Kepler. Chapter 4 presents an exploration of automatic optimizations for CUDA programming including a 3-step algorithm that has been proposed to tune the CUDA kernel parameters and enhance GPU resource utilization. We have also explored several CUDA kernel optimizations with some categorizations. It also presents a detailed analysis of the existing GPGPU frameworks/compilers including CUDA-lite, hiCUDA, OpenMPC, PGI, OpenACC, HMPP, R-Stream, and CUDA-CHiLL. Based on the analysis of kernel optimizations and existing GPGPU frameworks, RT-CUDA design specifications have been presented. Chapter 5 presents a review and selection of compiler framework for RT-CUDA implementations, a Restructuring Tool Algorithm (RTA-CUDA) to generate an optimized CUDA parallel program from a given sequential C program, a Parameter Tuning Algorithm to find an optimal set of CUDA kernel parameters generated by RTA-CUDA, design and implementation of Restructuring Tool (RT-CUDA) with the user manual and code examples. Chapter 6 discusses the performance evaluation of the tool that has been performed using basic linear algebra operations including Lapack BLAS benchmark, Jacobi iterative solver with different inter-block synchronization primitives, dense and sparse matrix operations. At the end, Chapter 7 summarizes this dissertation and highlights some possible enhancements in RT-CUDA.

CHAPTER 2

LITERATURE REVIEW

Performance study of general-purpose GPU programming have been reported [1] for applications such as SRAD structured grid, back-propagation unstructured grid, data encryptions standard, NeedlemanWunsch dynamic programming, and k-means data mining. Impressive speedups ranging from 2.9 to 35 for the above applications have been achieved over single threaded programs.

Although tremendous success has been achieved in numerical applications, some limitations have also been reported when the available parallelism is semi-static where the inherent parallelism is irregular. A CUDA implementation for the gravitational N-body simulations using GPU is reported [2]. The GPU performs force calculation and updating, while the host CPU performs the predictor, corrector, and integration steps. Implementation is based on two direct N-body integration codes, using the 4th order predictor-corrector Hermite integrator with block time-steps, and one Barnes-Hut tree-code, which uses a second order leapfrog integration. The above implementation merely maps the computation of pair-wise particle interactions onto the GPU which makes the time-consuming updating of the neighbor lists on the CPU a bottleneck since synchronization and frequent data transfer between host CPU and GPU can often be problematic for GPGPU implementations.

In the following sections, we have introduced recent developments regarding numerical algorithms implementation, automatic code transformations, CUDA kernel optimizations, performance modeling, auto-tuning, and microbenchmarking.

2.1 Numerical Algorithms Implementations

Dumitrescu et. al [3] have implemented fast matrix multiplication algorithms Strassen [4] and Winograd [5] on MIMD distributed memory architectures of ring and torus topologies; a generalization to a hyper-torus is also given. Complexity and efficiency are analyzed and good asymptotic behaviour is proved. The presented parallel implementations of fast matrix multiplication algorithms on MIMD architectures are proven to be faster than standard parallel algorithms, on ring or on torus topologies. Speed improvement becomes for matrix dimensions of 200 (about 10% faster) and, for sufficiently big n, measured timings are close to the theoretical predicted values, that is 30% when the local sequential method is the same; when sequential methods are different (fast sequential for fast parallel, standard sequential for standard parallel) maximal measured speed growth was 75% for n = 2048; this latter result is consistent with the one reported by Bailey

[6], for the same matrix dimension (in another context, a Cray-2 supercomputer), i.e. 101%, but with 35% of this improvement due to other causes. However, a disadvantage of these parallel fast algorithms is the fixed number of processors, while standard algorithms can be easily customized. The parallel mixed algorithm (MixT), more flexible from this point of view, is good only for standard sequential method. All proposed algorithms can be efficient on dedicated hardware. On a configurable topology, such as the supernode, these algorithms require a smaller number of processors for the same speed (compared with standard ones), an advantage in a multiuser environment. Authors conclude that fast matrix multiplication algorithms cannot be ignored, on MIMD computers as well as on SIMD computers. They can bring, by themselves, a considerable speed-up of applications, that is more important than the implied implementation difficulties.

Li et. al [7] provide efficient single-precision and integer GPU implementations of Strassens algorithm as well as of Winograds variant. On an NVIDIA C1060 GPU, a speedup of 32% - 35% is obtained for Strassens 4-level implementation and 33% - 36% for Winograds variant relative to the sgemm (integer version of sgemm) code in CUBLAS 3.0 when multiplying 16384 x 16384 matrices. The maximum numerical error for the single-precision implementations is about 2 orders of magnitude higher than those for sgemm when n = 16384 and is zero for the integer implementations.

Generalized sparse matrix-matrix multiplication is a key primitive for many high performance graph algorithms as well as some linear solvers such as multi-grid. Buluç et. al [8] present the first parallel algorithms that achieve increasing speedups for an unbounded number of processors. The algorithms are based on two-dimensional block distribution of sparse matrices where serial sections use a novel hypersparse kernel for scalability. They give a state-of-the-art MPI implementation of one of the algorithms. Experiments show scaling up to thousands of processors on a variety of test scenarios.

The Sparse Matrix-Vector product (SpMV) is a key operation in engineering and scientific computing. Methods for efficiently implementing it in parallel are critical to the performance of many applications. Modern Graphics Processing Units (GPUs) coupled with the advent of general purpose programming environments like NVIDIAs CUDA, have gained interest as a viable architecture for data-parallel general purpose computations. Most of the SpMV implementations using CUDA based on common sparse matrix format have already appeared. Among them, the performance of implementation based on ELLPACK-R format is the best. However, in this implementation, when the maximum number of nonzeros per row does substantially differ from the average, thread is suffering from load imbalance. A new matrix storage format called ELLPACK-RP [9] has been proposed, which combines ELLPACK-R format with JAD format, and implements the SpMV using CUDA based on it. The result proves that it can decrease the load imbalance and improve the SpMV performance efficiently.

A blocked sparse matrix-vector multiplication for NVIDIA GPUs [10] has been implemented. The implementation is faster on matrices having many high fill-ratio blocks but slower on matrices with low number of non-zero elements per row.

2.2 Exploration of Code Transformations for Enhancing Performance of CUDA Kernels

CUDA programming requires an expert level understanding of the memory hierarchy and execution model to reach peak performance. Even for experts, rewriting a program to exploit the architecture in achieving high speedups can be tedious and error prone. Several high-level interfaces [11, 12, 13] have been proposed to perform code translation to generate CUDA programs with less burden to the programmers. Most execution of a scientific program is spent on loops. Compiler analysis and compiler optimizations have been proposed to make the execution of loops faster. In the following we review the proposed approaches.

CUDA-lite [11] is an experimental enhancement to CUDA that allows programmers to deal only with global memory with transformations to leverage the complex memory hierarchy. A set of annotations describing certain properties of the data structures and code regions designated for GPU execution are proposed. The tool analyze the code along with these annotations and determine if the memory bandwidth can be conserved and latency can be reduced by utilizing any special memory types and/or by massaging memory access patterns. Upon detection of an opportunity, CUDA-lite performs the transformations and code insertions needed. CUDA-lite is designed as a source-to-source translator. The output is CUDA code with explicit memory-type declarations and data transfers for a particular GPU. The major transformations performed by CUDA-lite are as follows:

1. Inserting shared memory variables
2. Performing loop tiling
3. Generating memory coalesced loads and/or stores
4. Replacing the original global memory accesses with accesses to the corresponding data in shared memory

Authors claim the tool produces code with performance comparable to hand-coded versions.

A framework for source-to-source translation of standard OpenMP applications into CUDA-based code is proposed [12]. It has two phases: (1) a compile-time optimization techniques (OpenMP Stream Optimizer), and (2) OpenMP to GPGPU translation system (O2G baseline translator with CUDA optimizer). The OpenMP Stream optimizer takes as input a standard OpenMP program (CPU-oriented) and applies following high-level optimization techniques: parallel loop-swap and loop-collapsing, to generate and optimized OpenMP program for GPGPUs. The translation is done with the following steps:

1. It first identifies the potential kernel regions based on the defined interpretation of OpenMP constructs and directives.

2. Transform the identified kernel regions into separate kernel functions. At this stage, it performs both work partitioning and data mapping. For work partitioning, the compiler calculates the maximum number of threads needed for each work-sharing sub-region contained in the kernel region and then it calculates the number of blocks using default thread block size provided as a command line option. For data mapping, the compiler uses the OpenMP data sharing rules and do the mapping as follows:

   (a) Shared data are mapped to global memory

   (b) Threadprivate data are replicated and allocated on global memory for each thread

   (c) Private data are mapped to register banks assigned for each thread

After translation, it performs following CUDA optimizations:

1. Caching of frequently accessed global data: the compiler performs the requisite data flow analysis to identify temporal locality of global data, based on that it loaded frequently accessed global data into fast memory spaces such as register and shared memory.

2. Memory transpose for threadprivate array: this matrix transpose changes intra-thread array access patterns from row-wise to column-wise, so that adjacent threads can access adjacent data, as needed for coalesced accesses.

3. Memory transfer reduction: the compiler performs data flow analysis for each kernel and removes unnecessary data transfers between CPU and GPU.

For performance evaluations, it uses two important kernels (JACOBI and SPMUL) and two NAS OpenMP Parallel Benchmarks (EP and CG). Experimental results show that the described translator and compile-time optimizations work well on both regular (Jacobi and EP) and irregular (SPMUL and CG) applications, leading to performance improvements of up to 50x over the un-optimized translation (up to 328x over serial on a CPU). The work can be extended to include automatic tuning of optimizations to exploit shared memory and other special memory units more aggressively.

A high-level directive-based compiler (hiCUDA) [13] is proposed to ease the task of writing CUDA programs. The compiler translates a hiCUDA program to a CUDA program using a computation model to identify code regions that are intended to be executed on the GPU and a data model in which programmers allocate and de-allocate memory on the GPU and move data between the host memory and the GPU memory. The hiCUDA compiler, built around Open64 (version 4.1), consists of following three components:

1. The GNU 3 front-end, which is extended from the one in Open64.
2. A compiler pass that lowers hiCUDA directives, which uses several existing modules in Open64, such as data flow analysis, inter-procedural analysis and array specification analysis.
3. The CUDA code generator, which is extended from the C code generator in Open64.

Evaluation of five CUDA benchmarks (MM, CP, SAD, TPACF, RPES) shows that the provided simplicity and flexibility come at no expense to performance as execution times is within 2% of that of the hand-written CUDA version. The work can be extended to include the following within hiCUDA compiler:

- Automatic transformations of standard loops that is required before inserting hiCUDA directives.
- Data dependence analysis to validate the partitioning scheme of kernel computation and detect non-optimized memory access patterns.
- Delegate the work of inserting hiCUDA data directives to the compiler, which can determine an optimal data placement strategy using various data analyses.

A source-to-source compiler transformation (CUDA-CHiLL) [14] aims at alleviating the need for understanding memory hierarchy and execution model in writing optimized CUDA programs. It proposes a source-to-source transformation based on the polyhedral program transformation and ChiLLframework which is capable of composing transformations while preserving the correctness of the program at each step. It focuses on loop tiling, data copy, and unrolling. The authors claims that optimizing the BLAS library routines yields results comparable to hand-tuned versions in some cases and outperforming hand-tuned in other cases.

CUDA-ChiLL is not providing fully-automatic transformations as it is based on the transformation recipe interface which is a script that needs to be written by the programmer to instruct the compiler about how to do the transformations which is also an extra burden on the programming side. Alternatively, the better approach is to provide some compile time pragmas like CUDA-Lite and hiCUDA which the programmer can just insert into the existing un-optimized code.

A dynamic instrumentation infrastructure [15] for PTX programs has been implemented that procedurally transforms kernels and manages related data structures. The performing instrumentation within the GPU Ocelot dynamic compiler infrastructure provides unique capabilities not available to other profiling and instrumentation toolchains for GPU computing. To demonstrate the utility of this instrumentation capability, three example scenarios has been used: (1) performing workload characterization accelerated by a GPU, (2) providing load imbalance information for use by a resource allocator, and (3) providing compute utilization feedback to be used on-line by a simulated process scheduler that might be found in a hypervisor. Additionally, both (1) the compilation overheads of performing dynamic compilation and (2) the increases in runtimes when executing instrumented kernels have been measured. On average, compilation overheads due to instrumentation consisted of 69% of the time needed to parse a kernel module, in the case of the Parboil benchmark suite. Slowdowns for instrumenting each basic block ranged from 1.5x to 5.5x, with the largest slowdowns attributed to kernels with large numbers of short, compute-bound blocks.

A novel optimizing compiler [16] for general pur-pose computation on graphics processing units (GPGPU) has been developed. It addresses two major challenges of developing high performance GPGPU programs: effective utilization of GPU memory hierarchy and judicious management of parallelism. The input to the compiler is a nave GPU kernel function, which is functionally correct but without any consideration for performance optimization. The compiler analyzes the code, identifies its memory access patterns, and generates both the optimized kernel and the kernel invocation parameters. The optimization process includes vectorization and memory coalescing for memory bandwidth enhancement, tiling and unrolling for data reuse and parallelism management, and thread block remapping or address-offset insertion for partition-camping elimination. The experiments on a set of scientific and media processing algorithms show that the optimized code achieves very high performance, either superior or very close to the highly fine-tuned library, NVIDIA CUBLAS 2.2, and up to 128 times speedups over the naive versions. Another distinguishing feature of the compiler is the understandability of the optimized code, which is useful for performance analysis and algorithm refinement.

JCUDA [17], a programming interface, has been proposed for Java programmers to invoke CUDA kernels. Using this interface, programmers can write Java codes that directly call CUDA kernels, and delegate the responsibility of generating the Java-CUDA bridge codes and host-device data transfer calls to the compiler. Preliminary performance results show that this interface can deliver significant performance improvements to Java programmers.

2.3 Exploration of Optimizations for Enhancing Performance of CUDA Kernels

Programming issues for many-core architectures has been studied [18] and proposed the optimization strategies on following architectures:

- IBM Cyclops 64, a many-core chip architecture
- Many-core Graphics Processing Unit

It first presents a model for performance estimation of parallel FFT algorithm for abstract many-core architecture. This model is based on a cost function having three components: memory accesses, computation, and synchronization. Secondly, it presents a framework for fine-grained task-based execution. It provides effective hardware utilization by proper load balancing. On multi-GPU systems, it achieves near linear speedup, good dynamic load balancing and significant performance improvement over standard CUDA API. The framework has to be implemented at operating system level to perform the scheduling of tasks from host to device.

Optimizing programs using the Vector blocking techniques [19] over hybrid architectures (multicore and GPU) proved to be useful for improving performance of the matrix multiply routine (GEMM). Orders of magnitude acceleration is reported compared to multicore without GPU accelerators when architecture and algorithm-specific optimizations are used for implementing dense linear algebra solvers such as the MAGMA library [20]. A three-step optimization is proposed for the QR factorization [21]. QR is factorized as a sequence of tasks with chosen granularity. The kernel for each task is designed. Finally, static scheduling is used when a priori knowledge is available. Otherwise, dynamic scheduling is used by managing data availability and coherency. The reported performance is very close to that obtained using Linear Programming with some limited portability. The implementation complements kernels already available in the MAGMA library. For more details the reader may refer to [22] for the architecture and programming of GPUs.

Ryoo et. al [23] shows the complexity involved in optimizing applications for GeForce 8800 GTX using CUDA and one relatively simple methodology for reducing the workload involved in the optimization process. They show how optimizations interact with the architecture in complex ways, initially prompting an inspection of the entire configuration space to find the optimal configuration. Even for a seemingly simple application such as matrix multiplication, the optimal configuration can be unexpected. They also present metrics derived from static code that capture the first-order factors of performance and demonstrate how these metrics can be used to prune many optimization configurations, down to those that lie on a Pareto-optimal curve. This reduces the optimization space by as much as 98%.

Nath et. al [24] present an improved matrix-matrix multiplication routine (GEMM) in the MAGMA BLAS library that targets the Fermi GPUs. They show how to modify the previous MAGMA GEMM kernels in order to make a more efficient use of the Fermis new architectural features, most notably their extended memory hierarchy and sizes. The improved kernels run at up to 300 GFlop/s in double and up to 600 GFlop/s in single precision arithmetic (on a C2050), which is 58% of the theoretical peak. The improved kernels have been compared with the currently available in CUBLAS 3.1. Further, they show the effect of the new kernels on higher level dense linear algebra (DLA) routines such as the one-sided matrix factorizations, and compare their performances with corresponding, currently available routines running on homogeneous multicore systems.

Nath et. al [25] also present a new algorithm for optimizing the Symmetric Matrix Vector Product (SYMV) kernel on GPUs. The optimized SYMV in single precision brings up to a 7x speed up compared to the CUBLAS 4.0 NVIDIA library on the GTX 280 GPU. This SYMV kernel tuned for Fermi C2050 is 4.5x faster than CUBLAS 4.0 in single precision and 2x faster than CUBLAS 4.0 in double precision. Moreover, the techniques used and described in the paper are general enough to be of interest for developing high-performance GPU kernels beyond the particular case of SYMV.

Zein and Rendell [26] has explored the effect of these different options on the performance of a routine that evaluated sparse matrix vector products. They have proposed a process for analysing performance and selecting the subset of implementations that perform best. The potential for mapping sparse matrix attributes to optimal CUDA sparse matrix vector product implementation has also been discussed.

An optimized version of Sparse Matrix Vector (SpMV) Multiplication [27] has been implemented on NVIDIA GPUs using CUDA. The three optimizations has been performed that include: (1) optimized CSR storage format, (2) optimized threads mapping, and (3) avoiding divergence judgment. The evaluation has been done on GeForce 9600 GTX, connect to Windows xp 64-bit system. In comparison with NVIDIA's SpMV library and NVIDIA's CUDDPA library, the results show that optimizing sparse matrix-vector multiplication on CUDA achieves better performance than other SpMV implementations.

Another optimization of SpMV multiplication [28] has been proposed based on matrix bandwidth/profile reduction techniques. Computational time required to access dense vector is decoupled from SpMV computation. By reducing the matrix profile, the time required to access dense vector is reduced by 17% (for SP) and 24% (for DP). Reduced matrix bandwidth enables column index information compression with shorter formats, resulting in a 17% (for SP) and 10% (for DP) execution time reduction for accessing matrix data under ELLPACK format. The overall speedup for SpMV is 16% and 12.6% for the whole matrix test suite. The proposed optimization can be combined with other SpMV optimizations such as register blocking.

An improvied compressed sparse row storage (ICSR) [29] used to settle the problem of the global memory alignment in the vector kernel on Graphics processing Unit (GPU) is given. The experiments on matrices with different sizes demonstrate that the vector kernel with ICSR storage format could improve the performance by 5%-30% for SpMV comparing with vector kernel with CSR, especially for the largescale unstructured sparse matrix-vector product, the effect is more obvious.

An optimization scheme [30] has been proposed for a sparse matrix parallel iteration algorithm on a hybrid multi-core parallel system consisting of CPU and GPU. The scheme carries out performance improvement in two ways i.e. the multi-level storage structure and the memory access mode of CUDA. Experimental results show that the parallel algorithm on hybrid multi-core system can gain higher performance than the original linear Jacobi iteration algorithm on CPU. In addition, the optimization scheme is effective and feasible.

2.4 Performance Modeling Approaches

An analytical model [31] to estimate the execution cycles of parallel applications on GPU architectures has been proposed. It is based on following two metrics:

- MWP (Memory Warp Parallelism) represents the maximum number of warps per SM that can access the memory simultaneously during the time period from right after the SM processor executes a memory instruction from one warp until all the memory requests from the same warp are serviced. It is determined by the memory bandwidth, memory bank parallelism and the number of running warps per SM.

- CWP (Computation Warp Parallelism) represents the number of warps that the SM processor can execute during one memory warp waiting period plus one. A value one is added to include the warp itself that is waiting for memory values that is CWP is always greater than or equal to 1. Unlike arithmetic intensity, CWP also considers timing information. CWP is mainly used to decide whether the total execution time is dominated by computation cost or memory access cost. When CWP is greater than MWP, the execution cost is dominated by memory access cost. However, when MWP is greater than CWP, the execution cost is dominated by computation cost.

Evaluation shows that the geometric mean of absolute error of the proposed analytical model on micro-benchmarks is 5.4% and on GPU computing applications is 13.3%.

An analytical model [32] to predict the performance of general-purpose applications on a GPU architecture has been presented. The model is designed to provide performance information to an auto-tuning compiler and assist it in narrowing down the search to the more promising implementations. It can also be incorporated into a tool to help programmers better assess the performance bottlenecks in their code. To identify the performance bottlenecks accurately, an abstract interpretation (work flow graph) of a GPU kernel has been introduced based on which the execution time of a GPU kernel has been estimated. The proposed model captures full system complexity and shows high accuracy in predicting the performance trends of different optimized kernel implementations. The performance model has been validated for matrix multiply, prefix sum scan, FFT, and sparse matrix-vector benchmarks. The evaluation shows that there is good agreement between predicted and observed performance rankings for the various tuning versions of these kernels and that the model captures the effect of all major performance factors for GPU architecture.

An integrated analytical and profile-based CUDA performance modeling approach [33] to accurately predict the kernel execution times of sparse matrix-vector multiplication for CSR, ELL, COO, and HYB SpMV CUDA kernels has been proposed. Based on the experiments conducted on a collection of 8 widely-used testing matrices on NVIDIA Tesla C2050, the execution times predicted by the model match the measured execution times of NVIDIAs SpMV implementations very well. Specifically, for 29 out of 32 test cases, the performance differences are under or around 7%. For the rest 3 test cases, the differences are between 8% and 10%. For CSR, ELL, COO, and HYB SpMV kernels, the differences are 4.2%, 5.2%, 1.0%, and 5.7% on the average, respectively.

2.5 Auto-Tuning

Software tuning of high-performance kernels [34] for GPUs is critical for efficiently running linear solver algorithms such as the Basic Linear Algebra Subprograms (BLAS) kernels.

Cui et. al [35] discuss about their experiences in improving the performance of GEMM (both single and double precision) on Fermi architecture using CUDA, and how the new features of Fermi such as cache affect performance. It is found that the addition of cache in GPU on one hand helps the processers take advantage of data locality occurred in runtime but on the other hand renders the dependency of performance on algorithmic parameters less predictable. Auto-tuning then becomes a useful technique to address this issue. Their auto-tuned SGEMM and DGEMM reach 563 GFlops and 253 GFlops respectively on Tesla C2050. The design and implementation entirely use CUDA and C and have not benefited from tuning at the level of binary code.

Guo and Wang present an auto-tuning framework that can automatically compute and select CUDA parameters for SpMV to obtain the optimal performance on specific GPUs. The framework is evaluated on two NVIDIA GPU platforms: GeForce 9500 GTX and GeForce GTX 295.

Kamil et. al [36] presents a stencil auto-tuning framework that significantly advances programmer productivity by automatically converting a straightforward Fortran 95 stencil expression to tuned implementations in Fortran, C, or CUDA, thus allowing performance portability across diverse computer architectures, including the AMD Barcelona, Intel Nehalem, Sun Victoria Falls, and the latest NVIDIA GPUs. Results show that the generalized methodology delivers significant performance gains of up to 22x speedup over the reference serial implementation. Overall they demonstrate that such domain-specific auto-tuners hold enormous promise for architectural efficiency, programmer productivity, performance portability, and algorithmic adaptability on existing and emerging multicore systems.

Li et. al [37] describe some GPU GEMM auto-tuning optimization techniques that allow programmers to keep up with changing hardware by rapidly reusing, rather than reinventing, the existing ideas. Auto-tuning is a very practical solution where in addition to getting an easy portability, one can often get substantial speedups even on current GPUs (e.g. up to 27% in certain cases for both single and double precision GEMMs on the GTX 280).

Kurzak et. al [38] present a methodology for producing matrix multiplication kernels tuned for a specific architecture, through a canonical process of heuristic autotuning, based on generation of multiple code variants and selecting the fastest ones through benchmarking. It also is the authors belief that the process can be easily generalized to other types of workloads, including more complex kernels and more bandwidth-bound kernels. In principle, this should be the case as long as the code can be parameterized and its properties, such as demand for registers and shared memory, expressed as functions of the parameters.

2.6 Micro-Benchmarking

Wong et. al [39] develops a microbenchmark suite and measure the CUDA-visible architectural characteristics of the Nvidia GT200 (GTX280) GPU. Various undisclosed characteristics of the processing elements and the memory hierarchies are measured. This analysis exposes undocumented features that impact program performance and correctness. These measurements can be useful for improving performance optimization, analysis, and modeling on this architecture and offer additional insight on the decisions made in developing this GPU. The results validated some of the hardware characteristics presented in the CUDA Programming Guide [40], but also revealed the presence of some undocumented hardware structures such as mechanisms for control flow and caching and TLB hierarchies. In addition, in some cases the findings deviated from the documented characteristics (e.g., texture and constant caches).

Recent development in Graphic Processing Units (GPUs) has opened a new challenge in harnessing their computing power as a new general purpose computing paradigm. Strong implications are expected on computational science and engineering, especially in the area of discrete numerical simulation [41]. Modern GPUs use multiple streaming multiprocessors (SMs) with potentially hundreds of cores, fast context switching, and high memory bandwidth to tolerate ever-increasing latencies to main memory by overlapping long-latency loads in stalled threads with useful computation in other threads [42]. The Compute Unified Device Architecture (CUDA) is a simple C-like interface proposed for programming NVIDIA GPUs. However, porting applications to CUDA remains a challenge to average programmers. CUDA places on the programmer the burden of packaging GPU code in separate functions, of explicitly managing data transfer between the host and GPU memories, and of manually optimizing the utilization of the GPU memory [13]. So, in order to efficiently utilize the GPU resources, implementations showed be done with detailed understanding of the underlying architecture and CUDA kernel optimizations that is very tedious even for expert programmers and requires sufficient programming efforts as shows in the literature review above. This motivates us for the design and implementation of a software tool for restructuring a C-like program into an optimized CUDA program. Such a restructuring tool greatly simplifies programming for the best performance of GPU which contributes to the spreading of the use of GPU supercomputing applications, scientific computing, and more generally the applications of information technology. This research builds sufficient know-how and state-of-the-art tools for the efficient programming of GPUs that stimulate a long-term interest in the research and development of programming massively parallel computers and their applications especially in the Oil and Gas industry. Specifically, the research outcomes serve the graduate research program and the industry in the kingdom of Saudi Arabia.

CHAPTER 3

GATHERED DETAILED INFORMATION AND ANALYSIS OF RECENT GPU AND CUDA

Originally, GPUs were designed for graphics-based applications. With the elimination of key architecture limitations, GPUs have evolved to become more widely used for general-purpose computation. GPU consists of array of highly threaded streaming multiprocessor (SM) that has a number of streaming processors (SP). The number of execution units and CUDA cores depends on the architecture class of the GPU device. The most commonly used architectures in use today are Fermi [43] and Kepler [44].

3.1 GPU Architectures

3.1.1 Fermi Architecture

Fermi GPU has been developed with a completely new approach to design to create the world's first computational GPU. Following are some of the key improvements in Fermi since the original G80 and GT200 GPU architectures:

- **Improve Double Precision Performance:** while single precision floating point performance was on the order of ten times the performance of desktop CPUs, some GPU computing applications desired more double precision performance as well.

- **ECC support:** ECC allows GPU computing users to safely deploy large numbers of GPUs in datacenter installations, and also ensure data-sensitive applications like medical imaging and financial options pricing are protected from memory errors.

- **True Cache Hierarchy:** some parallel algorithms were unable to use the GPU's shared memory, and users requested a true cache architecture to aid them.

- **More Shared Memory:** many CUDA programmers requested more than 16 KB of SM shared memory to speed up their applications.

- **Faster Context Switching:** users requested faster context switches between application programs and faster graphics and compute interoperation.

- **Faster Atomic Operations:** users requested faster read-modify-write atomic operations for their parallel algorithms.

With these requests in mind, a processor has been designed that greatly increases raw compute horsepower, and through architectural innovations, also offers dramatically increased programmability and compute efficiency. The key architectural highlights of Fermi are:

- Third Generation Streaming Multiprocessor (SM)
  - 32 CUDA cores per SM, 4x over GT200
  - 8x the peak double precision floating point performance over GT200
  - Dual Warp Scheduler simultaneously schedules and dispatches instructions from two independent warps
  - 64 KB of RAM with a configurable partitioning of shared memory and L1 cache
- Second Generation Parallel Thread Execution ISA
  - Unified Address Space with Full C++ Support
  - Optimized for OpenCL and DirectCompute
  - Full IEEE 754-2008 32-bit and 64-bit precision
  - Full 32-bit integer path with 64-bit extensions
  - Memory access instructions to support transition to 64-bit addressing

    - Improved Performance through Predication

- Improved Memory Subsystem
    - NVIDIA Parallel DataCacheTM hierarchy with Configurable L1 and Unified L2 Caches
    - First GPU with ECC memory support
    - Greatly improved atomic memory operation performance

- NVIDIA GigaThreadTM Engine
    - 10x faster application context switching
    - Concurrent kernel execution
    - Out of Order thread block execution
    - Dual overlapped memory transfer engines

The first Fermi based GPU, implemented with 3.0 billion transistors, features up to 512 CUDA cores. A CUDA core executes a floating point or integer instruction per clock for a thread. The 512 CUDA cores are organized in 16 SMs of 32 cores each. The GPU has six 64-bit memory partitions, for a 384-bit memory interface, supporting up to a total of 6 GB of GDDR5 DRAM memory. A host interface connects the GPU to the CPU via PCI-Express. The GigaThread global scheduler distributes thread blocks to SM thread schedulers.

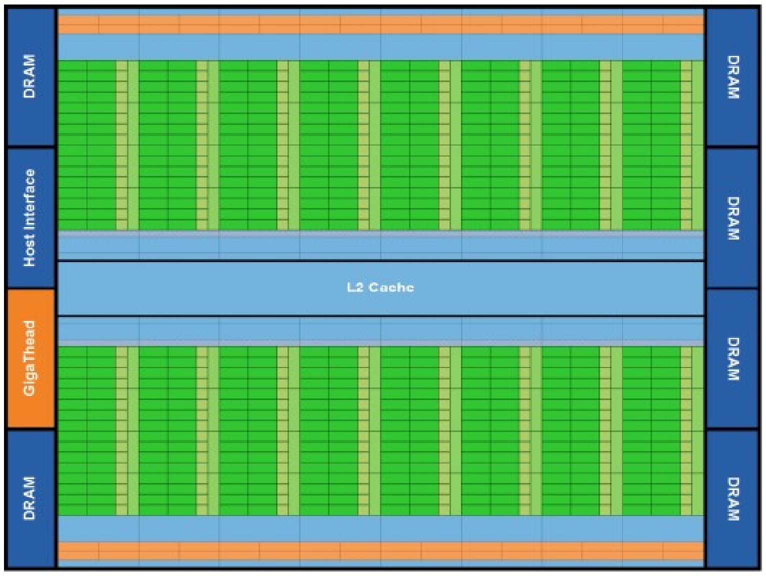


Figure 3.1: Fermi GPU Device Block Diagram

512 High Performance CUDA cores

Each SM (see Figure 3.2) features 32 CUDA processors - a fourfold increase over prior SM designs. Each CUDA processor has a fully pipelined integer arithmetic logic unit (ALU) and floating point unit (FPU). Prior GPUs used IEEE 754-1985 floating point arithmetic. The Fermi architecture implements the new IEEE 754-2008 floating-point standard, providing the fused multiply-add (FMA) instruction for both single and double precision arithmetic. FMA improves over a multiply-add (MAD) instruction by doing the multiplication and addition with a single final rounding step, with no loss of precision in the addition. FMA is more accurate than performing the operations separately. GT200 implemented double precision FMA.

In GT200, the integer ALU was limited to 24-bit precision for multiply operations; as a result, multi-instruction emulation sequences were required for in-

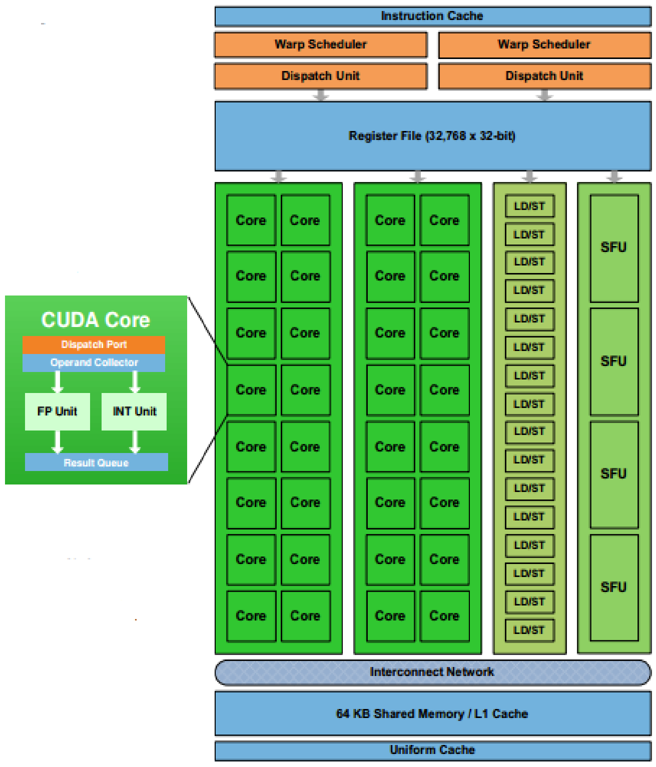


Figure 3.2: Fermi SM Architecture

teger arithmetic. In Fermi, the newly designed integer ALU supports full 32-bit precision for all instructions, consistent with standard programming language requirements. The integer ALU is also optimized to efficiently support 64-bit and extended precision operations. Various instructions are supported, including Boolean, shift, move, compare, convert, bit-field extract, bit-reverse insert, and population count.

16 Load/Store Units

Each SM has 16 load/store units, allowing source and destination addresses to be calculated for sixteen threads per clock. Supporting units load and store the data at each address to cache or DRAM.

Four Special Function Units

Special Function Units (SFUs) execute transcendental instructions such as sin, cosine, reciprocal, and square root. Each SFU executes one instruction per thread, per clock; a warp executes over eight clocks. The SFU pipeline is decoupled from the dispatch unit, allowing the dispatch unit to issue to other execution units while the SFU is occupied.

Designed for Double Precision

Double precision arithmetic is at the heart of HPC applications such as linear algebra, numerical simulation, and quantum chemistry. The Fermi architecture has been specifically designed to offer unprecedented performance in double precision; up to 16 double precision fused multiply-add operations can be performed per SM, per clock, a dramatic improvement over the GT200 architecture.

Dual Warp Scheduler

The SM schedules threads in groups of 32 parallel threads called warps. Each SM features two warp schedulers and two instruction dispatch units (see Figure 3.3), allowing two warps to be issued and executed concurrently. Fermi's dual warp scheduler selects two warps, and issues one instruction from each warp to a group of sixteen cores, sixteen load/store units, or four SFUs. Because warps execute independently, Fermi's scheduler does not need to check for dependencies from within the instruction stream. Using this elegant model of dual-issue, Fermi achieves near peak hardware performance.

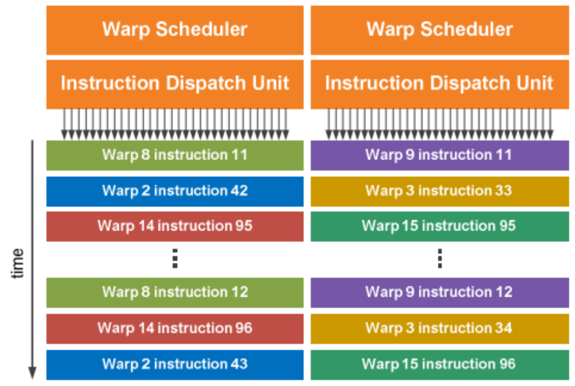


Figure 3.3: Fermi GPU Warp Scheduling Mechanism

Most instructions can be dual issued; two integer instructions, two floating instructions, or a mix of integer, floating point, load, store, and SFU instructions can be issued concurrently. Double precision instructions do not support dual dispatch with any other operation.

64 KB Configurable Shared Memory and L1 Cache

One of the key architectural innovations that greatly improved both the programmability and performance of GPU applications is on-chip shared memory. Shared memory enables threads within the same thread block to cooperate, facilitates extensive reuse of on-chip data, and greatly reduces off-chip traffic. Shared memory is a key enabler for many high-performance CUDA applications.

G80 and GT200 have 16 KB of shared memory per SM. In the Fermi architecture, each SM has 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache or as 16 KB of Shared memory with 48 KB of L1 cache.

For existing applications that make extensive use of Shared memory, tripling the amount of Shared memory yields significant performance improvements, especially for problems that are bandwidth constrained. For existing applications that use Shared memory as software managed cache, code can be streamlined to take advantage of the hardware caching system, while still having access to at least 16 KB of shared memory for explicit thread cooperation. Best of all, applications that do not use Shared memory automatically benefit from the L1 cache, allowing high performance CUDA programs to be built with minimum time and effort.

3.1.2 Kepler Architecture

NVIDIA's Kepler GPU architecture simplifies parallel programs development and revolutionize high performance computing. With more processing power in comparison to old generations of GPU devices, it solves the world's most difficult computing problems. Kepler GPU introduced new methods for parallel program optimizations and increased parallel workload execution on the GPU.

Comprising 7.1 billion transistors, Kepler GK110 is not only the fastest, but also the most architecturally complex microprocessor ever built. Adding many new innovative features focused on compute performance, GK110 was designed to be a parallel processing powerhouse for Tesla® and the HPC market.

Kepler GK110 will provide over 1 TFlop of double precision throughput with greater than 80% DGEMM efficiency versus 60-65% on the prior Fermi architecture.

In addition to greatly improved performance, the Kepler architecture offers a huge leap forward in power efficiency, delivering up to 3x the performance per watt of Fermi.

The following new features in Kepler GK110 enable increased GPU utilization, simplify parallel program design, and aid in the deployment of GPUs across the spectrum of compute environments ranging from personal workstations to supercomputers:

- **Dynamic Parallelism:** adds the capability for the GPU to generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths, all without involving the CPU. By providing the flexibility to adapt to the amount and form of parallelism through the course of a program's execution, programmers can expose more varied kinds of parallel work and make the most efficient use the GPU as a computation evolves. This capability allows less-structured, more complex tasks to run easily and effectively, enabling larger portions of an application to run entirely on the GPU. In addition, programs are easier to create, and the CPU is freed for other tasks.

- **Hyper-Q:** enables multiple CPU cores to launch work on a single GPU simultaneously, thereby dramatically increasing GPU utilization and significantly reducing CPU idle times. Hyper-Q increases the total number of connections (work queues) between the host and the GK110 GPU by allowing 32 simultaneous, hardware-managed connections (compared to the single connection available with Fermi). Hyper-Q is a flexible solution that allows separate connections from multiple CUDA streams, from multiple Message Passing Interface (MPI) processes, or even from multiple threads within a process. Applications that previously encountered false serialization across tasks, thereby limiting achieved GPU utilization, can see up to dramatic performance increase without changing any existing code.

- **Grid Management Unit:** Enabling Dynamic Parallelism requires an advanced, flexible grid management and dispatch control system. The new GK110 Grid Management Unit (GMU) manages and prioritizes grids to be executed on the GPU. The GMU can pause the dispatch of new grids and queue pending and suspended grids until they are ready to execute, providing the flexibility to enable powerful runtimes, such as Dynamic Parallelism. The GMU ensures both CPU and GPU generated workloads are properly managed and dispatched.

- **NVIDIA GPUDirect™:** a capability that enables GPUs within a single computer, or GPUs in different servers located across a network, to directly exchange data without needing to go to CPU/system memory. The RDMA feature in GPUDirect allows third party devices such as SSDs, NICs, and IB adapters to directly access memory on multiple GPUs within the same system, significantly decreasing the latency of MPI send and receive messages to/from GPU memory. It also reduces demands on system memory bandwidth and frees the GPU DMA engines for use by other CUDA tasks. Kepler GK110 also supports other GPUDirect features including Peer-to-Peer and

GPUDirect for Video

A full Kepler GK110 implementation includes 15 SMX units and six 64-bit memory controllers. Different products will use different configurations of GK110. For example, some products may deploy 13 or 14 SMXs. Key features of the architecture that will be discussed below in more depth include:

- The new SMX processor architecture
- An enhanced memory subsystem, offering additional caching capabilities, more bandwidth at each level of the hierarchy, and a fully redesigned and substantially faster DRAM I/O implementation.
- Hardware support throughout the design to enable new programming model capabilities

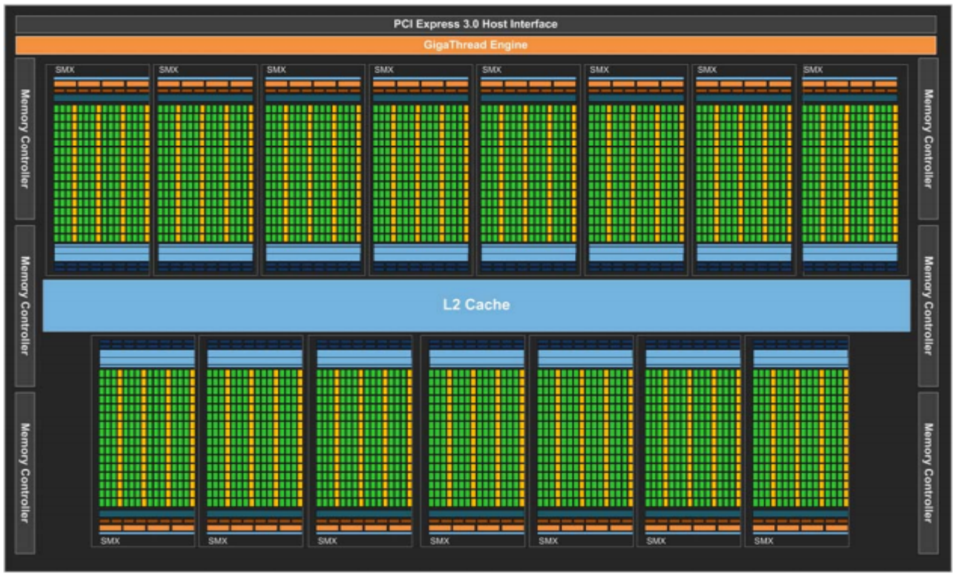


Figure 3.4: Kepler GPU Device Block Diagram

Streaming Multiprocessor (SMX) Architecture

Kepler GK110's new SMX introduces several architectural innovations that make it not only the most powerful multiprocessor we've built, but also the most programmable and power-efficient.

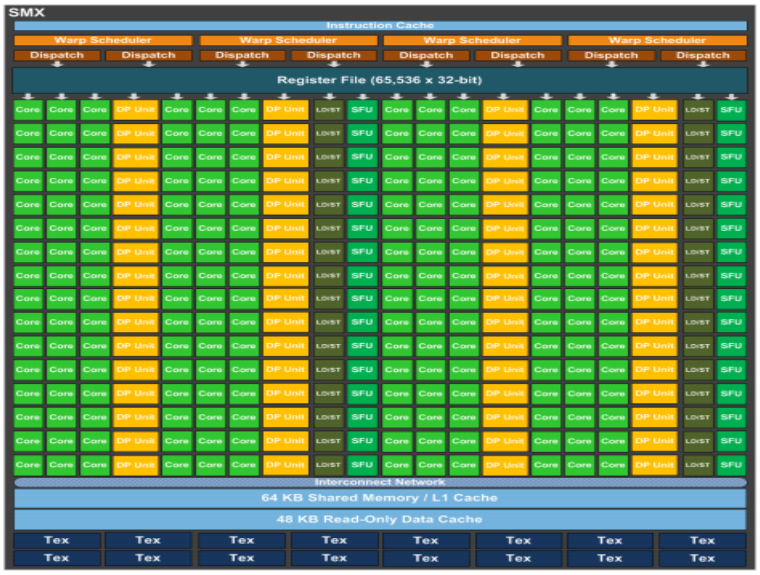

SMX
Instruction Cache
Warp Scheduler
Dispatch
Register File (65,536 x 32-bit)
Core
DP Unit
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
48 KB Read-Only Data Cache
Tex

Figure 3.5: Kepler SM Architecture

SMX Processing Core Architecture

Each of the Kepler GK110 SMX units feature 192 single-precision CUDA cores, and each core has fully pipelined floating-point and integer arithmetic logic units. In addition to these, Kepler GPU significantly increases the double precision performance which is one of the major application performance consideration in high performance computing applications. Kepler GPUs included 8x the number of SFUs (Special Function Units) in comparison to Fermi GPU that significantly improves the fast approximate transcendental operations.

The chip area design of the Kepler targets performance per watt and many optimizations have been made to benefit for both area and power. One of the power optimization is to use a larger number of processing cores that are running at a less power-hungry GPU clock.

Quad Warp Scheduler

With 4 warp schedulers and 8 instruction dispatch units, Kepler allows four warps with two independent instructions per warp to be dispatched per GPU cycle. Unlike Fermi, the instruction pair can included both single and double precision instructions.

New ISA Encoding: 255 Registers per Thread

Each thread in kepler can access four times more registers up to 255 registers per thread. This reduces register spilling and gives substantial speedups in terms of application performance. For example, QCD (Quantum ChromoDynamics) application in QUDA library gains performance increases up to 5.3x by utilizing more registers per thread and reduced register spilling to local memory.

Shuffle Instruction

Kepler introduces a new set of shuffle instructions to share data within a warp which was previously done by using shared memory only with separate load/store operations. Using shuffle instructions, threads within a warp can read values from other threads in the warp in all possible permutations. Figure 3.6 shows the different shuffle subsets available as CUDA intrinsics including indexed, next-thread (up and down), and butterfly style permutations transfers among threads in a warp.

Shuffle instructions perform store-and-load operations in a single step that obtains significant performance improvement over shared memory. It also reduces the required amount of shared memory per thread block.

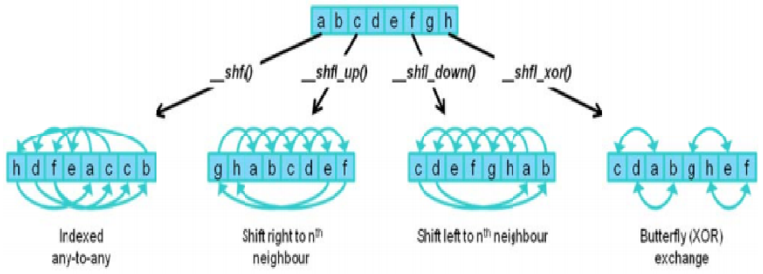


Figure 3.6: Kepler Shuffle Instructions

Atomic Operations

Read and write on shared data structures by parallel threads can cause race condition in results. This can be avoided by using atomic memory operations on shared data structures such as add, min, max and compare-and-swap are some of the examples of atomic operations. Each thread performs read, modify and write in these operations on a shared data without interruption by other threads. Most common uses of atomic operations are in parallel sort, reduction, and building data structures in parallel without using locks to avoid thread serialization.

Atomic operation throughput on Kepler GK110 is substantially improved compared to the Fermi generation. Atomic operation throughput to a common address is improved by 9x, to one operation per clock. Atomic operation throughput to independent addresses is also significantly accelerated and logic to handle address conflicts has been made more efficient. Atomic operations can often be processed at rates similar to generic load operations. This speed increase makes atomics fast enough to use frequently within kernel inner loops, eliminating explicit reduction passes that were previously required to consolidate results.

Kepler GK110 also introduces additional native support for 64-bit atomic operations. In addition to atomicAdd, atomicCAS, and atomicExch (supported by Fermi and Kepler GK104), GK110 supports native:

- atomicMin
- atomicMax
- atomicAnd
- atomicOr
- atomicXor

Other atomic operations which are not supported natively (for example 64-bit floating point atomics) may be emulated using the compare-and-swap (CAS) instruction.

Texture Improvements

The GPU's dedicated hardware Texture units are a valuable resource for compute programs with a need to sample or filter image data. The texture throughput in

Kepler is significantly increased compared to Fermi each SMX unit contains 16 texture filtering units, a 4x increase vs the Fermi GF110 SM.

In addition, Kepler changes the way texture state is managed. In the Fermi generation, for the GPU to reference a texture, it had to be assigned a “slot” in a fixed-size binding table prior to grid launch. The number of slots in that table ultimately limits how many unique textures a program can read from at run time. Ultimately, a program was limited to accessing only 128 simultaneous textures in Fermi.

With bindless textures in Kepler, the additional step of using slots isn't necessary: texture state is now saved as an object in memory and the hardware fetches these state objects on demand, making binding tables obsolete. This effectively eliminates any limits on the number of unique textures that can be referenced by a compute program. Instead, programs can map textures at any time and pass texture handles around as they would any other pointer.

Kepler Memory Subsystem L1, L2, ECC

Kepler's memory hierarchy (see Figure 3.7) is organized similarly to Fermi. The Kepler architecture supports a unified memory request path for loads and stores, with an L1 cache per SMX multiprocessor. Kepler GK110 also enables compiler-directed use of an additional new cache for read-only data, as described below.

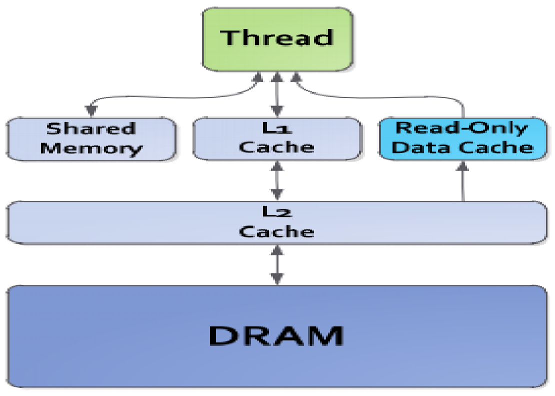


Figure 3.7: Kepler Memory Hierarchy

64 KB Configurable Shared Memory and L1 Cache

In the Kepler GK110 architecture, as in the previous generation Fermi architecture, each SMX has 64 KB of on-chip memory that can be configured as 48 KB of Shared memory with 16 KB of L1 cache, or as 16 KB of shared memory with 48 KB of L1 cache. Kepler now allows for additional flexibility in configuring the allocation of shared memory and L1 cache by permitting a 32KB / 32KB split between shared memory and L1 cache. To support the increased throughput of each SMX unit, the shared memory bandwidth for 64b and larger load operations is also doubled compared to the Fermi SM, to 256B per core clock.

48KB ReadOnly Data Cache

In addition to the L1 cache, Kepler introduces a 48KB cache for data that is known to be read-only for the duration of the function. In the Fermi generation, this cache was accessible only by the Texture unit. Expert programmers often found it advantageous to load data through this path explicitly by mapping their data as textures, but this approach had many limitations.

In Kepler, in addition to significantly increasing the capacity of this cache along with the texture horsepower increase, we decided to make the cache directly accessible to the SM for general load operations. Use of the read-only path is beneficial because it takes both load and working set footprint off of the Shared/L1 cache path. In addition, the Read-Only Data Cache's higher tag bandwidth supports full speed unaligned memory access patterns among other scenarios.

Use of this path is managed automatically by the compiler - access to any variable or data structure that is known to be constant through programmer use of the C99-standard "const __restrict" keyword will be tagged by the compiler to be loaded through the Constant Data Cache.

Improved L2 Cache

The Kepler GK110 GPU features 1536KB of dedicated L2 cache memory, double the amount of L2 available in the Fermi architecture. The L2 cache is the primary point of data unification between the SMX units, servicing all load, store, and texture requests and providing efficient, high speed data sharing across the GPU. The L2 cache on Kepler offers up to 2x of the bandwidth per clock available in Fermi. Algorithms for which data addresses are not known beforehand, such as physics solvers, ray tracing, and sparse matrix multiplication especially benefit from the cache hierarchy. Filter and convolution kernels that require multiple SMs to read the same data also benefit.

Memory Protection Support

Like Fermi, Kepler's register files, shared memories, L1 cache, L2 cache and DRAM memory are protected by a Single-Error Correct Double-Error Detect (SECDED) ECC code. In addition, the Read-Only Data Cache supports single-error correction through a parity check; in the event of a parity error, the cache unit automatically invalidates the failed line, forcing a read of the correct data from L2.

ECC checkbit fetches from DRAM necessarily consume some amount of DRAM bandwidth, which results in a performance difference between ECC-enabled and ECC-disabled operation, especially on memory bandwidth-sensitive applications. Kepler GK110 implements several optimizations to ECC checkbit fetch handling based on Fermi experience. As a result, the ECC on-vs-off performance delta has been reduced by an average of 66%, as measured across our internal compute application test suite.

Dynamic Parallelism

In a hybrid CPU-GPU system, enabling a larger amount of parallel code in an application to run efficiently and entirely within the GPU improves scalability and performance as GPUs increase in perf/watt. To accelerate these additional parallel portions of the application, GPUs must support more varied types of parallel workloads.

Dynamic Parallelism is a new feature introduced with Kepler GK110 that allows the GPU to generate new work for itself, synchronize on results, and control the scheduling of that work via dedicated, accelerated hardware paths, all without involving the CPU.

Fermi was very good at processing large parallel data structures when the scale and parameters of the problem were known at kernel launch time. All work was launched from the host CPU, would run to completion, and return a result back to the CPU. The result would then be used as part of the final solution, or would be analyzed by the CPU which would then send additional requests back to the GPU for additional processing.

In Kepler GK110 any kernel can launch another kernel, and can create the necessary streams, events and manage the dependencies needed to process additional work without the need for host CPU interaction. This architectural innovation makes it easier for developers to create and optimize recursive and data-dependent execution patterns, and allows more of a program to be run directly on GPU. The system CPU can then be freed up for additional tasks, or the system could be configured with a less powerful CPU to carry out the same workload. Figure 3.8 shows the advantage of dynamic parallelism in kepler in comparison of Fermi repetitive kernel invocation.

Dynamic Parallelism allows more varieties of parallel algorithms to be implemented on the GPU, including nested loops with differing amounts of parallelism, parallel teams of serial control-task threads, or simple serial control code offloaded to the GPU in order to promote data-locality with the parallel portion of the ap-

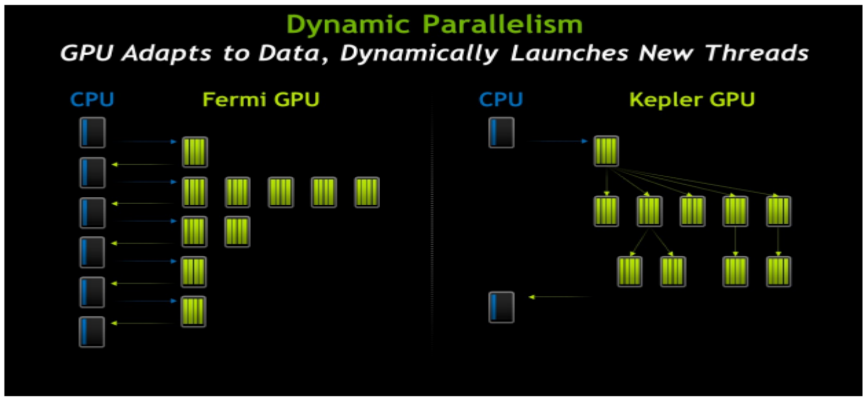


Figure 3.8: Fermi and Kepler Recursive Kernel Invocation Comparison

plication.

Because a kernel has the ability to launch additional workloads based on intermediate, on-GPU results, programmers can now intelligently load-balance work to focus the bulk of their resources on the areas of the problem that either require the most processing power or are most relevant to the solution.

Hyper Q

Hyper Q increases the total number of connections (work queues) between the host and CUDA Work Distributor (CWD) logic in the GPU by allowing 32 simultaneous, hardware - managed connections (compared to the single connection available with Fermi). Hyper - Q is a flexible solution that allows connections from multiple CUDA streams, from multiple Message Passing Interface (MPI) processes, or even from multiple threads within a process. Application that previously encountered false serialization across tasks, thereby limiting GPU utilization, can see up to a 32x performance increase without changing any existing code.

Each CUDA stream is managed within its own hardware work queue (see Figure 3.9), inter-stream dependencies are optimized, and operations in one stream will no longer block other streams, enabling streams to execute concurrently without needing to specifically tailor the launch order to eliminate possible false dependencies.

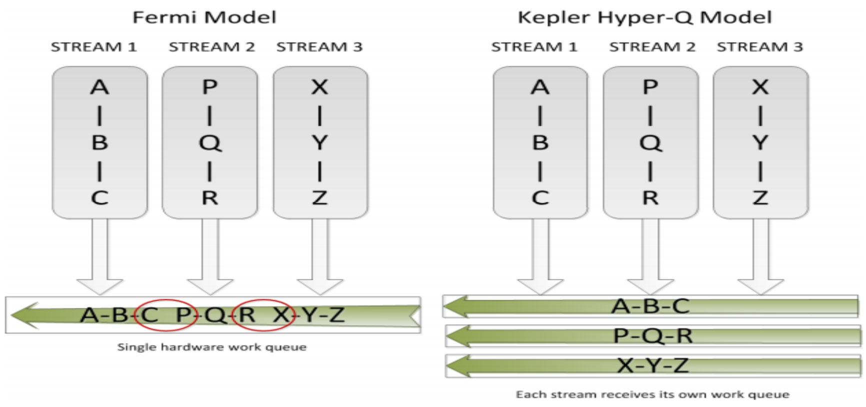


Figure 3.9: Multiple Stream Execution in both Fermi and Kepler

Grid Management Unit - Efficiently Keeping the GPU Utilized

New features in Kepler GK110, such as the ability for CUDA kernels to launch work directly on the GPU with Dynamic Parallelism, required that the CPU-to-GPU workflow in Kepler offer increased functionality over the Fermi design. On Fermi, a grid of thread blocks would be launched by the CPU and would always run to completion, creating a simple unidirectional flow of work from the host to the SMs via the CUDA Work Distributor (CWD) unit. Kepler GK110 was designed to improve the CPU-to-GPU workflow by allowing the GPU to efficiently manage both CPU- and CUDA-created workloads. Figure 3.10 shows the workflows for both Fermi and Kepler.

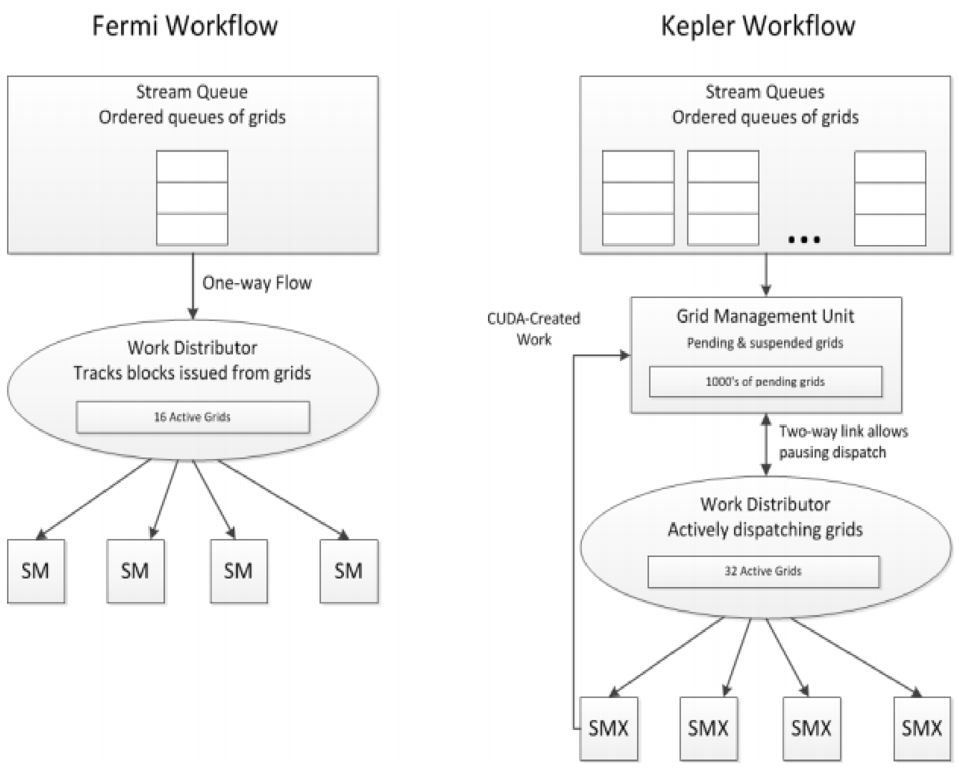


Figure 3.10: Fermi and Kepler Workflow

We discussed the ability of the Kepler GK110 GPU to allow kernels to launch work directly on the GPU, and it's important to understand the changes made in the Kepler GK110 architecture to facilitate these new functions. In Kepler, a grid can be launched from the CPU just as was the case with Fermi, however new grids can also be created programmatically by CUDA within the Kepler SMX unit. To manage both CUDA-created and host-originated grids, a new Grid Management Unit (GMU) was introduced in Kepler GK110. This control unit manages and prioritizes grids that are passed into the CWD to be sent to the SMX units for execution.

The CWD in Kepler holds grids that are ready to dispatch, and it is able to dispatch 32 active grids, which is double the capacity of the Fermi CWD. The

Kepler CWD communicates with the GMU via a bi-directional link that allows the GMU to pause the dispatch of new grids and to hold pending and suspended grids until needed. The GMU also has a direct connection to the Kepler SMX units to permit grids that launch additional work on the GPU via Dynamic Parallelism to send the new work back to GMU to be prioritized and dispatched. If the kernel that dispatched the additional workload pauses, the GMU will hold it inactive until the dependent work has completed.

NVIDIA GPUDirect™

When working with a large amount of data, increasing the data throughput and reducing latency is vital to increasing compute performance. Kepler GK110 supports the RDMA feature in NVIDIA GPUDirect, which is designed to improve performance by allowing direct access to GPU memory by third-party devices such as IB adapters, NICs, and SSDs. When using CUDA 5.0, GPUDirect provides the following important features:

- Direct memory access (DMA) between NIC and GPU without the need for CPU-side data buffering.
- Significantly improved MPISend/MPIRecv efficiency between GPU and other nodes in a network.
- Eliminates CPU bandwidth and latency bottlenecks
- Works with variety of third-party network, capture, and storage devices

Applications like reverse time migration (used in seismic imaging for oil & gas exploration) distribute the large imaging data across several GPUs. Hundreds of GPUs must collaborate to crunch the data, often communicating intermediate results. GPUDirect (see Figure 3.11) enables much higher aggregate bandwidth for this GPU-to-GPU communication scenario within a server and across servers with the P2P and RDMA features.

Kepler GK110 also supports other GPUDirect features such as Peer-to-Peer and GPUDirect for Video.

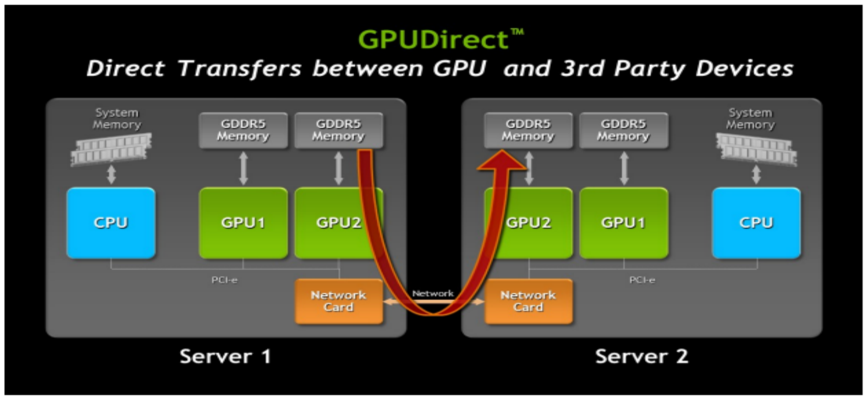


Figure 3.11: Kepler GPUDirect

3.2 GPU Execution Model

Figure 3.2 shows the block diagram of SM in Fermi and Figure 3.5 shows the block diagram of SM in Kepler. GM is linked to the GPU device through a very large data path of 320-bits wide. Through such a bus width, ten consecutive 32-bits (4 bytes) words can be fetched from global memory in a single cycle. The on-chip memory resource includes register files (64K or more per SM, see Table

3.1), shared memory (48KB or more per SM). To hide the long off-chip memory access latency, a high number of threads are supported to run concurrently. The threads are grouped in blocks which will be scheduled to SMs dynamically on the availability of each SM. These threads follow the single-program multiple-data (SPMD) program execution model. Within a block, threads are grouped in 32-threads instruction called warps, where each warp is being executed in the single-instruction multiple-data (SIMD) manner. A warp takes multiple cycles for computation instructions due to the limited number of functional units (SPs) within SM.

Feature	Quadro FX 7000	Tesla K20c
GPU Architecture	Fermi	Kepler
CUDA Driver / Runtime Version	5.0 / 5.0	5.5 / 5.5
CUDA Capability	2.0	3.5
Global Memory Size	4096 Mbytes	4800 Mbytes
Total CUDA Cores	16 (SM) x 32 (SP/SM) = 512	13 (SM) x 192 (SP/SM) = 2496
Shared Memory Size	49152 bytes	49152 bytes
Max Thread / SM	1536	2048
Max Thread / Block	1024	1024
Threads/Warp	32	32
Max Warps / SM	48	64
Max Thread Blocks / SM	8	16
32-bit Registers / SM	32768	65536
Max Registers / Thread	63	255
Hyper - Q	No	Yes
Dynamic Parallelism	No	Yes

Table 3.1: GPU Specifications

Figure 3.12 shows the execution hierarchy of a typical kernel function on a device. Each kernel initiates a set of blocks defined by the programmer as grid dimension with number of threads to be executed within each block while invoking the device kernel function. Now, the block scheduler dynamically schedules each thread block to one SM based on the availability of resources within SM while individual threads will be distributed among multiple SPs within the SM. An SM can handle at most 16 blocks at a time. Also, the possible number of concurrent blocks per SM depends on the number of warps per block, number of registers per block, and the shared memory usage per block.

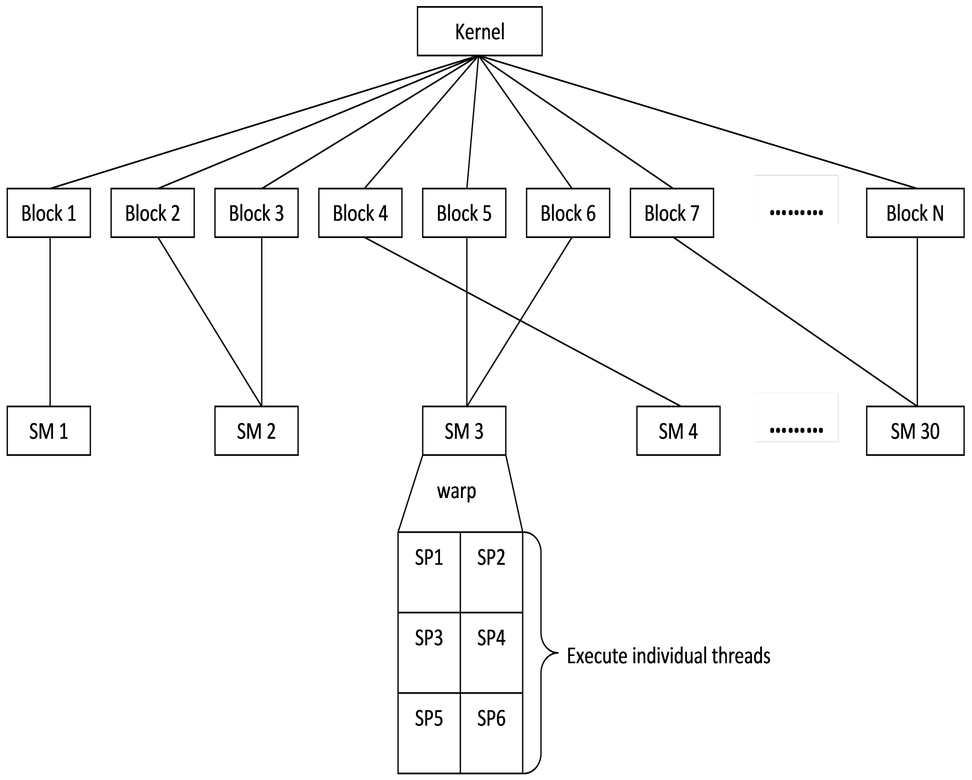


Figure 3.12: Kernel Execution Hierarchy

The number of simultaneous blocks are dependent upon shared memory, number of registers per block and warp per block. Fine-grained, data-parallel threads are the fundamental means of parallel execution. When a kernel is invoked, grid of threads is launched. Each thread that executes the kernel is given a unique thread ID. Threads in each block cooperate with each other and have access to shared memory, the cooperation between threads in different blocks are not possible.

GM is partitioned into segments of size equal to 32, 64 or 128 bytes and aligned to this size. The elements in one segment can be accessed by a single memory transaction. By considering the largest segment size of 128 bytes and also the data path of 512 bits, the compiler issues a single load/store instruction for 16 consecutive elements accessed by 16-threads (half warp) to reduce the number of memory transactions of global memory. So, the performance of memory transfers can incredibly be improved through the use of coalesced global memory accesses that is accessing a regular pattern of consecutive elements by a half warp (16 threads) based on some conditions. Therefore, if SPs are kept busy executing through warp switching then the whole transfer between GM and ShM is hidden by some execution which implies that the parallel program time does not account for such an expensive memory transfer. Since, shared memory is very small in size so we have to perform some loop transformation such as loop tiling, a mechanism to adjust loop execution to match with underlying machine or memory system, to make the availability of enough data for the active warp per SM.

Coalesced Global Memory Accesses

Global memory is the slowest memory on the GPU. When one begins to work with GPGPU, the parallel processing benefits can be incredibly beneficial, if you know how to work with coalesced memory accesses that is accessing a bank of memory by all threads in a group in one cycle. In order to achieve the most possible speedup, programmer has to incorporate the following in writing CUDA kernels:

1. Thinking about the computation in a data parallel fashion.

2. Transferring working data into the shared memory.

3. Scrutinizing how the code performs global memory accesses.

The reasons for the above considerations are that the shared memory is so much faster at reading and writing than global memory, and the memory module in modern GPUs can perform concurrent reads to sequential global memory positions for an entire thread group.

Conditions to achieve coalesced access

The simultaneous global memory accesses by each thread of a half-warp (16 threads) during the execution of a single read or write instruction will be coalesced into a single access if:

- The size of the memory element accessed by each thread is either 4, 8, or 16 bytes
- The elements form a contiguous block of memory
- The N^{th} element is accessed by the N^{th} thread in the half-warp, does not affect if any thread in between not accessing the global memory that is divergent warp.
- The address of the first element is aligned to 16 times the element's size

If any of the above condition is not satisfied then memory access will not be coalesced, increases memory accesses instead of single access. Figure 3.13 and

Figure 3.14, shows the two of the possible coalesced memory accesses satisfying the above conditions.

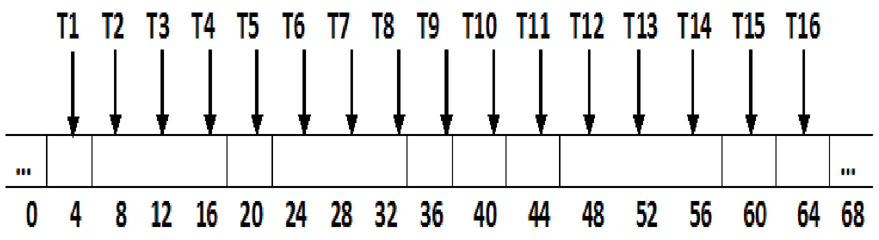


Figure 3.13: Coalesced Float Memory Access

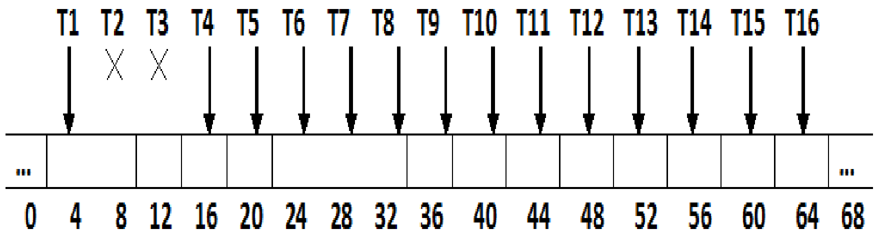


Figure 3.14: Coalesced Float Memory Access (divergent warp)

Figure 3.15-Figure 3.18 shows the access patterns which fails the mechanism of memory coalescing.

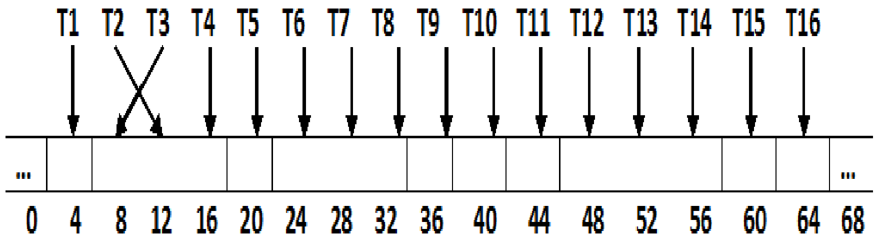


Figure 3.15: Non-Sequential Float Memory Access

3.3 Synchronization within SM and across SMs

GPUs are typically mapped well only to data-parallel or task-parallel applications which require relatively minimal communication between streaming multiproces-

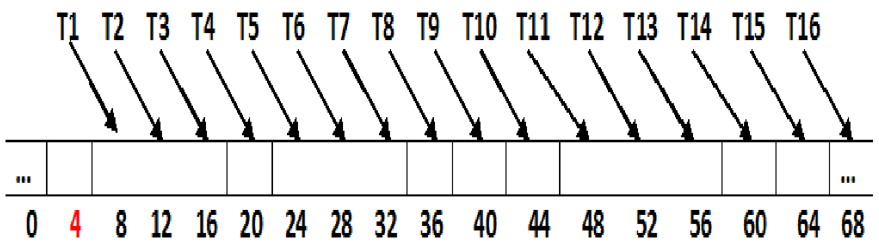


Figure 3.16: Misaligned Starting Address

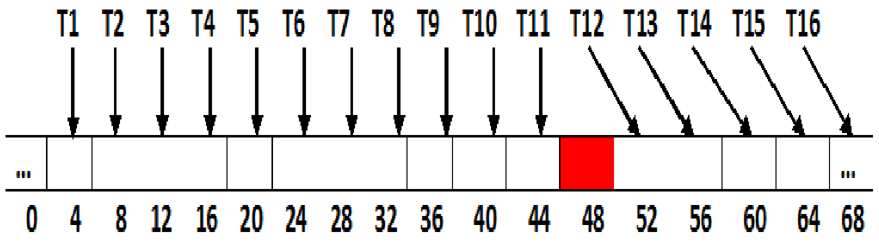


Figure 3.17: Non-Contiguous Float Memory Access

sors (SMs) on the GPU during their execution [45, 46, 47]. This tendency is essentially due to the lack of support for communication between SMs. Thread blocks cannot communicate through the per-SM shared memory. Although, such communication can take place through the GPU's global memory that needs barrier synchronization over SMs in order to complete the communication task between SMs. On the other hand, CUDA provides a synchronization function __syncthreads() to synchronize the execution of different threads within a block. This is due to the fact that threads within a block are executed by the same SMs

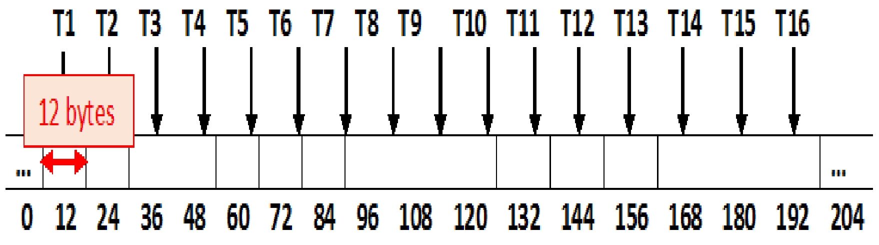


Figure 3.18: Non-Coalesced Float3 Memory Access

which can access same block of shared memory and execute same instructions.

To alleviate inter-block communication problem, CPU-based synchronization algorithms are proposed. Such an algorithm handles inter-block GPU communication by launching the kernel many times [48, 49]. This approach incurs significant overhead [50]. To eliminate this overhead, there are many synchronization algorithms (lock-based) implemented using global memory and atomic operations where a mutually exclusive (mutex) variable controls the execution of different blocks on SMs. Once a block finishes its computation on an SM, it atomically increments a mutex variable. Only after all thread blocks finish their computation the mutex variable will be equal to the target value and the barrier complete. However, atomic operations pose a fundamental parallelization problem. If multiple threads are required to perform the atomic operation at the same time, they are serialized, since only one thread is able to perform the operation and others must wait. To overcome this problem, lock-free synchronization methods are proposed. These algorithms utilize arrays of variables instead of using a single mutex variable and eliminate the need for different blocks to contend for the single mutex variable. By eliminating the single mutex variable, the need for atomic addition is removed. Using these methods, some threads (usually the first thread) from each block control the execution of the synchronization code in different blocks while the intra-block synchronization is accomplished by synchronizing the threads within the block with the existing barrier function __syncthreads().

Following sections explain some of the inter-block synchronization mechanisms found in literature. These approaches can be classified into three categories: (1) CPU-based synchronization, (2) Lock-based synchronization and (3) lock-free synchronization.

3.3.1 CPU-based synchronization

This is the simplest approach recommended by NVIDIA [51] for inter-block synchronization by exiting and re-entering the kernel that is considered as an implicit synchronization. This is done by dividing the kernel into multiple kernels based on the requirements of inter-block synchronization among the instructions. Figure 3.19 shows the flowchart of the CPU-based synchronization.

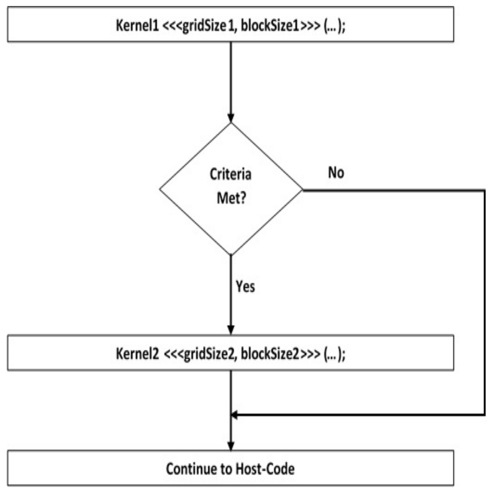

Kernel1 <<<gridSize1, blockSize1>>> (...);
Criteria Met?
No
Yes
Kernel2 <<<gridSize2, blockSize2>>> (...);
Continue to Host-Code

Figure 3.19: CPU Based Synchronization

3.3.2 Lock-based Synchronization

Lock-based synchronization approaches use atomic operations on global variables defined in the global memory. We have explored two versions for lock-based synchronization. The first version (Lock-based-v1) is explained in [52]. This approach allows only one block to continue while the other blocks terminates. So, this approach can be useful in only one - level reduction in which only one block will do the second iteration of reduction and related problems. While for applications such as tree reduction and Jacobi iterative solver requires all blocks to continue with the multiple iterations for which this approach is applicable. On the other hand, the second version (Lock-based-v2) which is proposed in [50] allows all blocks to continue. The basic idea of the two versions are shown in Figure 3.20 and Figure 3.21. When all threads of a block finish their work, the first thread of each block atomically increments the global variable gMutex and then checks its value. In Figure 3.20, the last block increments gMutex will continue while others terminate. However, in Figure 3.21 all blocks will wait until the value of the gMutex becomes equal kxN where k is the iteration number and N is the number of blocks. Once gMutex becomes equal to k x N than all blocks will continue to next iteration.

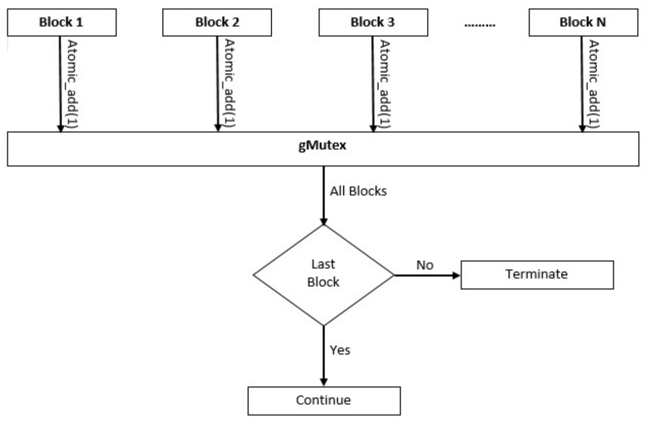


Figure 3.20: Lock-Based-V1 Synchronization

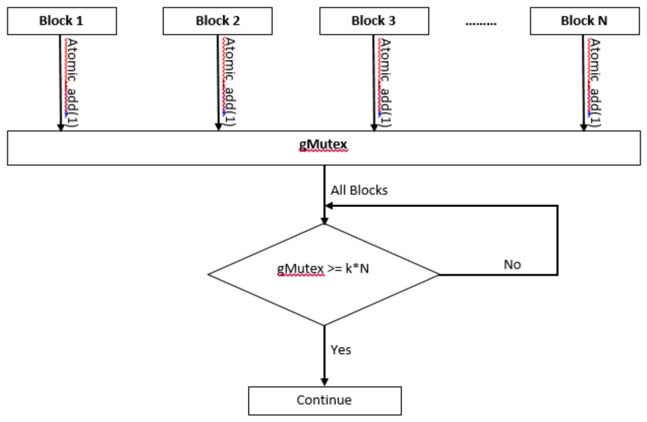


Figure 3.21: Lock-Based-V2 Synchronization

3.3.3 Lock-free Synchronization

In lock-free synchronization, each block uses one or more global variables to coordinate the synchronization requests from various blocks. We have explored two versions for lock-free synchronization. The first one (Lock-free-v1) is based on the work in [53] while the second (Lock-free-v2) is based on the work in [50]. Figure 3.22 and Figure 3.23 show the main idea in these versions. The first approach (Figure 3.22) uses only one array named Ain for synchronization. The first thread of each block increments its corresponding location in Ain array. After that it continuously checks whether others' location of Ain have been set to k where k is the iteration number. When it finds all locations are set to k, the threads block continue their work. The second approach (Figure 3.23) uses two arrays, named Ain and Aout, of length N for synchronization. When all threads of a block finish their work, the first thread of each block increments its location in the Ain array. Then, the first N threads of the first block in parallel check whether all blocks have written to their corresponding location in the Ain array. If so, these N threads write in parallel to the Aout array to inform other threads that the threads of this block have reached the synchronization point. Meanwhile, the first thread of each block continuously checks its location in the Aout array until the value is set to k.

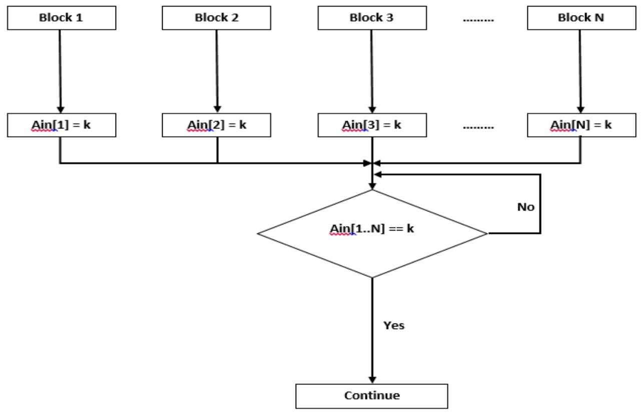


Figure 3.22: Lock-Free-V1 Synchronization

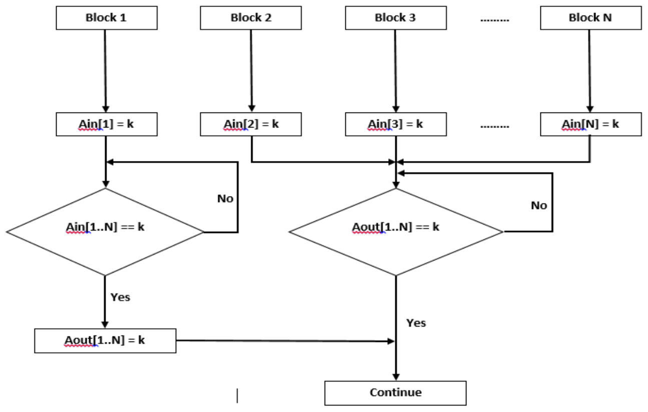


Figure 3.23: Lock-Free-V2 Synchronization

3.4 CUDA Programming Framework and Compiler

GPUs are designed as numerical computing engines, and they will not perform well on some tasks on which CPUs are designed to perform well. So, the most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs. This is why the CUDA (Compute Unified Device Architecture) programming model is designed to support joint CPU/GPU execution of an application.

CUDA is a parallel computing platform designed by NVIDIA for graphics processor units (GPUs). A CUDA program is a unified source code encompassing both the host and the device code. It consists of one or more phases that are executed on either the host (CPU) or a device that is a GPU. The phases that exhibit rich amount of data parallelism are implemented in the device code. The NVIDIA C compiler (nvcc) separates the two during the compilation process. The host code is straight ANSI C code; it is further compiled with the host's standard C compilers and runs as an ordinary CPU process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, called kernels, and their associated data structures [22]. Table 3.2 and Table 3.3 lists the different functions and variable declarations within the CUDA Program respectively. The device code is typically further compiled by the nvcc and executed on a GPU device.

In CUDA memory model [22], threads can access data in private local memory,

Function Declaration	**Executed on**	**Only Callable from**
__device__ float DeviceFunc()	Device	Device
__global__ void KernelFunc()	Device	Host
__host__ float HostFun()	Host	Host

Table 3.2: CUDA Function Declarations

	Memory	**Scope**	**Lifetime**
__local__ int LocalVar;	Local	Thread	Thread
__shared__ int SharedVar;	Shared	Block	Block
int GlobalVar;	Global	Grid	Application
__constant__ int Con-stantVar;	Constant	Grid	Application

Table 3.3: CUDA Device Variable Declarations

shared memory and global memory. The threads also have access to texture and constant memory. The shared memory is the only memory that is on-chip memory; this memory is visible to all threads within a block. The global memory is off-chip memory and can be accessed by the host and all threads. The local memory is the maximum memory allocated per thread. Table 3.4 summarizes the memory hierarchy in the GPU. A CUDA program that implemented in the device code exhibits rich amount of data parallelism. Invoking the kernel will launch a grid of blocks, group of threads. The dimension of the grid and the number of threads in a block can be determined by the programmer.

Memory	**Location**	**Cache**	**Accessibility**	**Scope**
Global	Off Chip	No	R/W	CPU + All threads
Texture	Off Chip	Yes	R	CPU + All threads
Constant	Off Chip	Yes	R	CPU + All threads
Shared	On Chip	-	R/W	All threads (in a block)
Local	Off Chip	No	R/W	Per Thread
Register	On Chip	No	R/W	Per Thread
Data Cache (Kepler Only)	On Chip	-	R	All blocks in SM

Table 3.4: GPU Memory Hierarchy

The CUDA programming model assumes that the CUDA threads execute on a physically separate device that operates as a coprocessor to the host running the C program, it also assumes that both host and device maintain their own memory space in DRAM, referred to as host memory and device memory. Therefore, a program manages the global, constant and texture memory spaces visible to kernels through calls to the CUDA runtime. This includes device memory allocation and deallocation as well as data transfer between host and device memory.

At its core, CUDA provides three key abstractions:

- A hierarchy of thread groups
- Shared memories
- Barrier synchronization

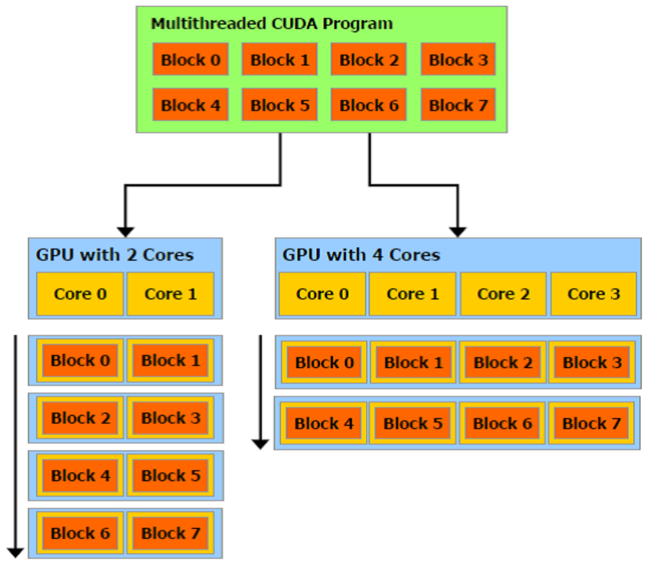


Figure 3.24: Block Assignment to Different Cores (Automatic Scalability)

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available processor cores, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of processor cores as illustrated by Figure 3.24 (Courtesy: NVIDIA), and only the runtime system needs to know the physical processor count.

3.5 CUDA APIs

CUDA C extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the __global__ declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new <<< ... >>> execution configuration syntax. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable.

For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (Dx, Dy), the thread ID of a thread of index (x, y) is (x + y Dx); for a three-dimensional block of size (Dx, Dy, Dz), the thread ID of a thread of index (x, y, z) is (x + y Dx + z Dx Dy).

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On Quadro FX7000 (Fermi) and Tesla K20c (Kepler) GPUs, a thread block may contain up to 1024 threads.

However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

Blocks are organized into a one-dimensional or two-dimensional grid of thread blocks as illustrated by Figure 3.25 (Courtesy: NVIDIA). The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed. Each block within the grid can be identified by a one-dimensional or two-dimensional index accessible within the kernel through the built-in blockIdx variable. The dimension of the thread block is accessible within the kernel through the built-in blockDim variable.

Threads within a block can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the __syncthreads() intrinsic function; __syncthreads() acts as a barrier at which all threads in the block must wait before any is allowed to proceed.

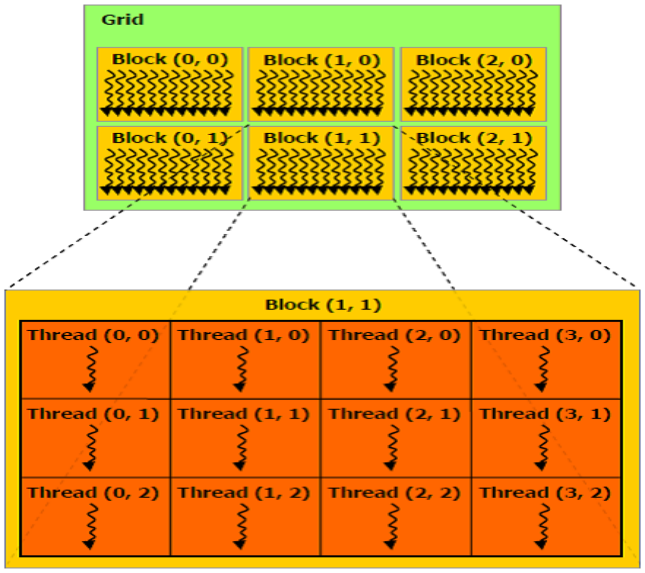


Figure 3.25: Gird of Thread Blocks

3.6 Profiling and Debugging

Debugging in software systems is a methodological process of finding errors in a system. It is intuitive that the complexity of the process of debugging a system increases with the complexity of the system being debugged. Debugging parallel and multi-threaded software in general is considered to be more complex than debugging single threaded software; as the execution of the program may not be consistent.

The key challenges while programming accelerators can be summarized as follows:

- Coordinating CPU code + device code
- Understanding what is going on in each kernel
- Exceptions
- Understanding memory usage
- Understanding performance characteristics

This section surveys available products that are suitable overcoming these challenges on GPUs: these products are used for debugging and analyzing the performance of many-core programs.

Debuggers are concerned with finding causes and use cases that could cause incorrectness in expected program output and behavior.

A typical work-flow for working with efficient parallel programs is to have an optimization phase:

- Look at how well new code behaves
- Use available toolsets:
  - Debugger:
    * Allinea DDT
    * Totalview

    - Performance analysis tools:
        * Vampir

- Profiling:
    - Collect aggregated information (time, counts, ...)
    - Global or per process/thread
- Tracing:
    - Save individual event records with precise timestamps per process/thread
    - Add event specific information represented as Timeline

The following sections list some of these tools that are suitable for working with Nvidia accelerators.

3.6.1 Allinea DDT

Allinea DDT provides application developers with a single tool that can debug hybrid CUDA, OpenMP and MPI applications on a single workstation or GPU cluster [54]. It is comprehensive and scalable resource for debugging CPU and GPU threads on the same screen.

Key Features

- Languages supported:

  - NVIDIA CUDA
  - OpenACC Directives
  - HMPP from Caps-Enterprise
  - PGI Accelerator
  - CRAY compiler

- Debugging GPU and CPU
  - Browse source, examine variables, control processes and threads
  - View all threads in parallel stack view with click for thread selection
  - Set breakpoints or stop on kernel launch
  - Control CUDA warps
  - DDT displays variables using Smart Highlighting for CUDA storage classes

    Built in to natively support the programming environment

- Full MPI support - view GPU and CPU threads simultaneously over many nodes

Educational License

The Allinea DDT CUDA Education pack has been designed to save people time whether they are:

- already teaching a course in parallel programming via CUDA, which should include dynamic debugging;

- developing a course in parallel programming via CUDA; or
- Just starting to think about developing a course in parallel programming via CUDA.

The pack includes:

1. Licence pack
   - Eleven Workstation CUDA scalar licences valid until 31st July 2014
2. Resources given
   - Introduction to Allinea Software
   - PGI white paper (by Allinea Software)
   - Debugging CUDA white paper (by Allinea Software)
   - CUDA-GDB paper (by NVIDIA)
   - Manycore systems white paper (by Allinea Software and CAPS) Suggested reading list
   - Pre-recorded webinar: "Allinea DDT and CUDA: Develop new efficient software"
3. Teaching material
   - Lecture - Introduction to CUDA debugging
   - Hands-on training, which includes walkthrough examples and exercises
   - Sample programs

4. Coming soon : In addition to above, the following will launched shortly and provided to all subscribers at no extra charge:

    - Assignment questions
    - Project suggestions
    - Exam questions
    - Quiz questions

Process Groups

With DDT, the user can change the debugger to focus on a single process or group of processes. The user then can step through the code, setting breakpoints only for a given process. If Focus on current Group is chosen then the entire group of processes will advance when stepping forward in a program and a breakpoint will be set for all processes in a group.

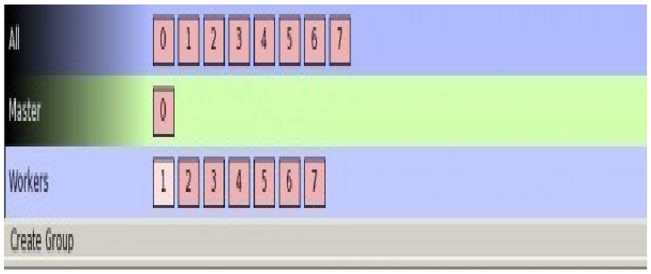
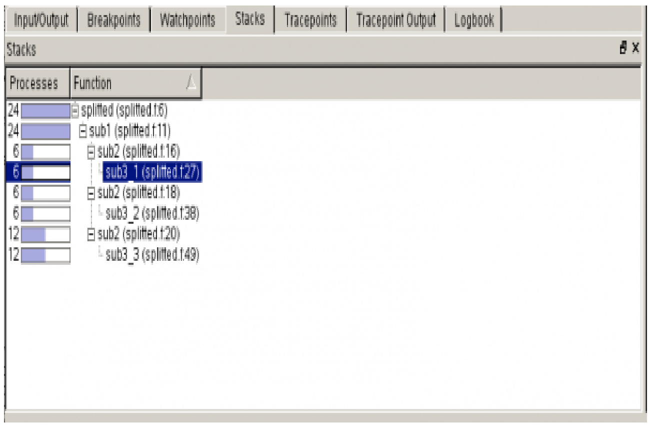


Figure 3.26: DDT Process Groups

Similary, when Focus on current Thread is chosen, then all actions are for an OpenMP thread. DDT doesn't allow to create a thread group. However, one can click the Step Threads Together box to make all threads to move together inside a parallel region. In the image shown above 3.26, this box is grayed out simply because the code is not an OpenMP code.

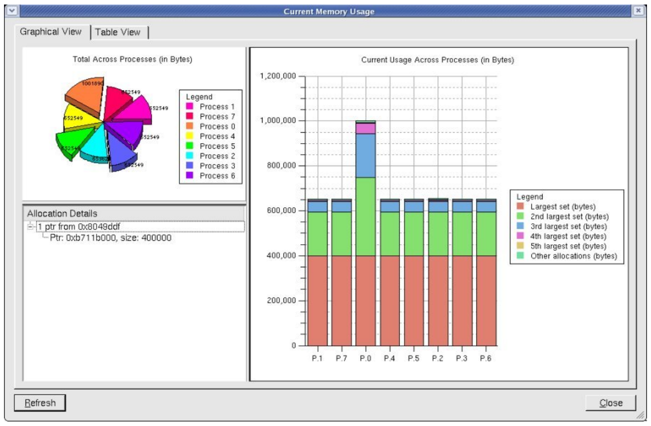
Parallel Stack View

Parallel stack view is a feature which should help users debug at high concurrencies which allows the user to see the position of all processors in a code at the same time from the main window. A program is displayed as a branching tree with the number and location of each processor at each point. Instead of clicking through windows to determine where each processor has stopped, the Parallel Stack View presents a quick overview which easily allows users to identify stray processes. Users can also create sub-groups of processors from a branch of the tree by right clicking on the branch. A new group will appear in the Process Group Window at the top of the GUI.



Processes	Function
24	splitted (splitted.f:6)
24	sub1 (splitted.f:11)
6	sub2 (splitted.f:16)
6	sub3_1 (splitted.f:27)
6	sub2 (splitted.f:18)
6	sub3_2 (splitted.f:38)
12	sub2 (splitted.f:20)
12	sub3_3 (splitted.f:49)

Figure 3.27: DDT Stack View



```
1 ptr from 0x8049ddf
  Ptr: 0xb711b000, size: 400000
```

Figure 3.28: DDT Memory Usage

Memory usage can be analysed

3.6.2 TotalView Debugger

TotalView is a dynamic source code and memory debugger for C, C++ and FORTRAN applications. TotalView for CUDA allows Linux X86-64 users to debug both the CPU and GPU code in CUDA applications, using familiar TotalView GUI methods.

Key Features

- Single step operation advances all of the GPU hardwre threads in the same warp
- Also advance the execution of more than one warp.
- Newly, it Supports OpenACC directives
- Debugging host and device code in the same session
- CUDA running directly on Tesla or Fermi hardware

- Linux and GPU device thread visibility
- Full visibility to the hierarchical device, block, and thread memory
- Navigating device threads by logical and device coordinates
- CUDA function calls, host pinned memory regions and CUDA contexts
- Handling CUDA functions inline and on the stack
- Command line interface (CLI) commands for CUDA functions
- Applications that use multiple NVIDIA devices at the same time
- MPI applications on CUDA-accelerated clusters

Memory debugger features

Memory Debugger:

- Streamlined
- Collaborative
- Shows Memory errors
- Memory status
- Memory leaks
- Buffer overflows
- MPI memory debugging

- Remote memory debugging

Figure 3.29 shows a typical view of the total view debugger.

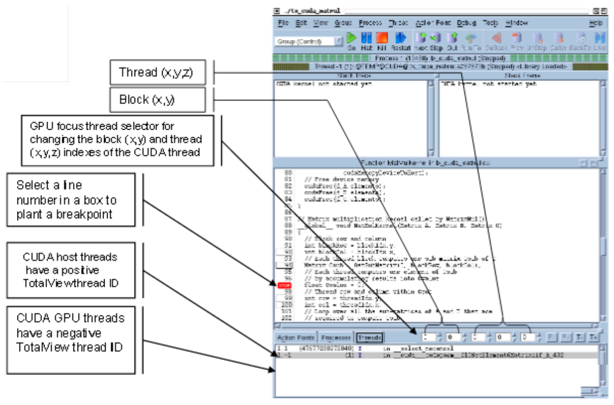

Thread (x,y,z)
Block (x,y)
GPU focus thread selector for changing the block (x,y) and thread (x,y,z) indexes of the CUDA thread
Select a line number in a box to plant a breakpoint
CUDA host threads have a positive TotalView thread ID
CUDA GPU threads have a negative TotalView thread ID

Figure 3.29: Total View Debugger

3.6.3 Vampir toolset

As debuggers main purpose is usually finding errors that cause application incorrectness; performance analysis tools have the main concern of finding opportunities of enhancing the performance of an application; or, stated in another way, to find the spots that are hindering the parallel program from reaching its full speed potential. This section lists some tools that are concerned with this purpose on Nvidia accelerators.

Vampir software gives the user the ability of visual performance analysis. While vampirTrace's goal is for instrumentation and measurement.

VampirTrace performance monitor gives detailed insight into the runtime behavior of accelerators. This enables an extensive performance analysis and optimization of hybrid programs written in CUDA, OpenCL, and PyCUDA. Vampir-

Trace is capable of tracing GPU accelerated applications and generates exact time stamps for all GPU related events. The information can be used to generate quick profiles or can also be graphically analyzed using Vampir. Vampir allows interactive navigation (zooming, moving) through the timelines of the execution of a parallel application annotated with a lot of statistics like time consumed, number of invocations, messages statistics, performance counter support, etc. The latest addition also allows capturing of GPU performance counters.

Key features

- locate load imbalances and understand what the application is actually doing
- Integration into the build process by supplied compiler wrappers
- GPU performance counter support via CUDA Performance Tools Interface (CUPTI)
- parallel analysis engine to support interactive trace analysis
- Hierarchical process folding in the master timeline.
- Introduction of combinable peer-to-peer communication metrics in the performance radar.
- Pre-selection of processes or threads prior to loading performance data.

Work flow examples

Vampire trace traces both CUDA single threaded applications and multithreaded that uses multi GPUs. A typical workflow is as follows:

1. Instrument the application using VampirTrace

2. Run application with an appropriate test-set

    Should only run for a few minutes

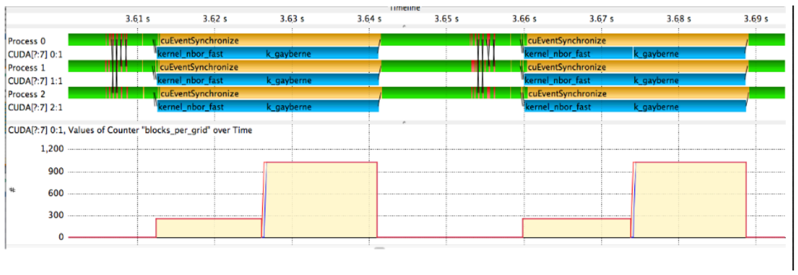
3. Analyze the trace file with Vampir GUI in the level of detail that is wished.

4. Analyze usage of GPU using Vampir trace:

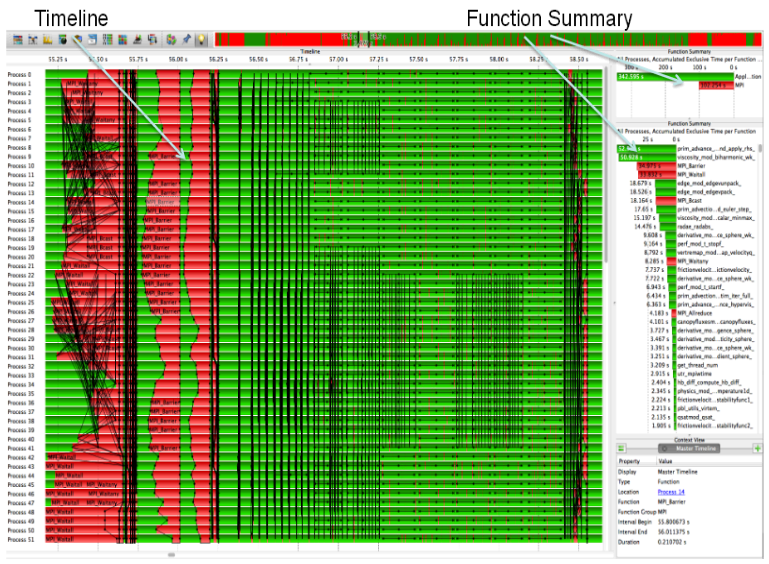
    Interaction with CPU

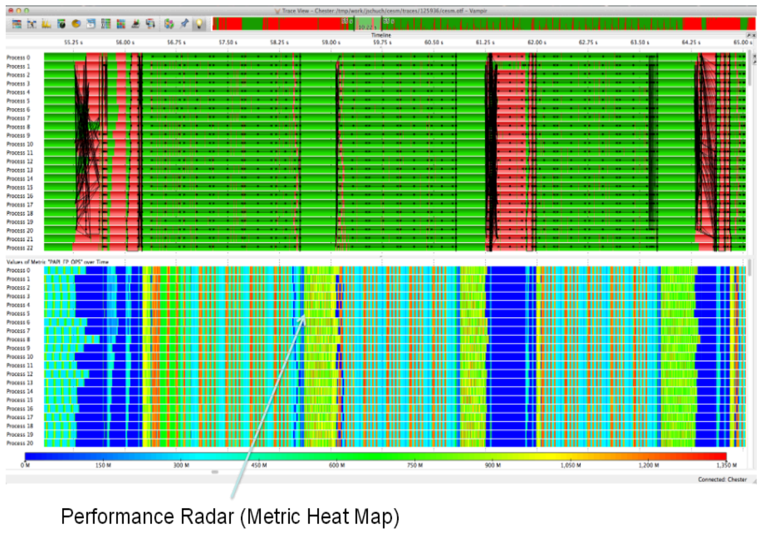
    Kernel activity and GPU related metrics

5. GPU streams displayed as: CUDA[device:stream] process:thread

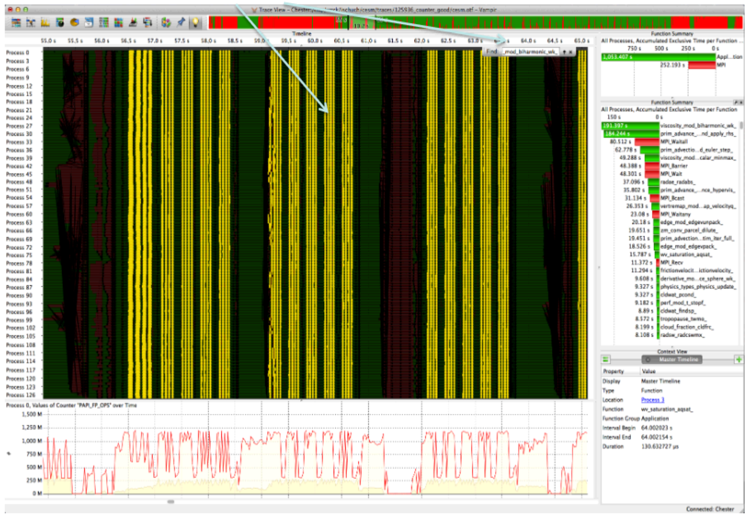


As an example : the following figures show how this program traces communication and computation activities for both single threaded and multithreaded applications.

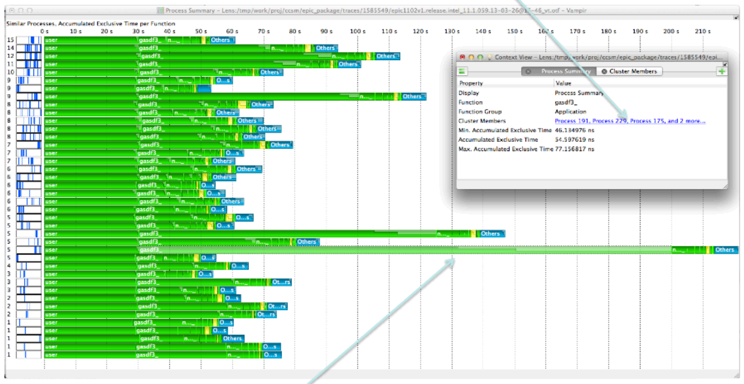

Timeline
Function Summary


Performance Radar (Metric Heat Map)

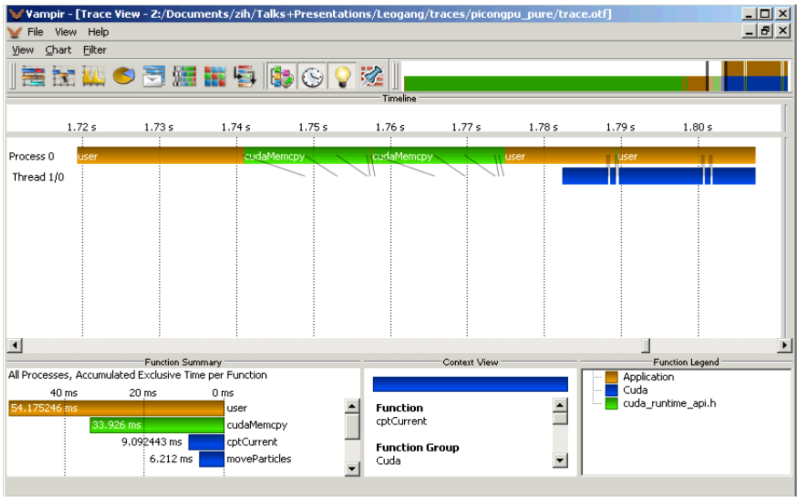
Function Search

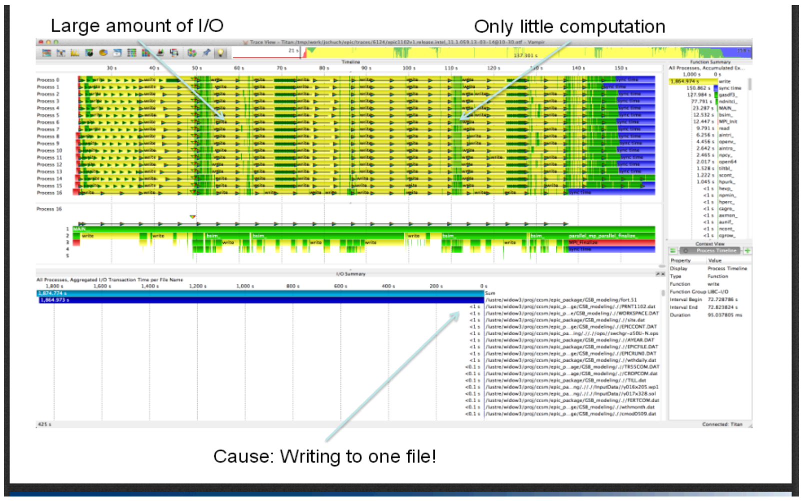


Detailed Process List



Find Outlier Processes




Large amount of I/O
Only little computation
Cause: Writing to one file!

CHAPTER 4

ANALYSIS OF COMPILER RESTRUCTURING AND OPTIMIZATIONS

4.1 Exploration of Automatic Optimization for CUDA Programming

We proposed a CUDA kernel restructuring algorithm, a general strategy to achieve maximum possible performance by better utilization of the machine. In CUDA, the worker threads are identified by thread ID and being organized by blocks which are identified by block ID. This identification is used in a kernel to define a mapping of computations to threads (workers). The proposed restructuring algorithm aimed at generating efficient CUDA kernels. It is based on the three key concepts that are explained in detail in following subsections.

4.1.1 Tiling

In CUDA the programmer has to explicitly transfer data from slow low-level Global Memory (GM) which is visible by all SMs to a fast high-level shared memory (ShM) within each SM. Tiling the code is to account for the small ShM capacity. The execution style is based on transferring small amount of data followed by data processing. While transforming the code, it is required to perform proper calculation of effective address of array elements (results) based on the workers identifiers which are the block ID and thread ID. It is required to design an algorithm/mechanism that can be used to apply loop tiling on any CUDA program with proper memory hierarchy optimizations. Tiling is guided by the following steps:

1. Identification of proper tile size to be stored in shared memory based on the limited capacity of ShM per kernel block.

2. Loop transformations and proper identification of range of outer and inner loops.

3. Effective address calculations of the array elements to be accessed within the loop iterations (see section 4.1.2).

4. Boundary check for avoiding the out of bound array index access.

5. Synchronization among loading of data into ShM, execution of operations, and storing the results back into GM.

4.1.2 Coalesced Global Memory Access

In this section, the objective is to restructure the code so that at each warp execution access to GM is done according to a coalesced access pattern to amortize the excessive access cost. Fetching a group of data elements which are stored in distinct memories (coalesced access) is critical to amortize the high cost of accessing GM compared to the speed of the logic. The key idea is to determine all possible mapping.

In CUDA a 1-D kernel having NW threads is represented as a set of N blocks each has W elements. To assign some work to each individual thread, each kernel thread is identified by the block b to which it belongs to and some offset t, i.e. $th_{id} = b.W + t$ or as a vector $th_{id} = (b,t)_{N,W}$,where $0 \leq b \leq N-1$ and $0 \leq t \leq W-1$. Suppose we have a 2-D array of U.V computation results which are stored using row-major scheme as U rows and V columns, the address of the element in row r and column c is $EA = (r,c)_{U,V} = r.U + c$, where $0 \leq r \leq U-1$ and $0 \leq c \leq V-1$. Assigning a thread (worker) to compute a result requires defining a mapping from the thread IDs onto the results so that when the SPMD program is run, each thread uses its own ID in the code to determine the result that it must compute. The mapping of threads IDs onto the result address admits a few possible mapping solutions for $EA = (r,c)_{u,v}$ as computes:

1. $EA = ((b,t)_{N,W}, c)_{U,V}$ — $N \times W = U$, each thread has one loop to compute V results, no coalesced access

2. $EA = (r, (b,t)_{N,W})_{U,V}$ — $N \times W = V$, each thread has to compute U results,

coalesced access

3. $EA = ((b,t')_{N,W},(b',t)_{N,W})_{U,V}$ — $N \times W' = U$ and $N' \times W = V$, each thread has two loops (denoted by ') to computes $(U \times V)/(N \times W)$ results, coalesced access

4. $EA = ((b',t)N',W,(b,t')_{N,W})_{U,V}$ — $N' \times W = U$ and $N \times W' = V$, each thread has two loops (denoted by ') to computes $(U \times V)/(W \times N)$ results, coalesced access

Note that a coalesced access takes place only when the offset, or second component of EA, is mapped to the thread index, i.e. identified by offset t. The reason is that warps are formed by successive thread IDs for any dimension, i.e. according to row major organization. Table 4.1 shows the possible mappings of CUDA for 1-D and 2-D kernels (blocks and threads) to a 2-D array of results of size space $N \times W$ with corresponding tile size (upper parameter) and coalesced (Yes) or non-coalesced (No) accesses. Similar approach is used for higher dimension kernels.

For example, assume a 2-D(U,U) array res() of results, and $T \times T$ as being the tile size. Let's use a 1D kernel defined by $thid = (b,t)_{N,W}$. For 1-D kernel, we may use the solution shown in the third row of Table 4.1. The corresponding constraints leads to N=U/T blocks and each block has each W=T threads. The effective address of a result res() is EA=(b*T+t')*U+b'*T+t. Each kernel thread consists of a double nested loop, the outer loop (t': U/T iterations) and inner loop (b': T iterations). It is clear that access is coalesced because t is in the least

1D Kernel		2D Kernel	
$th_{id} = b.W + t = (b,t)_{N,W} \| 0 \leq b \leq N-1$ and $0 \leq t \leq W-1$ $EA = (r,c)_{U,V} = r.U + c$, $0 \leq r \leq U-1$ and $0 \leq c \leq V-1$ Note: X' is a local loop within the thread		$th_{id} = (bx.Wx + tx, by.Wy + ty)$ $= ((bx,tx)_{N_x,W_x}, (by,ty)_{N_y,W_y})$— $0 \leq bx \leq Nx-1, 0 \leq by \leq Ny-1$ $0 \leq tx \leq Wx-1, 0 \leq ty \leq Wy-1$ $EA = (r,c)_{U,V} = r.U + c$, $0 \leq r \leq U-1$ and $0 \leq c \leq V-1$	
$((b,t)_{N,W}, c)_{U,V}$ N.W=U	U No	$((bx,tx)_{N_x,W_x}, (by,ty)_{N_y,W_y})$ Nx.Wx=U, Ny.Wy=V	1 No
$(r, (b,t)_{N,W})_{U,V}$ N.W=V	V Yes	$((by,ty)_{N_y,W_y}, (bx,tx)_{N_x,W_x})$ Nx.Wx=U, Ny.Wy=V	1 Yes
$((b,t')_{N,W}, (b',t)_{N,W})_{U,V}$ N.W'=U	(U.V)/(N.W) Yes	$((by,tx)_{N_y,W_x}, (bx,ty)_{N_x,W_y})$ Ny.Wx=U, Nx.Wy=V	1 No
$((b',t)_{N,W}, (b,t')_{N,W})_{U,V}$ N'.W=U	(U.V)/(N.W) No	$((bx,ty)_{N_x,W_y}, (by,tx)_{N_y,W_x})$ Nx.Wy=U, Ny.Wx=V	1 Yes

Table 4.1: Possible 1-D and 2-D Kernel mapping to a 2-D Array of results

significant position.

4.1.3 Resource Optimization

Within each SM, ShM is partitioned among active blocks which are assigned to SM for simultaneous execution. Therefore the tile sizes must be selected such that the tile data locality that must be loaded into ShM does not constrain the maximum number of active blocks which can be assigned to an SM at a time.

The block size must be chosen less than or equal to tile size such that each thread in a block loads one or more elements of a tile into ShM. This will reduce instruction fetch and processing overhead of load instruction since the device perform one instruction fetch for a block of threads which is in SIMT manner. On the other hand, too large block sizes must be avoided limiting the number of active blocks per SM due to large number of warps per block. The number of active warps must be no less than the maximum warps per SM (for full occupancy) in any given SM to avoid limiting the number of active threads per SM. Active Blocks can be calculated using equation 4.1.

$$Active \quad Blocks = min\begin{bmatrix} min\left(\left\lceil \frac{Warp \quad Per \quad SM}{Warp \quad Per \quad Block} \right\rceil, Max. \quad Blocks \quad per \quad SM\right) \\ min\left(\left\lceil \frac{Shared \quad Memory \quad Per \quad SM}{Shared \quad Memory \quad Per \quad Block} \right\rceil, Max. \quad Blocks \quad per \quad SM\right) \end{bmatrix} \tag{4.1}$$

$$Here,$$

$$Warps \quad Per \quad Block = \frac{Threads \quad Per \quad Block}{Threads \quad Per \quad Warp} \tag{4.2}$$

$$\begin{aligned} Shared \quad Memory \quad Per \quad Block &= Tile \quad Size \times Data \quad Element \quad Size \\ &\times \quad Number \quad of \quad Data \quad Elements \quad to \quad load \quad for \quad one \quad result \end{aligned} \tag{4.3}$$

$$\begin{aligned} Warps \quad Per \quad Block &= \frac{256}{32} = 8 \\ Shared \quad Memory \quad Per \quad Block &= 256 \times 4 \times 2 = 2048 \\ Active \quad Blocks &= min\begin{bmatrix} min[\lceil \frac{32}{8} \rceil, 8] \\ min[\lceil \frac{16384}{2048} \rceil, 8] \end{bmatrix} \\ &= min\begin{bmatrix} min[4, 8] \\ min[8, 8] \end{bmatrix} \\ &= min\begin{bmatrix} 4 \\ 8 \end{bmatrix} = 4 \end{aligned}$$

$$Active \quad Kernel \quad Blocks \quad Per \quad SM(AKBPSM) = \frac{TotalKernelBlocks}{TotalSMs} \tag{4.4}$$

$$Here, Total \quad Kernel \quad Blocks = Application \quad Space \quad Size/Tile \quad Size$$

$$S - Cycles = \frac{Active \quad Blocks \times Threads \quad Per \quad Block}{SPs \quad per \quad SM} \tag{4.5}$$

For example, if Threads per Block is 256, Tile Size is 256, Data Element Size is

4 bytes, and Number of Data Elements to load for one result is 2, then the Active Blocks is 4. Suppose Warps Per SM is 32, Shared Memory Per SM is 16384, and Max. Blocks Per SM is 8. Therefore the number of active blocks that can be handled by an SM at a given time can be calculated using eq. 4.1.

To expose to peak performance, the application threads must be massively and uniformly spread over the SMs so that the only performance saturation comes from mapping the application to the GPU. Furthermore, peak performance will be expected because all the SM and SPs are involved in the execution. Since, there are two levels of kernel block and threads scheduling in the device. The blocks are first scheduled to be executed on each SM and then each SM schedules the individual threads within a block to multiple SPs within the SM based on selecting one warp at a time. The repetitions due to first scheduling can be analysed as average kernel blocks per SM and the repetitions due to second scheduling as small cycles (S-Cycles) which occur due to limited number of SPs (Thread Processors) that can execute one thread at a time.

These repetitions should satisfy the following conditions to achieve peak performance:

1. Both AKBPSM and S-Cycles should be greater than or equal to 1.

2. S-Cycles should be an integer value to balance the threads among multiple SPs.

3. S-Cycles should be as large as possible.

4. AKBPSM should be the least possible to minimize serialization.

4.1.4 Proposed CUDA Restructuring Algorithm

The proposed restructuring tool algorithm (RTA) converts a C source loop into an optimized CUDA Kernel. RTA carry out loop tiling (Section 4.1.1), complex address transformation to implement coalesced global memory access (Section 4.1.2), and a set up the kernel parameters based on solving the resource constraints equations (Section 4.1.3). The proposed RTA restructuring algorithm is based on the following steps:

Step 1- Kernel dimension and thread granule size

1. **Kernel Dimension:** map kernel dimension to the dimension of the computation. Implement the kernel with parametric number of blocks and number of threads in a block.

2. **Thread Granule Size:** If for each result [Comp./Comm.=1/s ¿ Threshold] then each thread is assigned one result, i.e. size of kernel is identical to size of results, else each thread computes s results (size of kernel is 1/s the size of result), where Comp. is time to compute a result and Comm. is time to fetch operands for a result.

Step 2-Loop Tiling and Coalesced Access

1. **Symbolic Tiling:** Tile the resulting loop (or loops) by generating all possible tiled loop arrangements.

2. **Tiled Solutions:** Select one or more tiled arrangements.

3. **Coalesced Access:** swap Addresses for getting thread identifier in LSB position, e.g. successive threads in a warp access successive memory banks.

Step 3- Resource Optimization

1. **Kernel, Block, and Tile Sizes:** Compute the best possible combination of Threads per block (TPB) and the Tile Size (TS) to get the optimal distribution of blocks and threads among SMs and SPs respectively. We need to generate all possible TPB and TS, and their respective Warps Per Block (WPB) and Shared Memory Per Block (ShMPB) using the eqs. (4.2 and 4.3).

2. **Active Blocks:** Identify Active Blocks using eq. 4.1 for each of the combination of TPB and TS

3. **S-Cycle:** Calculate S-Cycles for each of the combinations using eq. 4.5 and select the combinations that have the maximum value.

4. **Kernel Block / SM:** Calculate AKBPSM for the selected combinations and the one that has the minimum value of AKBPSM will give the best performance.

In section 4.1.7, we present an example for converting a C loop into an optimized CUDA Kernel.

4.1.5 Algorithm Complexity

In general, the proposed algorithm follows 3 simple steps to convert a C source loop into an optimized CUDA kernel.

In step 1 of the algorithm, the mapping of the largest array dimension (N_c), in the computation, to the kernel dimension (N_k) is basically O(1) except when N_c exceeds 3 (largest kernel dimension) in which case it becomes $O(N_c/3)$ as $N_c/3$ different kernels must be created for the computation. Similarly, if the condition in the thread granule size is not satisfied, a few results (s) are bundled in one thread and k steps are needed to check for the new thread coalesced memory access, in total $O(k * N_c/3)$ steps are needed.

In step 2, typically N x D possible tiled loop arrangements can be created where N is the possible tile sizes and D is the array dimension that needs to be partitioned in tiles. But this can be reduced to focus one a subset of tiled loop arrangements. After finding the required loop arrangements the array addresses can be converted into coalesced access in I steps where I is the number of non-coalesced array addresses within the tiled loop.

In step 3, finding the optimal values of TPB and TS takes a brute-force approach to search over all the possible combination of TPB and TS. The programmer has to compute M x N records of calculated parameters as identified in step 3 of section 4.1.4 where M is the possible values of TPB and N is the possible values of TS. So, the overall algorithm complexity is O(M x N).

4.1.6 Proof of Correctness

The correctness of the results of each converted application using the algorithm are guaranteed by comparing with the results of serial version of each application on CPU using the subset of the problem sizes. The result showed that our conversions produce correct resultant values. We have also trace the resultant matrix indices with the mapping of block and thread ids which are also found to be corrected.

4.1.7 Example

In this section, we will show the working steps of writing a matrix multiplication application from the sequential code (Code Listing 4.1, for N x N matrices) to optimized CUDA kernel.

Step 1: a 2-D kernel dimension is selected to match with problem dimension where each thread is mapped to computing one result. Due to the limited data locality and few arithmetic operations in the statement, each thread will compute one resultant element C[i][j].

Listing 4.1: Matrix Multiplication Sequential Code

```
void matrix_multiply(float **C, float **B, float **A, int
    N)
{
    for(int i=0; i < N; i++)
        for(int j=0; j < N; j++){
            C[i][j] = 0;
            for(int k=0; k < N; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}
```

Step 2: Code Listing 4.2 shows the tiled version of Code Listing 4.1 by using general strategy of loop tiling for uniprocessors that is split each loop of a nested loop-set into a pair of adjacent loops in the loop nest, with the outer loop (tiling loop) traversing tiles (blocks), and the inner loop (intra-tile loop) covering the iteration points within the tile. Code Listing 4.3 shows the corresponding CUDA kernel implementation using 2D blocks and threads that maps the outer four loops of Code Listing 4.2 to the blocks and threads dimensions in Code Listing 4.3. At this stage, accessing to matrix C and B are satisfying the mappings of coalesced memory access as shows in second row of 2D kernel mappings in Table 4.1 while access to matrix A is not coalesced.

Code Listing 4.4 shows the modified kernel to perform coalesced loads of matrix A and B using shared memory and coalesced stores to the resultant matrix C. The thread identifier is used as a linear index so that successive threads in a warp will access neighbouring memory modules. Here, we are assuming the same dimensions for thread blocks and matrix tiles. We also need to add barrier synchronization among threads of the same block using __syncthreads() between tiles load and compute statement within the traversal of all tiles of matrices A and B. Also a barrier is required before storing the resultant tile of matrix C due to difference in the traversal order of load/store and computation statements.

Step 3: For Tesla C2070 using the resource optimization strategy as explained in section 4.1.3, we found optimal values for threads per block and tile sizes as TPB = 32 * 16 512 and TS = 32 * 64 = 2048. Code Listing 4.5 shows the modified kernel of Code Listing 4.4 to handle the case of $TPB < TS$, for this we need to add loop for each load, compute and store statement to correctly load the whole tile, compute the results, and store the whole resultant tile to the destination.

Listing 4.2: Matrix Multiplication Tiled Version

```
void tiled_matrix_multiply(float **C, float **B, float **
   A, int N)
{
    for(int by=0; by < N; by+=TILE_Y)
        for(int bx=0; bx < N; bx+=TILE_X)
            for(int ty=0; ty < TILE_Y; ty++)
                for(int tx=0; tx < TILE_X; tx++)
                    for(int bk=0; bk < N; bk+=TILE_X)
                        for(int k=0; k < TILE_X; k++)
                            C[by+ty][bx+tx] = A[by+ty][bk
                               +k] * B[bk+k][bx+tx];
}
```

Listing 4.3: Matrix Multiplication CUDA Kernel

```
__global__ void
tiled_matrix_multiply(float *C, float *B, float *A, int N
   )
{
    int by = blockIdx.y * TILE_Y;
    int bx = blockIdx.x * TILE_X;
    int ty = threadIdx.y;
    int tx = threadIdx.x;

    for(int bk=0; bk < N; bk+=TILE_X)
        for(int k=0; k < TILE_X; k++)
            C[(by + ty) * N + bx + tx] = A[(by + ty) * N
               + bk + k]
                                          * B[(bk + k) * N
                                             + bx + tx];
}
```

Listing 4.4: CUDA Kernel with Coalesced Memory Access

```
__global__ void  coalesced_matrix_multiply(float *C,
   float *B, float *A, int N)
```

```
{
    int by = blockIdx.y * TILE_Y;
    int bx = blockIdx.x * TILE_X;
    int ty = threadIdx.y;
    int tx = threadIdx.x;
    float Csub=0;
    __shared__ float As[TILE_Y][TILE_X];
    __shared__ float Bs[TILE_X][TILE_X];
    for(int bk=0; bk < N; bk+=TILE_X){
        As[ty][tx] = A[(by + ty) * N + bk + tx];
        Bs[ty][tx] = B[(bk + ty) * N + bx + tx];
        __syncthreads();
        for(int k=0; k < TILE_X; k++)
            Csub += As[ty][k] * Bs[k][tx];
    }
    __syncthreads();
    C[(by + ty) * N + bx + tx] = Csub;
}
```

Listing 4.5: Optimized CUDA Kernel

```
__global__ void  gen_coalesced_matrix_multiply(float *C,
   float *B, float *A, int N)
{
    int by = blockIdx.y * TILE_Y;
    int bx = blockIdx.x * TILE_X;
    int ty = threadIdx.y;
    int tx = threadIdx.x;
    float Csub[TILE_Y/BLOCK_Y];
    __shared__ float As[TILE_Y][TILE_X];
    __shared__ float Bs[TILE_X][TILE_X];
    for(int bk=0; bk < N; bk+=TILE_X){
        for(int i=0; i < TILE_Y/BLOCK_Y; i++){
            As[ty + i * BLOCK_Y][tx] = A[(by + ty + i *
               BLOCK_Y)  * N + bk + tx];
        }
        for(int i=0; i < TILE_X/BLOCK_Y; i++){
            Bs[ty + i * BLOCK_Y][tx] = B[(bk + ty + i *
               BLOCK_Y)  * N + bx + tx];
        }
        __syncthreads();
        for(int i=0; i < TILE_Y/BLOCK_Y; i++)
            for(int k=0; k < TILE_X; k++)
                Csub[i] += As[ty + i * BLOCK_Y][k] * Bs[k
                   ][tx];
    }
    __syncthreads();
    for(int i=0; i < TILE_Y/BLOCK_Y; i++)
        C[(by + ty + i * BLOCK_Y) * N + bx + tx] = Csub[i
            ];
}
```

4.1.8 Performance Evaluation

Non-Coalesced Vs. Coalesced Global Memory Access

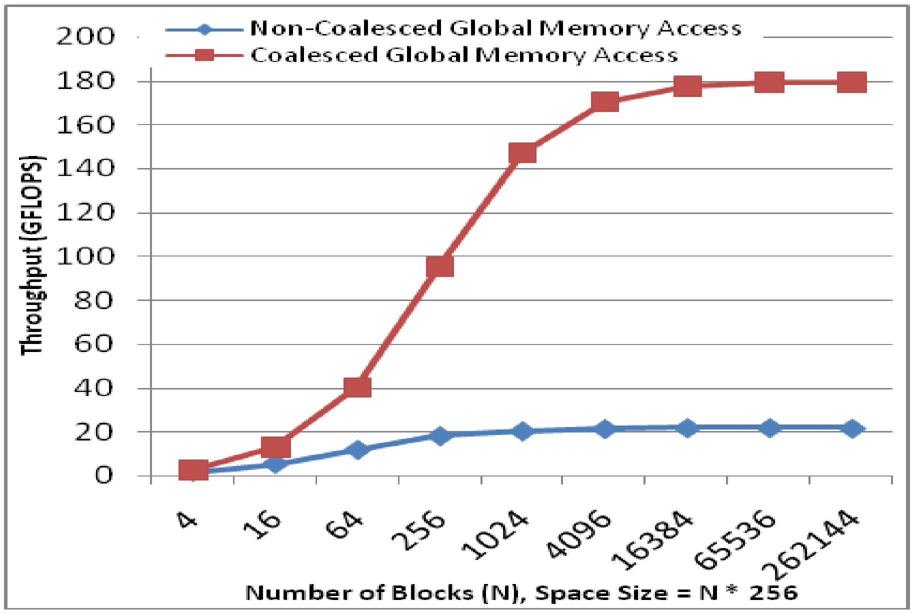


Figure 4.1: Matrix Multiplication using Shared Memory with (a) Non-Coalesced Global Memory Access and (b) Coalesced Global Memory Access

Figure 4.1 shows the GPU throughput (GFLOPS) of two 2D kernel mapping solutions for the matrix multiply. These solutions correspond to a tiled loop with and without coalesced GM access which are illustrated in the 2nd columns of Table 4.1: Possible 1-D and 2-D Kernel mapping to a 2-D Array results at the 2nd (No) and 3rd (Yes) rows, respectively. According to the solution (3rd row), a tile is first loaded into ShM from GM using a coalesced access and do the computations while data is in ShM. As in coalesced global memory access, threads in half warp (16 threads) access consecutive memory locations in one cycle so reducing the memory accesses by an ideal factor of 94%.The above solution allows reducing the program execution time by 87.87%.

Even with no coalesced GM memory access, copying a tile from GM onto ShM before execution is faster (about 22%) than loading the SM registers directly from GM. Figure 4.2 shows the corresponding throughput (GFLOPS). This reduction in time is due to the data re-use in shared memory. So, it is highly recommended to do tiling with the use of shared memory when the application needs to re-use the data among different loop iterations:

$$L_{G\rightarrow R} : Data \quad Loading \quad to \quad Registers \quad directly \quad from \quad Global \quad Memory$$
$$L_{G\rightarrow ShM\rightarrow R} : Data \quad Loading \quad to \quad Registers \quad from \quad Global \quad Memory \quad via \quad Shared \quad Memory \quad with \quad data \quad re-use$$
$$L_{G\rightarrow R} > L_{G\rightarrow ShM\rightarrow R}$$

Block Sizes Comparison

We refer to Resource Optimization described in Section 4.1.3. Increasing the number of threads per block may decrease the performance due to restriction in the concurrent number of blocks per SM which reduces SM capacity utilization.

A 256-thread block (option 1) has 8 warps each has 32 threads. Thus each SM will be assigned 4 blocks at a time. While a 484-thread block (option 2) has 16 warps leading each SM to be assigned 2 blocks at a time. Comparing the above two

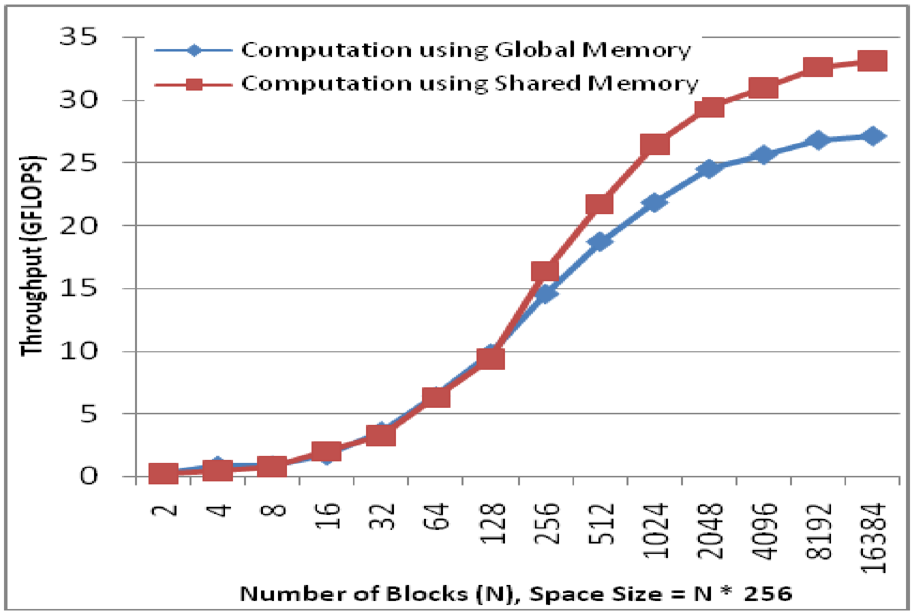


Figure 4.2: Matrix Multiplication using Computations with (a) Global Memory and (b) Shared Memory

options, it is clear that option 1 provides larger S-cycles and smaller AKBPSM. Here, option 1 represents a case for the best possible resource utilization. Figure 4.3 shows the GPU throughput (GFLOPS) for both options in which the solution corresponding to option 1 is more than 5 times faster than that corresponding to option 2.

4.1.9 Application Results Comparison

Matrix Multiplication

We have analysed the structure of matrix multiplication kernels using CUDALite [11] approach and NVIDIA SDK approach [22]. Both of these implementations used arbitrary values for defining threads per block (TPB) and tile size (TS) which are not optimal values in terms of resource optimization as we have explained in

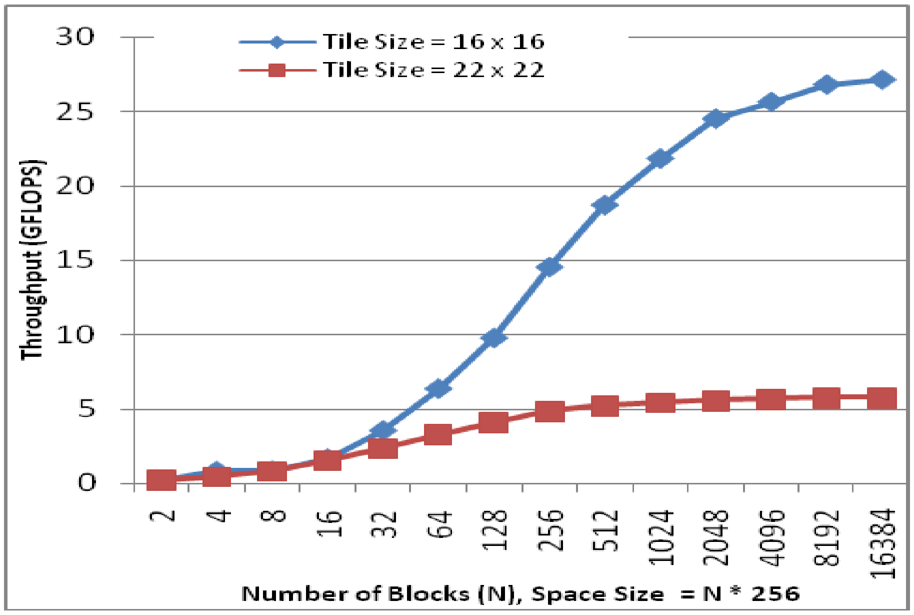


Figure 4.3: Matrix Multiplication using only global memory with different number of threads per block (a) 16 x 16 = 256 threads/block and (b) 22 x 22 = 484 threads/block

section 4.1.3. In CUDALite, each thread work on the entire row of the tile resulting in very few threads per block (TPB = 32 as shown in Table 4.2 that only 1 warp per block) which is not sufficient to hide latency of the global memory transfers. Also, in CUDALite, a tile allocation is also done for results which causes large shared memory usage per thread block that restricts the number of Active Blocks (AB = 1, see Table 4.2, can be calculated using eq. 4.1) that highly reduces the S-Cycles to 1. In NVIDIA SDK approach, 2D thread blocks of 16 x 16 dimensions is defined with same tile sizes so each thread work on one element of each tile but these values produces large number of average kernel blocks per SM which causes increased overhead of blocks allocation and thus limited performance. The optimal value of TPB and TS for Tesla C2070 GPU are 512 and 2048 respectively as proposed by our restructuring algorithm and gives the minimum execution time in comparison of the other approaches.

Tesla C2070 (N = 2048 x 2048)							
	TPB	**TS**	**AB**	**TKB**	**S-Cycles**	**AKBPSM**	**Exec. Time**
Restructuring Algorithm	512	2048	3	2048	48	146.2857143	2.4486
NVIDIA SDK	256	256	6	16384	48	1170.285714	2.6268
CUDALite	32	1024	1	4096	1	292.5714286	21.2396

Table 4.2: Parameters comparison of different implementations of Matrix Multiplication

Furthermore, we have applied our resource optimization technique as explained in section 4.1.3 on Volkov matrix multiplication algorithm [19] and also on matrix multiplication kernel generated by GPGPU compiler [16]. In both cases, our restructuring algorithm selects optimal values of TPB and TS. Table 4.3 shows the results of application of resource optimization on Volkov matrix multiplication and Table 4.4 shows the application of resource optimization on matrix multiplication kernel by GPGPU compiler.

Quadro FX 7000 (N = 4096 x 4096)							
	TPB	**TS**	**AB**	**TKB**	**S-Cycles**	**AKBPSM**	**Exec. Time**
Restructuring Algorithm	256	4096	4	4096	32	256	0.2520
Volkov-MM	64	1024	8	16384	16	1024	0.2609

Table 4.3: Parameters comparison of Volkov MM implementations

Tesla C2070 (N = 2048 x 2048)							
	TPB	**TS**	**AB**	**TKB**	**S-Cycles**	**AKBPSM**	**Exec. Time**
Restructuring Algorithm	256	1024	3	4096	24	292.5714286	0.5561
GPGPU Compiler - MM	256	256	3	16384	24	1170.285714	0.5585

Table 4.4: Parameters comparison of matrix multiplication kernel generated by GPGPU

Matrix Scaling

We have also analysed the matrix scaling kernel shown as an example in CUDALite [11] paper. We have found similar problems of limited number of active blocks due to large shared memory usage and also large number of average kernel blocks per SM due to small number of threads per blocks as explained in the previous section 4.1.3 in the case of matrix multiplication. The optimal value of TPB and TS for Tesla C2070 GPU are 512 and 4096 respectively as proposed by our restructuring algorithm (see Table 4.5) and gives the minimum execution time in comparison of the CUDALite approach.

Tesla C2070 (N = 2048 x 2048)							
	TPB	**TS**	**AB**	**TKB**	**S-Cycles**	**AKBPSM**	**Exec. Time**
Restructuring Algorithm	512	4096	3	1024	48	73.14285714	0.0014
CUDALite	32	1024	1	4096	1	292.5714286	0.0096

Table 4.5: Parameters comparison of different implementations of Matrix Scaling

Matrix Transpose

NVIDIA provides optimized kernels of matrix transpose by analysing the architectures of shared memory and global memory. In these optimizations, tiles are allocated in shared memory in such a way that the access to the shared memory by different threads at the same time should be free from shared memory bank conflicts. Furthermore, access to global memory by concurrent thread blocks will be done in different partitions of global memory to load the tile from the source matrix and store the tile into transposed matrix. We have applied our resource optimization strategy to two different matrix transpose kernels as provided in NVIDIA SDK. TPB = 512 is obtained as an optimal value for threads per block that maximize S-Cycles (see Table 4.6 and 4.7) and hence minimize the execution time in comparison of the defined parameters in NVIDIA documentation.

Quadro FX 7000 (N = 2048 x 2048)							
	TPB	**TS**	**AB**	**TKB**	**S-Cycles**	**AKBPSM**	**Exec. Time**
Restructuring Algorithm	512	1024	3	4096	48	256	0.0776
NVIDIA SDK	256	1024	5	4096	40	256	0.1084

Table 4.6: Parameters comparison of Matrix Transpose kernels with no shared memory bank conflicts

Quadro FX 7000 (N = 2048 x 2048)							
	TPB	**TS**	**AB**	**TKB**	**S-Cycles**	**AKBPSM**	**Exec. Time**
Restructuring Algorithm	512	1024	3	4096	48	256	0.0800
NVIDIA SDK	256	1024	5	4096	40	256	0.1234

Table 4.7: Parameters comparison of Matrix Transpose kernels with diagonal tiles mapping to blocks to avoid partition camping

Matrix Vector Multiplication

We have also applied our resource optimization technique as explained in section 4.1.3 on matrix vector multiplication kernel generated by GPGPU compiler [16]. In this case also, our restructuring algorithm selects optimal values of TPB and TS. Table 4.8 shows the results of application of resource optimization on matrix vector multiplication kernel by GPGPU compiler.

Quadro FX 7000 (N = 2048 x 2048)							
	TPB	**TS**	**AB**	**TKB**	**S-Cycles**	**AKBPSM**	**Exec. Time**
Restructuring Algorithm	64	256	8	65536	16	4096	0.0008
GPGPU Compiler - MV	16	256	8	65536	4	4096	0.0012

Table 4.8: Parameters comparison of matrix-vector multiplication kernel generated by GPGPU compiler

4.2 CUDA Kernel Optimizations

CUDA kernel can be optimized in many possible ways with good understanding of the needs of the application. NVIDIA suggests a cyclic process namely APOD (Asses, Parallelize, Optimize, Deploy) [55] to help application developers

Type	**Optimization**
Manual	Vectorization
	Texture Fetching
	Coalesced Global Memory Access
	Kepler Shuffle Instructions
User Driven	Loop Collapsing
	Thread / Thread-Block Merge
	Parallel Loop Swap
	Strip Mining
	Bank Conflict Free Shared Memory Access
	Using Read Only Data Cache
Compiler	Common Sub-Expression Elimination
	Loop Invariant Code Motion
	Loop Unrolling

Table 4.9: CUDA Kernel Optimizations Categorizations

to rapidly identify the portions of their code that would most readily benefit from GPU acceleration, rapidly realize that benefit, and begin leveraging the resulting speedups in production as early as possible. Using APOD, a programmer can apply and test the optimization strategies incrementally as they are learned. Optimizations can be applied at various levels, from overlapping data transfers with computation all the way down to fine-tuning floating-point operation sequences.

We have explored several optimizations that can be categorized into three classes based on their application:

1. **Manual:** This set of optimizations can only be applied by the programmer himself through manual code analysis and modifications.

2. **User Driven:** This set of optimizations can be applied by an automatic process/compiler providing few hints from the programmer.

3. **Compiler:** This set of optimizations is already applied within the current

cuda compiler (nvcc) and applied automatically.

Table 4.9 categorizes these optimizations into different types.

4.2.1 Manual Optimizations

Vectorization improves the bandwidth utilization by using one of the vector data types such as float2, float4, float8 in CUDA as structs with some special data alignment [56, 57]. Global memory transactions in GPU are aligned to 128 bytes even if the actual data load is less than 128 bytes. So, if a specific thread within a warp performs 8 loads of float data type with sequence of load instructions then the GPU perform 8 different memory transactions. On the other hand, if a float8 type is used and perform single float8 load operation then it can be done by only one global memory transaction. So, to improve global memory bandwidth utilization, the programmer should use vector data types in the case when consecutive data loads are not aligned to 128 bytes. **Texture Fetching** utilized the texture memory that is a read-only portion of memory in device memory (DRAM) that has been cached (off-chip cache) on access [57, 56]. It has been accessible by all threads and host. It is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together achieve best performance. Texture references that are bound to CUDA arrays can be written to via surface-write operations by binding a surface to the same underlying CUDA array storage. Reading from a texture while writing to its underlying global memory array in the same kernel launch should be avoided because the texture caches are read-only and are not invalidated when the associated global memory is modified. So, texture fetches from addresses that have been written via global stores in the same kernel call returned undefined data. The data in texture can consist of 1, 2, or 4 elements of any of the following types: 1) Signed or unsigned 8, 16, or 32-bit integers, 2) 16-bit floating point values, and 3) 32-bit floating point values. Arrays declared in texture memory can be used in kernels by invoking texture intrinsic provided in CUDA such as tex1D(), tex2D(), and tex3D() for 1D, 2D, and 3D CUDA arrays respectively. Before invoking a kernel that uses texture memory, the texture must be bound to a CUDA array or device memory by calling cudaBindTexture(), cudaBindTexture2D(), or cudaBindTexturetoArray(). **Coalesced Global Memory Access** refers to combining multiple memory accesses into a single transaction [58]. Global memory is the slowest memory on the GPU. Simultaneous global memory accesses by each thread of a half-warp (16 threads) during the execution of a single read and write instruction are coalesced into a single access. This is achieved based on the following conditions: 1) the size of the memory element accessed by each thread is either 4, 8, or 16 bytes, 2) the elements to be accessed form a contiguous block of memory, 3) the N^{th} element is accessed by the N^{th} thread in the half-warp, does not affect if any thread in between not accessing the global memory that is divergent warp, and 4) the address of the first element is aligned to 16 times the element's size. **Kepler's Shuffle Instructions** perform data exchange between threads within a warp [59]. It is more faster than the use of shared memory. This feature allows the threads of a warp to exchange data with each other directly without going through shared (or global) memory. So, this can present an attractive way for applications to rapidly interchange data among threads. There are four variants of shuffle instructions in CUDA that are __shfl(), __shfl_up(), __shfl_down(), and __shfl_xor(). Shuffle instructions can be used to free up shared memory to be used for other data or to increase warp occupancy and to perform warp-synchronous optimizations (removing __syncthreads()). All the __shfl() intrinsics take an optional width parameter which permits sub-division of the warp into segments. For example, to exchange data between 4 groups of 8 lanes in a SIMD manner. If width is less than 32 then each subsection of the warp behaves as a separate entity with a starting logical lane ID of 0. A thread may only exchange data with others in its own subsection. Width must have a value which is a power of 2 so that the warp can be subdivided equally; results are undefined if width is not a power of 2, or is a number greater than warpSize.

4.2.2 User Driven Optimizations

Loop Collapsing is a technique to transform some nested loops into a single-nested loop to reduce loop overhead and improve runtime performance [60] specifically for irregular applications such as sparse matrix vector multiplication (spMV). Such applications pose challenges in achieving high performance on GPU programs because stream architectures are optimized for regular program patterns. It improves the performance of the application in three ways: 1) the amount of parallel work (the number of iterations, to be executed by GPU threads) is increased 2) inter-thread locality is increased 3) control flow divergence is eliminated, such that adjacent threads can be executed concurrently in an SIMD manner [12]. **Thread and Thread-Block Merging** enhance the data sharing among thread blocks to reduce the number of global memory accesses [16, 61]. Thread-Block Merge determines the workload for each thread block while Thread Merge decides the workload for each thread. If data sharing among neighbouring blocks is due to a global to shared memory (G2S) access, Thread-Block Merge should be preferred to better utilization of the shared memory. When data sharing is from a global to register (G2R) access, Thread Merge from neighbouring blocks should be preferred due to the reuse of registers. If there are many G2R accesses, which lead to data sharing among different thread blocks, the register file is not large enough to hold all of the reused data. In this case, Thread-Block Merge should be used and shared memory variables should be introduced to hold the shared data. In addition, if a block does not have enough threads, Thread-Block Merge instead of Thread Merge should also be used to increase the number of threads in a block even if there is no data sharing. Thread Merge achieves the effects of loop unrolling. It combines several threads' workload into one thread (combining N neighbouring blocks along column direction into one). By doing this, they can share not only shared memory but also the registers in the register file. Furthermore, some control flow statements and address computation can be reused, thereby further reducing the overall instruction count. The limitation is that an increased workload typically requires a higher number of registers, which may reduce the number of active threads that can fit in the hardware. **Parallel Loop Swap** is used to improve the performance of regular data accesses in nested loops [62, 12]. It uses to transform non-continuous memory accesses within the loop nest to a continuous memory access which is a candidate for the coalesced global memory access optimization. **Strip Mining** splits a loop into two nested loops [63, 64, 65, 19]. The outer loop has stride equal to the strip size and the inner loop has strides of the original loop within a strip. This technique is also used in loop tiling. In loop tiling or loop blocking, loops are also interchanged after performing strip mining to improve the locality of memory references that is why loop tiling is also called strip-mine-and-interchange. **Bank Conflict Free Shared Memory Access** improves performance by reordering the data into shared memory such that the memory addresses requested by the consecutive threads in a half-warp should be mapped to different memory banks of shared memory [66]. Shared memory banks are organized such that successive 32-bit words are assigned to successive banks and the bandwidth is 32 bits per bank per clock cycle. In GPUs, the warp size is 32 threads and the number of banks is 16. So, a shared memory request for a warp is split into one request for the first half of the warp and one request for the second half of the warp. However, no bank conflict occurs if only one memory location per bank is accessed by a half warp of threads. **Using Read-Only Data Cache**, introduced in Kepler in addition to L1 cache, can benefit the performance of bandwidth-limited kernels [67, 59]. This is the same cache used by the texture pipeline via a standard pointer without the need to bind a texture beforehand and without the sizing limitations of standard textures. This feature is automatically enabled and managed by the compiler, access to any variable or data structure that is known to be constant through programmer use of the C99-standard "const __restrict__" keyword tagged by the compiler to be loaded through constant data cache.

4.2.3 Compiler Optimizations

Common Sub-Expression Elimination is a compiler optimization technique that searches for instances of identical expressions, evaluates to the same value, and replace them with a single variable holding the computed value [68, 69]. It enhances the application performance by reducing the number of floating point operations. In CUDA, common sub-expression elimination can be used to avoid redundant calculations for the initial address of an array. **Loop Invariant Code Motion** (also called hoisting or scalar promotion) is a compiler optimization that has been performed automatically [70, 46]. Loop invariant code is a set of statements or expressions within the body of a loop that can be moved outside of the body without affecting the semantics of the program. It makes loops faster by reducing the amount of code that executes in each iteration of the loop. The CUDA C compiler automatically applies this optimization technique to the PTX code. **Loop Unrolling** is a compiler optimization technique that is applied for the known trip counts at the compile time either by using the constants or templating the kernel [71]. NVIDIA compiler also provides a directive '#pragma unroll' to explicitly activate the loop unrolling on a particular loop.

4.3 Analysis of GPGPU Framework

Not surprisingly, there is a growing research for reducing the difficulty of programming GPU devices [72, 73, 74]. Even though the programming model of Compute Unied Device Architecture (CUDA) offer a more programmer friendly interface , programming GPUs is still considered error-prone and complex, in comparison to programming CPUs with standing parallel programming models, such as OpenMP [75]. Most recently, quite a few directive-based GPU programming models have been proposed from both the research community (hiCUDA [13], OpenMPC [76] , etc.) and industry (PGI Accelerator [77], HMPP , R-Stream [78], OpenACC, OpenMP for Accelerators [79], etc.). On the surface, these models appear to offer different levels of abstraction, and expected programming effort for code restructuring and optimization. Following are the details of optimizations that have been implemented in some of the tools for GPU programming found in literature.

4.3.1 CUDA-lite

The high burden of correctly exploiting the architecture of memory hierarchy for performance gains in GPUs; motivated [11] to introduce CUDA-lite. It takes as input a naïve CUDA code that treats the memory as a single entity instead of a hierarchical one. The naïve CUDA code could be annotated with proposed extensions to maximize the efficiency of the transformation.

CUDA-lite does not affect parallelization decisions. It only operates under the state of how the program got parallelized. CUDA-Lite performs the following transformations:

- Inserts shared memory variables
- Performs loop tiling
- Generates memory coalesced loads and/or stores

Replaces global memory accesses by corresponding shared memory ones

4.3.2 hiCUDA

hiCUDA [13] is a directive based language for programming NVIDIA GPUs. hiCUDA stands for high-level CUDA. The authors intended an abstraction that closely matches CUDA model. They wanted a CUDA with a new and simpler directives set.

hiCUDA's goal is not to automate optimizations, rather it is to make it easier for the programmer to program CUDA, for example it provides simple directives to ease data transfers between CPU and GPU. The programming model of hiCUDA still depends on explicit optimizations by the programmer, such as utilizing shared memory or constant memory.

hiCuda does very few implicit optimizations. Namely it tries to minimize the size of shared memory used based on the life time of shared memory variables.

Also they restrict the distribution strategy for loops over threads to be cyclic for memory coalescing purpose.

4.3.3 OpenMPC (OpenMP to GPGPU)

[76] had proposed a set of directives to be considered along with OpenMP directives [79]. Their work would translate such annotated code into CUDA.

OpenMPC does the following types of compiler optimizations for GPU memory access:

1. Techniques to optimize data movement between CPU and GPU. The authors developed two types of inter-procedural dataflow analysis to accomplish the following:

   (a) Overriding transfers to GPU global memory when the global memory already have up to date values of relevant variables

   (b) Overriding transfers to CPU from GPU memory if the relevant value wasn't used in the CPU part before it is written.

2. Techniques to enhance inter-thread locality

   (a) Parallel loop swap

   (b) Loop collapsing

OpenMPC also does optimizations via auto tuning. The auto tuner is a prototype that the authors built to automatically analyze the program and optional user settings.

4.3.4 PGI

The PGI Accelerator programming model [77] is a directive-based model targeting general hardware accelerators, even though it currently supports only CUDA GPUs. The PGI model allows very high-level abstraction similar to OpenMP; at the minimum level, the user just has to specify regions, called Accelerator Compute Regions, where loops will be compiled into accelerator kernels. The user can do this by inserting directives into the host program without any additional modication on the original code structures; all the other details, such as actual accelerator kernel-code generation, accelerator initialization, data transfers amid a host and an accelerator, or startup and shutdown of an accelerator, are handled by the PGI Accelerator compiler. The PGI model also allows users to provide additional information to the compilers, such as specification of accelerator's local data region, bounds of accessed arrays, guidance on mapping of loops onto an accelerator, and so on. The PGI Accelerator directives can be categorized as two types: directives for managing parallelism and those for managing data. The directives for parallelism guide types of parallelism to execute loops, and the ones for data deal with data traffic between the host and the accelerator. One good feature in the PGI Accelerator model is the data region. The data region sets boundaries where data are moved between the host and the accelerator; if a single data region encloses many compute regions, the compute regions can reuse the data already allocated on the accelerator. This can dramatically reduce the overhead of data movement and is important for optimized performance.

4.3.5 OpenACC

OpenACC, which is initiated by a consortium of CAPS, CRAY, PGI, and NVIDIA, is the first standardization effort toward a directive-based, general accelerator programming model portable across device types and compiler vendors. [80] tested the PGI version of OpenACC.

Like PGI Accelerator, OpenACC has two types of directives: directives for managing parallelism and directives for managing data. However, the OpenACC directives are further extended to express additional features not available in the PGI Accelerator model.

4.3.6 HMPP

HMPP [81] is another directive-based, GPU programming model targeting both CUDA and OpenCL. It also provides very high-level abstraction on GPU programming, similar to PGI Accelerator.

HMPP model is based on the concept of codelets, functions that can be remotely executed on hardware accelerators like GPUs. Because the codelet is a base unit containing computations to be offloaded to GPUs, porting existing applications using HMPP often requires manual modication of code structures, such as outlining of a code region or re-factoring existing functions. For optimized data management, HMPP uses the concept of a group of codelets.

By grouping a set of codelets, data in the GPU memory can be shared among different codelets, without any additional data transfers between CPU and GPU.

Combining the codelet grouping with HMPP advanced load/delegated store directives allows the same data-transfer optimizations as the data region in the PGI model. One unique feature in the HMPP model is that the same codelet can be optimized differently depending on its call sites.

4.3.7 R-Stream

R-Stream [78] is a high-level, architecture-independent programming model that is based on the polyhedral model [82]. It targets various architectures, such as STI Cell, SMP with OpenMP directives, Tilera, and CUDA GPUs.

R-stream performs affine scheduling to transform the code into both fine-grained and coarse grained parallelism. They choose not to enable multi-buffering since in GPUs, latency hiding is done automatically by hardware scheduler.

For global memory coalescing, in current implementation, the authors choose to not perform data layout transformations on an array residing in device memory (because they need to be carefully considered). They perform data re-layouts on transferred local copies of the arrays, at the time of the transfer. This mechanism can easily be extended to handle re-layouts in device memory at the time of copy from host to device memory.

They also transfer imperfect loop nests into perfect ones via loop interchange, strip-mining and fusion.

They also perform shared memory promotion and tiling. They perform tiling via a communication generation phase when there is a need to explicitly transfer data between different memories.

For enhanced memory utilization, privatization is performed so that each thread writes to its own variables.

Loop fusion and unrolling is done to reduce control overhead. Other techniques mentioned in paper are loop fission, loop interchange and strip-mining and data permutation, but it is not clear from the paper whether they apply them and in what context.

4.3.8 CUDA-CHiLL

CUDA-CHiLL accepts a collection of commands that is called a transformation recipe or a transformation strategy. This collection of commands provides a much more abstract level than CHiLL [14]. The Proposed high level transformations may be accompanied by low level CHiLL commands. Thus, each recipe is an individual strategy for modifying the code structure. Every line within a transformation recipe describes a transformation to be applied to the input code. Typically, CUDA-CHiLL abstractions combine many CHiLL commands in a single high-level command.

Although CUDA-CHiLL depends on command based transformations commands, applying these transformations still needs some optimization heuristics. For memory hierarchy optimizations, the authors used adaptations from [83].

The following list summarizes the performance heuristics implemented in CUDA-CHiLL:

- **Dependencies and Parallelization:** the authors permute the loops in the nest so that any loop carrying dependence is within a thread or a thread block. In the latter case, thread synchronization must be inserted. Loops representing blocks should not carry dependencies, as this would require costly global synchronization. Different levels of sub-loops within a loop nest can be parallelized as long as they use the same thread block size and proper synchronization are inserted.

- **Global Memory Coalescing:** All data is initially copied into global memory. Therefore, the authors select a loop order such that the x dimension for the thread index is linearly accessing the bulk of its data in global memory, resulting in coalesced access. If global memory coalescing is not possible due to interference with another array accessed in a different order, an array that is reused across threads may be copied by different threads into shared memory in a coalesced order, and accessed directly from shared memory.

- **Shared Memory and Bank Conflicts:** Data shared across threads, either as a result of the global memory access coalescing optimization above, or through significant inherent reuse, are placed in shared memory. Shared memory accesses need to avoid bank conflicts along the thread index, and two-dimensional arrays that are loaded into shared memory linearly along one dimension and accessed linearly in another dimension will require padding of one of the dimensions to avoid shared memory bank conflicts.

- **Maximize Reuse in Registers:** Registers provide low latency storage local to a thread, and data that are reused within a thread can benefit by being placed in registers. Due to the large register file size, tile sizes for register tiling are also good candidates for partitioning the computation across streaming multiprocessors, thus avoiding an additional level of tiling and overhead.

The authors also suggested using an auto-tuning framework to choose among resulting codes from various possibilities of parameters.

To understand the level of abstraction that each programming model/compiler provides, Table 4.10 summarizes the features and optimization approaches acquired in these compilers. Here, Explicit means that the feature is enabled by some user provided hints to the compiler, Implicit indicates that the compiler automatically handles the feature without user intervention, Indirect shows that the users can manually control the compiler to use the feature, and Imp-dep means that the feature is implementation dependent. The table shows that R-Stream provides the highest level of abstraction in comparison to other models/compilers found in literature as most of the features are handled implicitly in R-Stream. It also shows that hiCUDA and CUDA-CHiLL provide the lowest level of abstraction among other tools as the programmer has to control most of the features explicitly. However, lower level of abstraction may be beneficial in some cases that allow enough control over various optimizations and features specific to the underlying GPU architecture to achieve optimal performance. On the other hand, high level of abstraction sometimes limits the application coverage of the tool. RT-CUDA provides the same level of abstraction as R-Stream but also provides user-defined configurations to control various optimizations and features of the underlying GPU architecture to explore the effects of different kernel optimizations.

Features		CUDA-lite	hiCUDA	OpenMPC	PGI	OpenACC	HMPP	R-Stream	CUDA-CHiLL	RT-CUDA
Code regions to be offloaded		None	Structured blocks	Structured blocks	Loops	Structured blocks	Loops	Loops	Loops	Structured blocks
Loop Mapping		None	Parallel	Parallel	Parallel Vector	Parallel Vector	Parallel	Parallel	Explicit	Parallel
Data Management	GPU memory allocation and free Data movement between CPU and GPU	None	Explicit	Explicit/Implicit	Explicit/Implicit	Explicit/Implicit	Explicit/Implicit	Implicit	Explicit	Implicit
Compiler Optimizations	Loop Transformations Data Management Optimizations	Explicit Implicit	- Implicit	Explicit Explicit/Implicit	Implicit Explicit/Implicit	Imp-dep	Explicit Explicit/Implicit	Implicit	Explicit	Explicit/Implicit Implicit
GPU-specific features	Thread Batching Utilization of special memories	Implicit	Explicit	Explicit/Implicit	Indirect/Implicit	Indirect/Implicit Indirect/Imp-dep	Explicit/Implicit Explicit	Explicit/Implicit Implicit	Explicit None	Implicit Indirect/Explicit

Table 4.10: Summary of Features and Optimizations in Different Tools

4.4 RT-CUDA Design Specifications

To target the complex nature of the GPU architecture, programs often have to go through profound transformations. GPUs featured a massive number of active threads running in a machine that is primarily designed to process applications with abundant data parallelism. RT-CUDA is a software tool to integrate complex loop transformations for parallelism and flexible data movement to fully utilize the GPU architecture features that most impact performance. Decomposing the computational space is required to match the levels of parallelism on the GPU in addition to explicitly being aware of the two levels of parallelism of the GPU represented by the two grid dimensions and three block dimensions. The device memory is an explicit large flat memory with very small caches at the compute modules without cache coherence. To hide the latency of global memory fetches, some data may need to be copied to shared memory or registers. Synchronization across thread blocks is needed for many iterative solvers. The optimal optimization strategy may not be the obvious one, so various different versions of the program may need to be tested and evaluated using an auto-tuning procedure. The ability of invocation of optimized external Libraries offers substantial performance advantages over regular code translation. Following are the list of various optimization specifications that require programming efforts to assist the compiler in generating an efficient CUDA program that utilizes the GPU features in an effective manner to achieve best possible application performance. Figure 4.4 shows the RT-CUDA code transformation strategy to handle each of these optimization specifications, see Chapter 5 for code transformation details.

Input/Output GPU Memory Allocation

- Allocating memory for GPU input and output
- Explicit transfer of data between host (CPU) and device (GPU)

Computation Partitioning and Decomposition

- Nested loop computation and iteration space partitioning to match the GPU index space dimensions with two levels of parallelism that block-level and thread-level
- Parallel computations construction by subdividing the iteration space of a loop into blocks or tiles with a fixed maximum size to fit in the cache/shared memory

- Setting the optimal shape and size of the tile to take advantage of the target GPU memory architecture, maximizing reuse while maintaining a data footprint that meets memory capacity constraints

- De-constructing an iteration space into a control loop and tile loop. Also, perform related transformations such as array transformation, loop partitioning, and prefetching using shared memory

- Mapping threads to results (thread granularity), block organization and dimension, kernel organization and dimension

Locality optimizations and Datacopy Transformations

- Loop transformations that target the efficient use of the deep memory hierarchy such as copy of data into lower-latency portions of the GPU memory hierarchy as compared to global memory with the objectives of balancing thread computation time versus data transfer time

- Explicit copy of threads shared data in shared memory within the SM to get benefit from reuse and low-latency accesses from concurrently running threads

- Use of special portions of GM that are constant and texture memories to hide the high latency of GM fetches by utilizing constant and texture caches respectively

- Efficient utilization of large size of the register file (16K, 32K, or 64K register per SM depending on the GPU architecture) to exploit significant reuse within a thread

- The shared memory and the register bank in a SM are dynamically partitioned among the active thread blocks running on SM. Therefore, register and shared memory usages per thread block can be a limiting factor preventing full utilization of execution resources.

Parallel Memory Bandwidth

- GM is organized as a 16-module parallel memory. According to the interleaved storage scheme neighboring data elements are stored in contiguous parallel memories. Threads within the same warp should be mapped to access data that lies in distinct storages in the device memory so that multiple thread warp accesses are coalesced whenever possible in global memory

- Data accesses in shared memory by different threads in a warp are serialized if the data located in the same bank of memory. To avoid this serialization, threads within the same warp should be mapped to access data that lies in distinct memory banks to maximize data volume under the same access latency

- Data access requests to global memory could be reordered in parallel by multiple channels and banks. However, the memory bandwidth is efficiently utilized when the accesses to the memory channels are balanced, without congested channels

Optimization of Architectural Parameters

- The parameters are the number of results computed per thread, number of threads per block, number of blocks per kernel, and kernel dimension. Searching the optimal combination of parameters is based on determining their scope and searching the best combination by using analysis and auto-tuning
- Computing resource management and machine occupancy is subject to a complex set of constraints

Use of automatic compiler optimization and/or programmer-guided optimization

- Guide the compiler for applying specific optimizations to a scope of code such loop unrolling, kepler's data cache utilization, common-sub expression elimination, and etc.

Synchronization across SMs

- Synchronization across blocks is not supported directly in hardware, and is costly and must be avoided unless absolutely required for program correctness
- a low-cost barrier intra-block call permits synchronization between threads within a block

- CUDA model does not support global synchronization mechanisms
- Synchronization across thread blocks can be accomplished only by returning from a kernel call, after which all threads executing the kernel function are guaranteed to be finished, and global memory data modified by threads in different thread blocks are guaranteed to be globally visible

Invocation of Optimized external Libraries

- Some external libraries have been optimized at lower level programming and may deliver substantial performance advantages over regular code optimizations
- NVIDIA cuBLAS library is optimized for dense linear algebra, while cuSPARSE is its counterpart for sparse arrays
- Library details are hidden from the user
- Efficient invocation of external libraries require full understanding of its parameters and related implementation logic to select proper values for each parameter

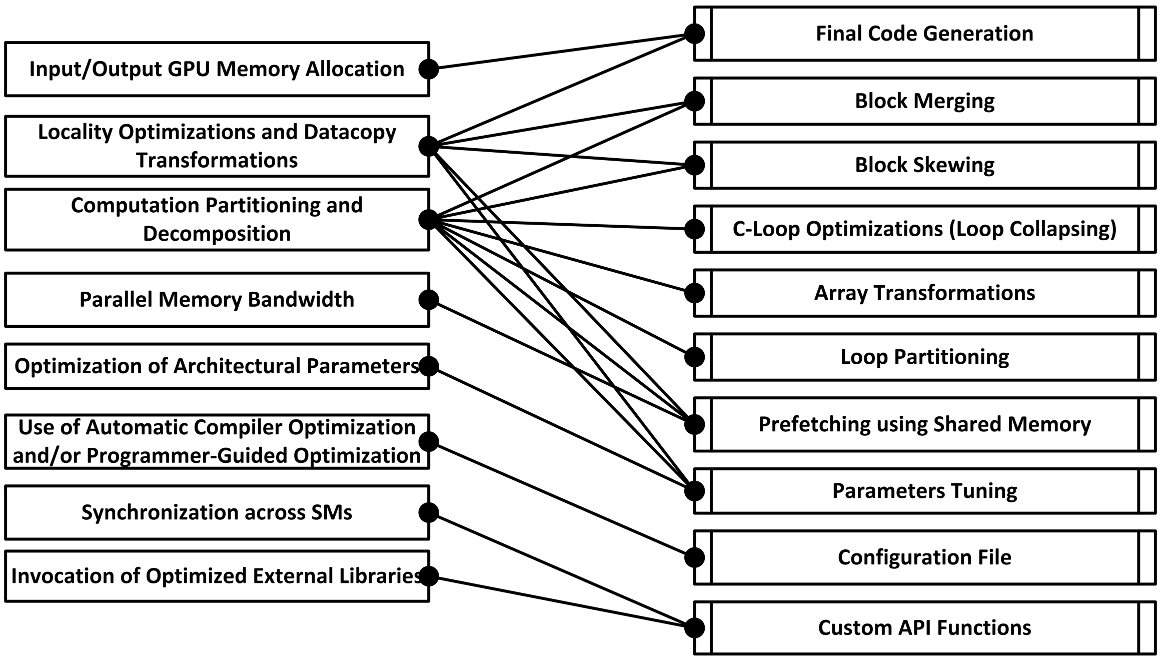

Input/Output GPU Memory Allocation
Locality Optimizations and Datacopy Transformations
Computation Partitioning and Decomposition
Parallel Memory Bandwidth
Optimization of Architectural Parameters
Use of Automatic Compiler Optimization and/or Programmer-Guided Optimization
Synchronization across SMs
Invocation of Optimized External Libraries
Final Code Generation
Block Merging
Block Skewing
C-Loop Optimizations (Loop Collapsing)
Array Transformations
Loop Partitioning
Prefetching using Shared Memory
Parameters Tuning
Configuration File
Custom API Functions

Figure 4.4: RT-CUDA Code Transformation Strategy

CHAPTER 5

CODE TRANSFORMATION

5.1 Review and Selection of Compiler Framework

The design and implementation of a compiler is not a trivial task, it requires tremendous amount of work and significant amount of patience to deal with developing quirks and hints in productions codes. Given the speed of the evolution of new computer architectures specifically in the field of High Performance Computing (HPC) and the amount of new languages that are being developed, a significant amount of development and effort would be required to design ad-hoc compilers to cover specific languages or to extend language features such as code optimizations.

5.1.1 YaCF

YaCF [84], Yet another Compiler Framework, is a compiler framework designed to create source to source translators, code analysis tools, or just to teach compiler technology without the need of learning large pieces of code. It has been developed in Python by taking advantage of its introspection capabilities and inherent code flexibility to ease the writing of source transformations or manipulations. Using a set of widely known object oriented patterns, implementing code transformations in YaCF is only a matter of writing a few lines of code. Several sub-classes, modules and packages have been included within YaCF to solve particular problems within source-to-source (StS) code translations. Following are the key characteristics of the framework:

- **A flexible parser:** to explore several different annotation schemes, language extensions and idioms, it provides a flexible front end where such modifications could be done efficiently.

- **Portability:** it is possible to use the compiler on several different platforms, from laptops to clusters. In addition, different users, such as students or collaborators, will be able to use it without having to invest too much time and energy in learning how to use it. The compiler is easily movable from one machine to another and it has been written in a common and portable language.

- **Debuggability:** the user will be able to run the StS process step by step or be able to show what each phase is doing at any given point. One of the potential uses of this compilation framework is to teach compiler technology, thus it an added feature to review or stop any process of the translation, so the user can easily see what is going on. As such, the inclusion of a graphical visualization of the Abstract Syntax Tree (AST) at any given moment would be a plus

- **Simplicity:** powerful but simple object oriented approach.

The main objective of YaCF is to develop StS transformations. It generates the intermediate representation of the original source code in the form of an augmented syntax tree (AST). A Symbol Table (ST) is used for information to augment the AST. YaCF components have been grouped together into three packages: FRONTEND, MIDDLEEND and BACKEND, through which the Internal Representation (IR) of YaCF is used. Figure 5.1 illustrates the overall StS transformation process.

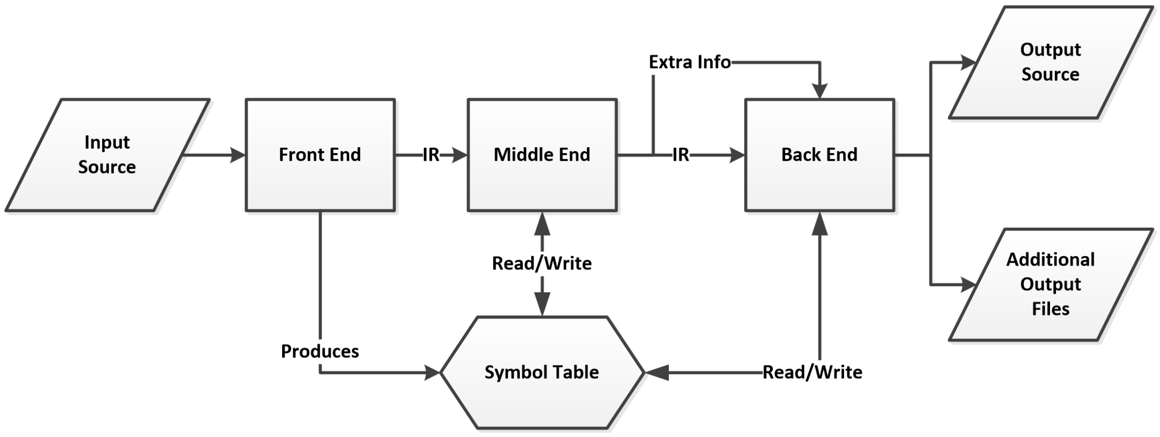


Figure 5.1: Overall translation workflow executed by a typical YaCF driver

YaCF parses the input source to generate an IR (Front End Process), performs the transformations (Middle End Process) and un-parsed (or re-written) the modified IR once again into the input language (Back End Process).

The code translation can be splitted into two separate steps implemented in two class hierarchies:

1. **Filter:**searching for a particular pattern or idiom in the code.

2. **Mutator:** applying the desired transformation on the nodes matching the criteria.

Complex transformations are performed using several nested Filters and Mutators that are grouped together into a Runner class with source storage facilities and code templates.

Examples:

Listing 5.1: A simple implementation of a Filter that will iterate through all the declarations of a given

```
class ExampleFilter(GenericFilterVisitor):
""" Returns the first node matching the example node
"""
def __init__(self):
   def condition(node):
      if type(node) == c99_ast.Decl:
         return True
            return False
super(ExampleFilter, self).__init__(condition_func =
   condition)
```

Listing 5.2: A more complex example of Filter where only those declarations inside a particular function

```
class ExampleFilter(GenericFilterVisitor):
""" Returns the first node matching the example node
"""
def __init__(self):
   self._inside_foo = False
   def condition(node):
      if type(node) == c99_ast.Decl \
          and self._inside_foo = True:
           return True
      return False
super(ExampleFilter, self).__init__(condition_func =
   condition)

def visit_FuncDef(self,node):
    if node.name == "foo":
       self._inside_foo = True
    self.generic_visit(node.body)
    self._inside_foo = False
```

Listing 5.3: Example of a Mutator that will apply a transformation to all declarations within a subtree

```
class ExampleMutator(AbstractMutator):
""" Apply a mutation
"""
def filter(self, ast):
   def is_decl:
      if type(node) == c_ast.Decl:
         return True
      return False
   return DeclFilter(ast, condition_func = is_decl)
def mutatorFunction(self, ast):
   # .... do something here with the matching node
   return ast
```

5.1.2 ANTLR

ANTLR [85] is a powerful parser generator that can be used to read, process, execute, or translate structured text or binary file. It can be used to build all sorts of languages, tools, and frameworks. Several commercial tools have been developed based on ANTLR such as Twitter search for query parsing, data warehouse and analysis systems for Hadoop, SQL Developer IDE and its migration tools in Oracle, and the NetBeans IDE parser for C++. From a formal language description called a grammar, ANTLR generates a parser for that language that can automatically build parse trees, which are data structures representing how a grammar matches the input. ANTLR also automatically generates tree walkers that you can use to visit the nodes of those trees to execute application-specific code.

Implementing Language in ANTLR

To implement a language, an application has to be built that reads sentences and reacts appropriately to the phrases and input symbols it discover. A language is a set of valid sentences, a sentence is made up of phrases, a phrase is made up of subphrases and vocabulary symbols. To react appropriately, the interpreter or translator has to recognize all of the valid sentences, phrases and subphrases of a particular language. Recognizing a phrase means we can identify the various components and can differentiate it from other phrases. After recognition, the application can perform a suitable operation for transformation/translation. Programs that recognize languages are called parsers or syntax analyzers. A grammar is just a set of rules, each one expressing the structure of a phrase. The ANTLR tool translates grammars to parsers that look remarkably similar to what an experienced programmer might build by hand. Grammars themselves follow the syntax of a language optimized for specifying other languages.

Parsing is done in two separate stages (as shown in Figure 5.2) that simulates the working of a human brain to read text:

1. **Lexical Analysis:** the process of grouping characters into words or symbols (tokens) is called lexical analysis or simply tokenizing. The program that tokenizes the input is called a lexer. It can group related tokens into token classes, or token types, such as INT (integers), ID (identifiers), FLOAT (floating-point numbers), and so on. The lexer groups vocabulary symbols into types when the parser cares only about the type, not the individual symbols. Tokens consist of at least two pieces of information: the token type (identifying the lexical structure) and the text matched for that token by the lexer.

2. **Parsing:** the second stage is the actual parser and feeds off of the generated tokens to recognize the sentence structure. By default, ANTLR generated parsers build a data structure called a parse tree or syntax tree that records how the parser recognized the structure of the input sentence and its component phrases.

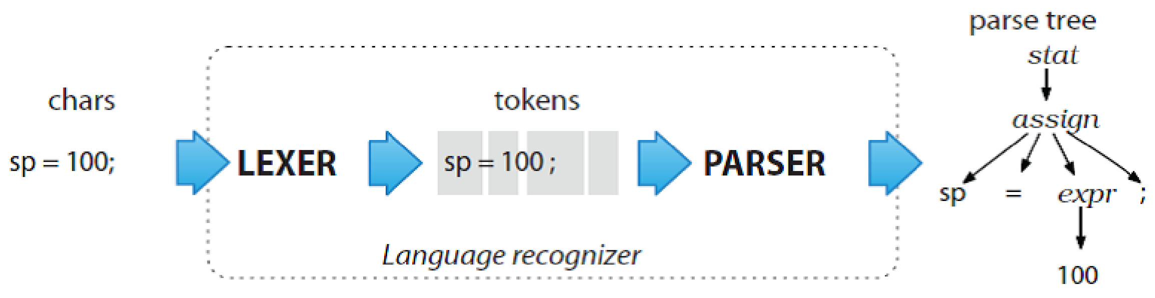


Figure 5.2: Parsing Stages

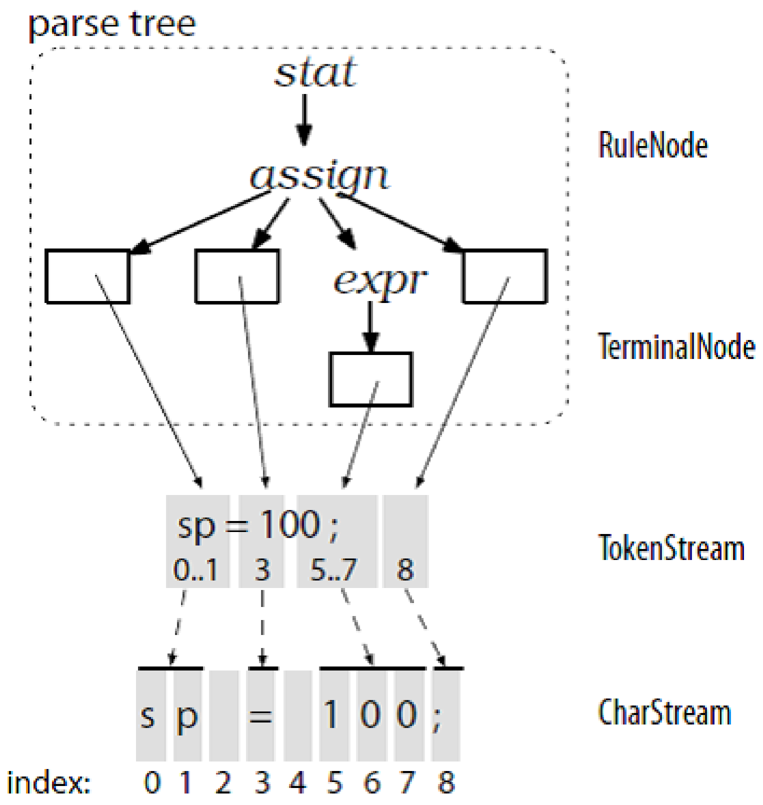
The interior nodes of the parse tree are phrase names that group and identify their children. The root node is the most abstract phrase name, in this case stat (short for “statement”). The leaves of a parse tree are always the input tokens. By producing a parse tree, a parser delivers a handy data structure to the rest of the application that contains complete information about how the parser grouped the symbols into phrases. Trees are easy to process in subsequent steps and are well understood by programmers. Better yet, the parser can generate parse trees automatically. By operating off parse trees, multiple applications that need to recognize the same language can reuse a single parser. The other choice is to embed application-specific code snippets directly into the grammar, which is what parser generators have done traditionally.

Parse trees are also useful for translations that require multiple passes (tree walks) because of computation dependencies where one stage needs information from a previous stage. In other cases, an application is just a heck of a lot easier to code and test in multiple stages because it’s so complex. Rather than re-parse the input characters for each stage, we can just walk the parse tree multiple times, which is much more efficient.

Building Language Applications using Parse Trees

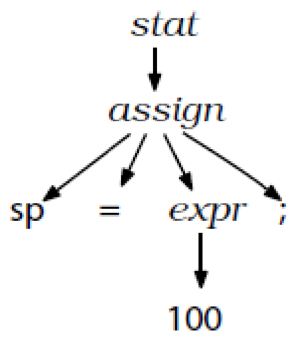
To make a language application, we have to execute some appropriate code for each input phrase or subphrase. The easiest way to do that is to operate on the parse tree created automatically by the parser. The nice thing about operating on the tree is that we’re back in familiar Java territory. There’s no further ANTLR syntax to learn in order to build an application.

As explained in the previous section, lexers process characters and pass tokens to the parser, which in turn checks syntax and creates a parse tree. The corresponding ANTLR classes are CharStream, Lexer, Token, Parser, and ParseTree. The "pipe" connecting the lexer and parser is called a TokenStream. The diagram below illustrates how objects of these types connect to each other in memory.

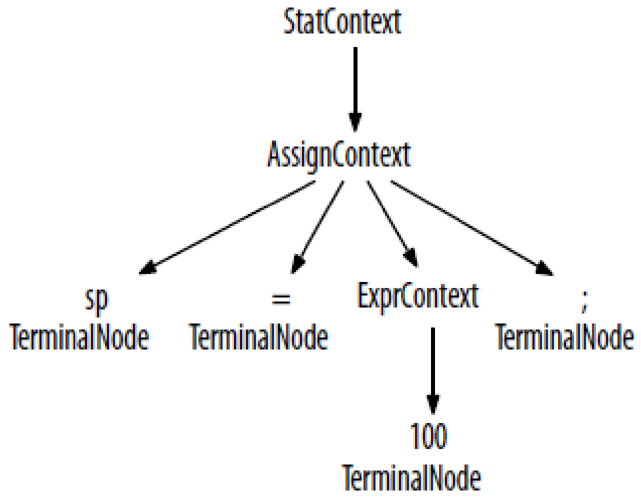


These ANTLR data structures share as much data as possible to reduce memory requirements. The diagram shows that leaf (token) nodes in the parse tree are containers that point at tokens in the token stream. The tokens record start and stop character indexes into the CharStream, rather than making copies of substrings. There are no tokens associated with whitespace characters (indexes 2 and 4) since we can assume our lexer tosses out whitespace.

The figure also shows ParseTree subclasses RuleNode and TerminalNode that correspond to subtree roots and leaf nodes. RuleNode has familiar methods such as getChild() and getParent(), but RuleNode isn't specific to a particular grammar. To better support access to the elements within specific nodes, ANTLR generates a RuleNode subclass for each rule. The following figure shows the specific classes of the subtree roots for an assignment statement example, which are StatContext, AssignContext, and ExprContext:



Parse tree



Parse tree node class names

These are called context objects because they record everything we know about the recognition of a phrase by a rule. Each context object knows the start and stop tokens for the recognized phrase and provides access to all of the elements of that phrase. For example, AssignContext provides methods ID() and expr() to access the identifier node and expression subtree.

Given this description of the concrete types, one could write code by hand to perform a depth-first walk of the tree. Typical operations are things such as computing results, updating data structures, or generating output. Rather than writing the same tree-walking boilerplate code over again for each application, though, the tree-walking mechanisms can be used that ANTLR generates automatically.

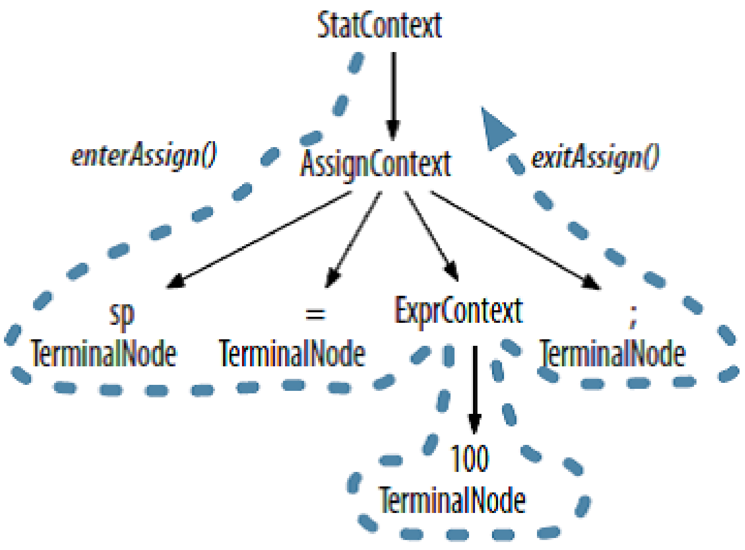
Parse-Tree Listeners and Visitors

ANTLR provides support for two tree-walking mechanisms in its runtime library. By default, ANTLR generates a parse-tree listener interface that responds to events triggered by the built-in tree walker. The listeners themselves are exactly like SAX document handler objects for XML parsers. SAX listeners receive notification of events like startDocument() and endDocument(). The methods in a listener are just callbacks that are used to respond to a checkbox click in a GUI application.

Parse-Tree Listeners

To walk a tree and trigger calls into a listener, ANTLR's runtime provides class ParseTreeWalker. To make a language application, a ParseTreeListener implementation can be built containing application-specific code that typically calls into a larger surrounding application.

ANTLR generates a ParseTreeListener subclass specific to each grammar with enter and exit methods for each rule. As the walker encounters the node for rule assign, for example, it triggers enterAssign() and passes it the AssignContext parse-tree node. After the walker visits all children of the assign node, it triggers exitAssign(). The tree diagram shown below shows ParseTreeWalker performing a depth-first walk, represented by the thick dashed line.



It also identifies where in the walk ParseTreeWalker calls the enter and exit methods for rule assign. (The other listener calls aren't shown.)

And the diagram in Figure 5.3 shows the complete sequence of calls made to the listener by ParseTreeWalker for an statement tree.

The beauty of the listener mechanism is that it's all automatic. Programmers don't have to write a parse-tree walker, and the listener methods don't have to explicitly visit their children.

Parse-Tree Visitors

In addition to parse-tree listener, ANTLR generates a visitor interface from a grammar with a visit method per rule to control the walk itself. Here's the familiar visitor pattern operating on the parse tree:

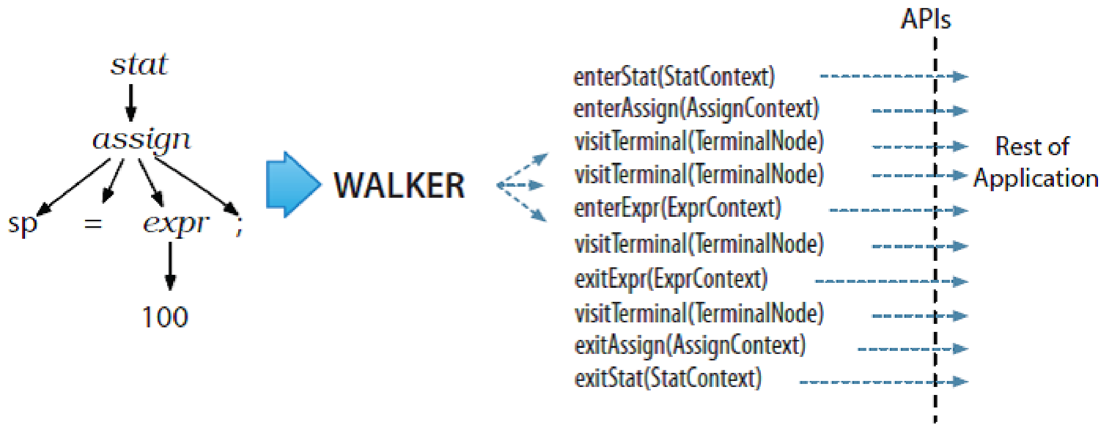
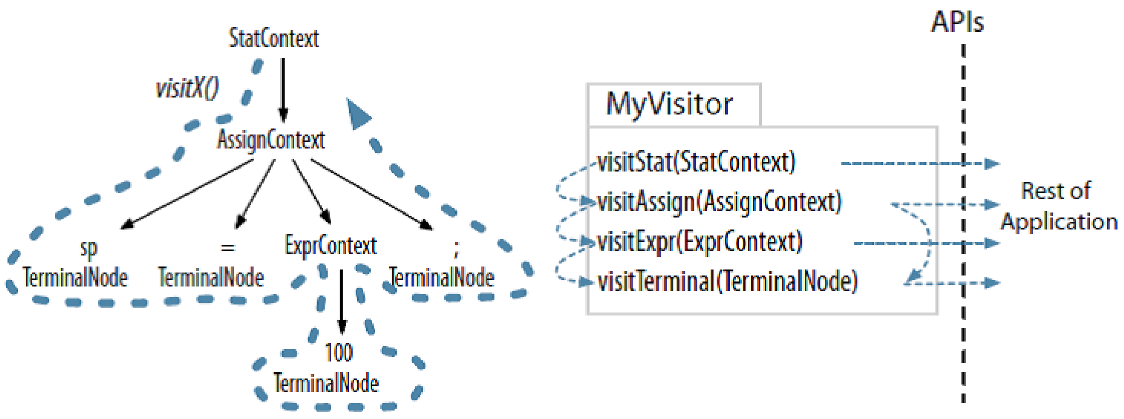

stat
assign
sp
=
expr
;
100
WALKER
APIs
enterStat(StatContext)
enterAssign(AssignContext)
visitTerminal(TerminalNode)
visitTerminal(TerminalNode)
enterExpr(ExprContext)
visitTerminal(TerminalNode)
exitExpr(ExprContext)
visitTerminal(TerminalNode)
exitAssign(AssignContext)
exitStat(StatContext)
Rest of Application

Figure 5.3: ParseTreeWalker call sequence



The thick dashed line shows a depth-first walk of the parse tree. The thin dashed lines indicate the method call sequence among the visitor methods. To initiate a walk of the tree, the application-specific code would create a visitor implementation and call visit().

Listing 5.4: Parse Tree Visitor Example

```
ParseTree tree = ... ; // tree is result of parsing
MyVisitor v = new MyVisitor();
v.visit(tree);
```

ANTLR's visitor support code would then call visitStat() upon seeing the root node. From there, the visitStat() implementation would call visit() with the children as arguments to continue the walk. Or, visitMethod() could explicitly call visitAssign(), and so on.

5.1.3 Framework Selection

Both YaCF and ANTLR provides a complete functionality to write new compilers and translators. We have found ANTLR more convenient to use for our proposed code transformation. The selection of ANTLR is based on the following reasons:

- ANTLR generates lexer and parser based on standard grammar in Java which is easy to understand and modify. While YaCF produces its own internal representation that have to be used for code transformations.
- ANTLR parser generates a parse tree based on the standard grammar and provides classes to traverse the parse tree. So, the parse tree generated through ANTLR can be modified based on the required code transformations. While YaCF generates an Aug-mented Syntax Tree (AST) based on the own internal representation which cannot be modified directly.
- ANTLR provides methods for entry and exit of each parse tree node that can be overridden to implement code transformations by using standard Java APIs and code structures. While YaCF provides the concept of Filter and Mutator classes to search the pattern and implement code transformations by using a specific code structure and local API functions.

- In general, ANTLR provides a generic and easy to understand framework to implement code transformations in standard Java language while YaCF has its own class structures to implement the code transformation that need to be learnt by the programmer to use the framework.

5.2 RTA-CUDA Description

RTA-CUDA (Restructuring Tool Algorithm for CUDA) has been developed to be implemented in a source-to-source transformation tool to convert a standard C function (input) containing computational loops into an optimized CUDA kernel (output) based on some user-defined variables. The algorithm consists of the following steps as shown in Fig. 5.4:

5.2.1 C-Loop Optimizations (Loop Collapsing)

Merge the nested loops if they are independent and calculate array indices based on the new loop variable. For example, if i and j are two independent loops such that i = [0 to N] and j = [0 to N]. The new loop index (idx) will be equal to [0 to N * N] and i, j will be the quotient and remainder of the division of idx with N respectively such that 'i' represents row of the matrix and 'j' represents column of the matrix. So, instead of two nested loops, we will have now one main loop.

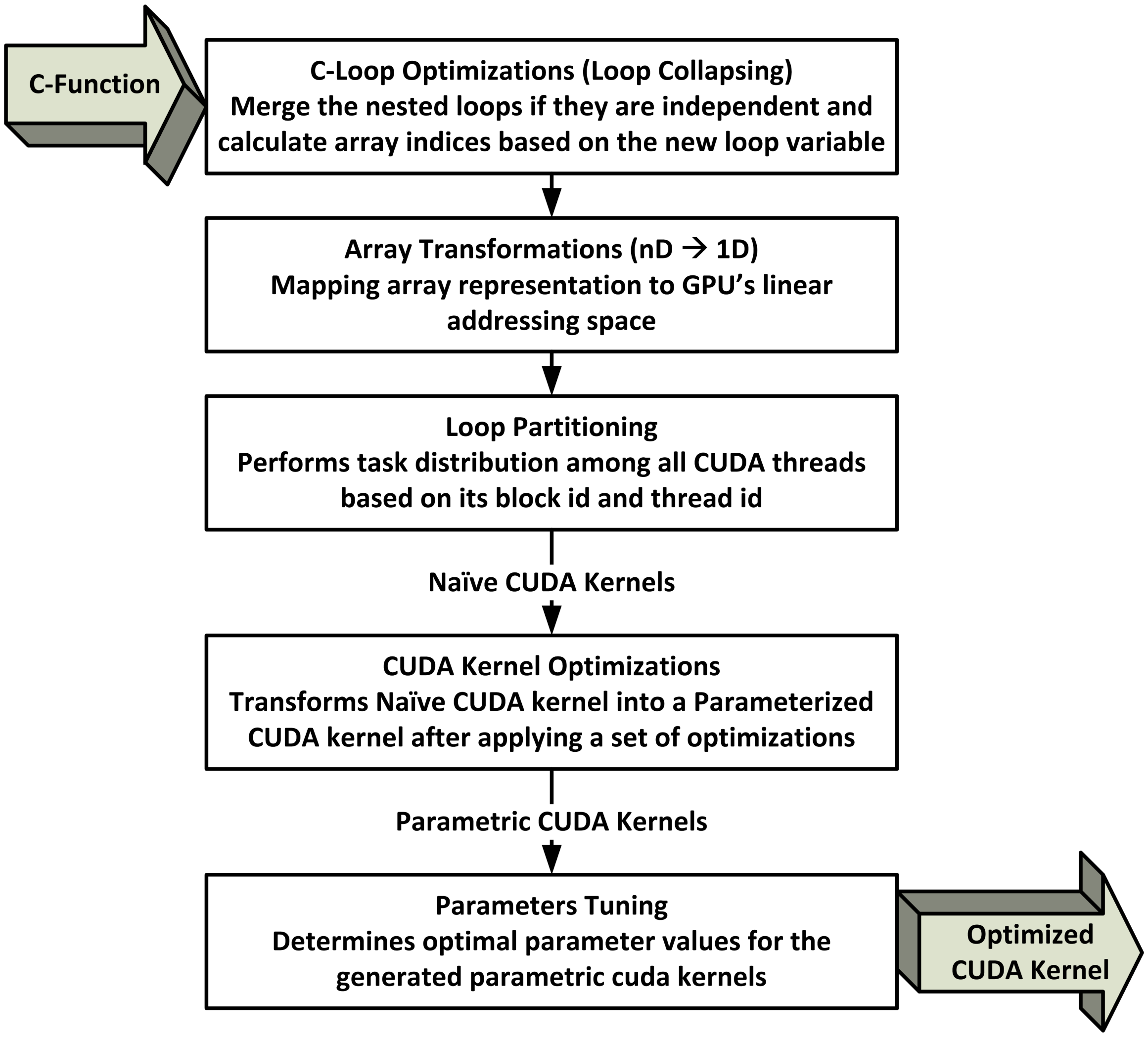

C-Function
C-Loop Optimizations (Loop Collapsing)
Merge the nested loops if they are independent and calculate array indices based on the new loop variable
Array Transformations (nD → 1D)
Mapping array representation to GPU's linear addressing space
Loop Partitioning
Performs task distribution among all CUDA threads based on its block id and thread id
Naïve CUDA Kernels
CUDA Kernel Optimizations
Transforms Naïve CUDA kernel into a Parameterized CUDA kernel after applying a set of optimizations
Parametric CUDA Kernels
Parameters Tuning
Determines optimal parameter values for the generated parametric cuda kernels
Optimized CUDA Kernel

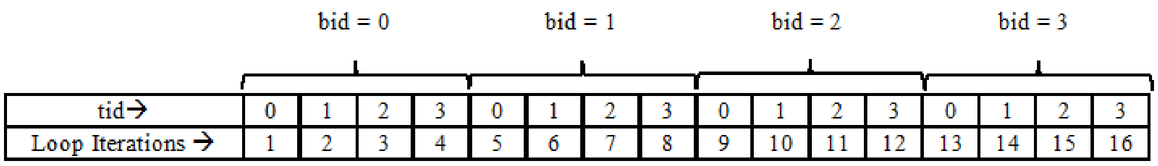
Figure 5.4: Restructuring Tool Algorithm

5.2.2 Array Transformation (nD → 1D)

In GPUs, device memory can be allocated only as linear memory so CUDA arrays are restricted to be allocated as 1D arrays while standard C language supports multi-dimensional arrays. In this step, all the multi-dimensional array accesses in the expressions are converted to linear array representations. For example, C[i][j] will be represented as C[i * N + j] where N is the width of the array.

5.2.3 Loop Partitioning

Partition the main loop among all cuda threads. This obtains task distribution among all cuda threads based on its block id and thread id. Fig. 5.5 shows the task distribution among 4 cuda blocks with 4 threads per block. Each cuda thread identifies its working element of the resultant array using the block id and thread id within the cuda thread block. At this stage, each thread is mapped to one element of the resultant array. So, the loop is replaced by the statements to calculate the loop index to generate a Naïve CUDA Kernel.



	bid = 0				bid = 1				bid = 2				bid = 3			
tid→	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
Loop Iterations →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 5.5: Task Distribution among all threads

5.2.4 CUDA Kernel Optimizations

In this step, each of the generated Naïve CUDA Kernel is transformed into a Parameterized CUDA Kernel after applying a set of optimizations as shown in Fig. 5.6.

Block Merging

At this stage, each thread block is mapped to one block of resultant matrix/vector. Each thread within the block calculates one element of the resultant. To increase the thread granularity, each thread block can be mapped to multiple resultant blocks vertically. The number of blocks to be merged is defined as an input

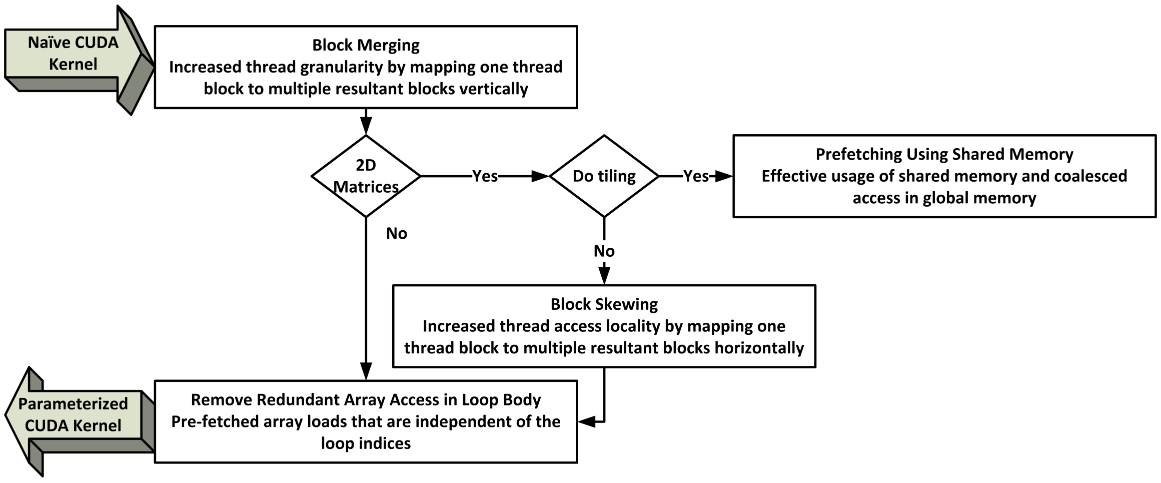

Naïve CUDA Kernel
Block Merging
Increased thread granularity by mapping one thread block to multiple resultant blocks vertically
2D Matrices
Yes
Do tiling
Yes
Prefetching Using Shared Memory
Effective usage of shared memory and coalesced access in global memory
No
No
Block Skewing
Increased thread access locality by mapping one thread block to multiple resultant blocks horizontally
Parameterized CUDA Kernel
Remove Redundant Array Access in Loop Body
Pre-fetched array loads that are independent of the loop indices

Figure 5.6: CUDA Kernel Optimizations in RTA-CUDA

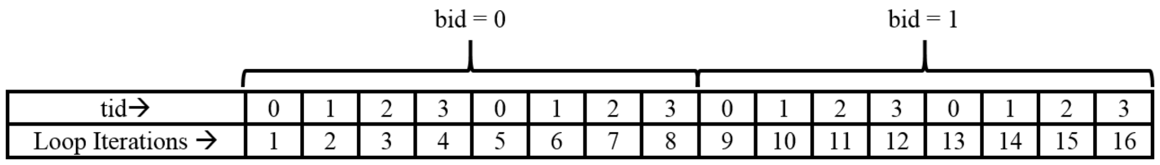
parameter and the optimal value of block merging can be obtained after running the parameter tuning algorithm as explained in section 5.2.5. Fig. 5.7 shows the task distribution among 2 cuda blocks with 4 threads per block after merging 2 resultant blocks that is each thread calculates two resultant elements at the same offset in consecutive resultant blocks. This is done by the following steps:

1. Convert the resultant variable into an array stored in local memory to compute the multiple elements simultaneously in a pipelined fashion

2. Replace the first index of resultant matrix with the increment of loop index m where m defines the number of elements to be calculated by each thread

3. Replace the resultant variable into array with loop index m

4. Update row index calculations with multiple of number of blocks to be merged



tid→	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3
Loop Iterations →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 5.7: Task Distribution among all threads with block merging applied

Prefetching using Shared Memory

To effectively use the shared memory and do coalesced access in global memory. The matrices in the loop that are going to be access by each thread in row-major can be tiled and loaded into shared memory with coalesced access. This is done by the following steps:

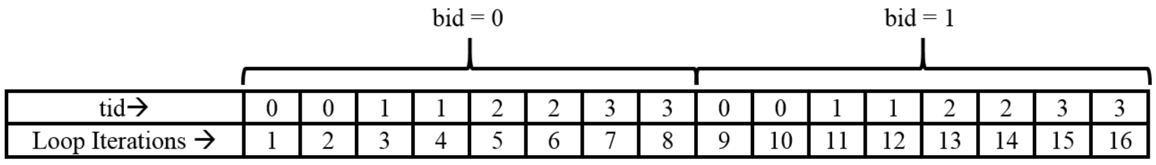
1. Declare shared variable for the tile of the given matrix to load the tiled rows of the number of blocks merged in the previous step
2. Load the tile of the given matrix into shared memory by accessing the tiled rows in a coalesced manner such that each thread of the block access the consecutive elements in the same row
3. Add barrier to synchronize all threads (__syncthreads()) after loading the tile
4. Replace array access in the loop with the shared tile
5. Add barrier to synchronize all threads (__syncthreads()) after calculating the tile
6. Load the tile next consecutive tile of the given matrix into shared memory to be used in the computation of next iteration

7. Add barrier to synchronize all threads (__syncthreads()) after calculating the tile

8. Modify the main loop to traverse all tiles of the given matrix

9. Calculate the remaining tile loaded in the last iteration.

10. unroll the loops to load and calculate the tile.



tid→	0	0	1	1	2	2	3	3	0	0	1	1	2	2	3	3
Loop Iterations →	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 5.8: Task Distribution among all threads with block skewing applied

Block Skewing

To increase the thread access locality, each thread block can be mapped to multiple resultant blocks horizontally. The number of blocks to be skewed is defined as an input parameter and the optimal value of block skewing can be obtained after running the parameter tuning algorithm as explained in section 5.2.5. Fig. 5.8 shows the task distribution among 2 cuda blocks with 4 threads per block after skewing 2 resultant blocks that is each thread calculates two resultant elements at the consecutive offset in the resultant blocks. This is done by the following steps:

1. Convert the resultant variable into a matrix stored in local memory to hold the results of merged and skewed elements

2. Replace the second index of resultant matrix with the increment of loop variable n where n defines the number of elements skewed in a resultant block

3. Add second dimension of the resultant variable with loop index n

4. Update column index calculations with multiple of number of blocks to be skewed

Remove Redundant Array Access in Loop Body

At this stage, check for repeated load access of array elements within the newly created merged and skewed loops (m and n) in sections 5.2.4 and 5.2.4. If such an access is detected then replace it with local variable and move the loading of this element prior to the loop.

These steps generate a parameterized CUDA kernel with three parameters that are BLOCKSIZE (number of threads per block), MERGE_LEVEL (number of blocks to be merged), and SKEW_LEVEL (number of blocks to be skewed). The optimal values of these parameters can be obtained using the parameter tuning algorithm as explained in the following section 5.2.5.

5.2.5 Parameters Tuning Algorithm

Algorithm 1 Parameters Tuning Algorithm

findOptimalParameters(N, CC)

Parameters:

N = Total Number of Elements in the Resultant Matrix

CC = Compute Capability of GPU Device

Constants and Keywords:

params = Structure of GPU Parameters

minBS = Minimum BLOCKSIZE, maxBS = Maximum BLOCKSIZE

minML = Minimum MERGE_LEVEL, maxML = Maximum MERGE_LEVEL

minSL = Minimum SKEW_LEVEL, maxSL = Maximum SKEW_LEVEL

KB = Kernel Blocks

RPT = Registers Per Thread

ShM = Shared Memory Per Block

RPB = Registers Per Block

WPB = Warps Per Block

ABW = Active Blocks Limit based on WPB

ABShM = Active Blocks Limit based on ShM

ABR = Active Blocks Limit based on RPB

CompleteParamsList = Structure Array for all Possible Kernel Parameters

CandidateParamsList = Structure Array for Candidate Kernel Parameters

OptimalParams = Structure for final Optimal Kernel Parameters

Algorithm:

```
1: Load params for compute capability of CC
```

```
2: for bs=minBS to maxBS Step *2 do
3:     if $N \mod bs \neq 0$ then
4:         continue
5:     end if
6:     for ml=minML to maxML Step *2 do
7:         for sl=minSL to maxSL Step *2 do
8:             KB = INT(N/bs/ml/sl)
9:             if KB = 0 then
10:                continue
11:            end if
12:            Compile the kernel only to determine the required $RPT$ and $ShM$
13:            RPB = ROUND(RPT x bs, params.RegisterAllocationUnitSize)
14:            WPB = CEILING(bs/params.ThreadsPerWarp)
15:            ABW = FLOOR(params.WarpsPerSM/WPB)
16:            ABShM = FLOOR(params.MaxSharedMemoryPerBlock/ShM)
17:            ABR = FLOOR(params.RegisterFileSize/RPB)
18:            Add $< bs, ml, sl, ABW, ABShM, ABR >$ into CompleteParam-
  sList
19:        end for
20:    end for
21: end for
```

```
22: for all p in CompleteParamsList do
23:     if $p.ABW > 0$ and $p.ABShM > 0$ and $p.ABR > 0$ then
24:         Add p into CandidateParamsList
25:     end if
26: end for
27: mintime = 0
28: for all p in CandidateParamsList do
29:     Execute the kernel with BLOCKSIZE=p.bs, MERGE_LEVEL=p.ml,
    SKEW_LEVEL=p.sl
30:     determine execution time (ktime) of the kernel
31:     if $ktime > 0$ and $(mintime = 0$ or $mintime > ktime)$ then
32:         mintime = ktime
33:         OptimalParams = p
34:     end if
35: end for
```

Algorithm 1 determines the optimal parameters (BLOCKSIZE, MERGE_LEVEL, and SKEW_LEVEL) for the generated parametric CUDA kernel. The size of cuda grid will be determined by dividing the total number of elements in the resultant array with the product of all three parameters.

The algorithm evaluates the generated parametric kernel with various possible combinations of BLOCKSIZE, MERGE_LEVEL and SKEW_LEVEL. The pruning of the list of possible parameters is used at three levels to reduce the repeated compilation and execution of the kernel. The three levels of pruning are as follows:

1. **Array Block Level:** This will skip those values of BLOCKSIZE which does not equally distribute the number of resultant elements among all threads (see step 3).

2. **Kernel Block Level:** This will skip those values of MERGE_LEVEL and SKEW_LEVEL which does not distributed the number of resultant elements among all kernel blocks (see step 9).

3. **Active Block Level:** This will skip those combinations of parameters which requires more than the available resources such as number of registers, shared memory and number of threads per SM (see step 31).

The algorithm takes kernel source file, number of resultant elements (N) and GPU Compute Capability (CC) for the target GPU device. It first loads the parameters for the given compute capability such as Register Allocation Unit Size, Threads Per Warp, Warps Per SM, Maximum Shared Memory Per Block, Register File Size, and etc (see step 1). It then loop over all possible combination of BLOCKSIZE, MERGE_LEVEL, and SKEW_LEVEL limiting to the range given by user with appropriate pruning (Array Block Level and Kernel Block Level) of the parameters as explained above. For each combination, it compiles the kernel with ptx information to determine the required number of Registers Per Thread (RPT) and Shared Memory (ShM) per block (see step 12). Then, calculate and store the restricted number of Active Blocks by Warp (ABW), Active Blocks by

Shared Memory (ABShM), and Active Blocks by Registers (ABR) into a structured list (CompleteParamsList) (see steps 13 - 18). Then, it performs parameters pruning at Active Block Level and generate a list of possible optimal parameters (CandidateParamsList) (see steps 22 - 26). Finally, it execute the kernel for each combination of parameters in CandidateParamsList and determine the final optimal parameters (OptimalParams) that gives the minimum execution time (see steps 28 - 1).

5.3 RT-CUDA Design

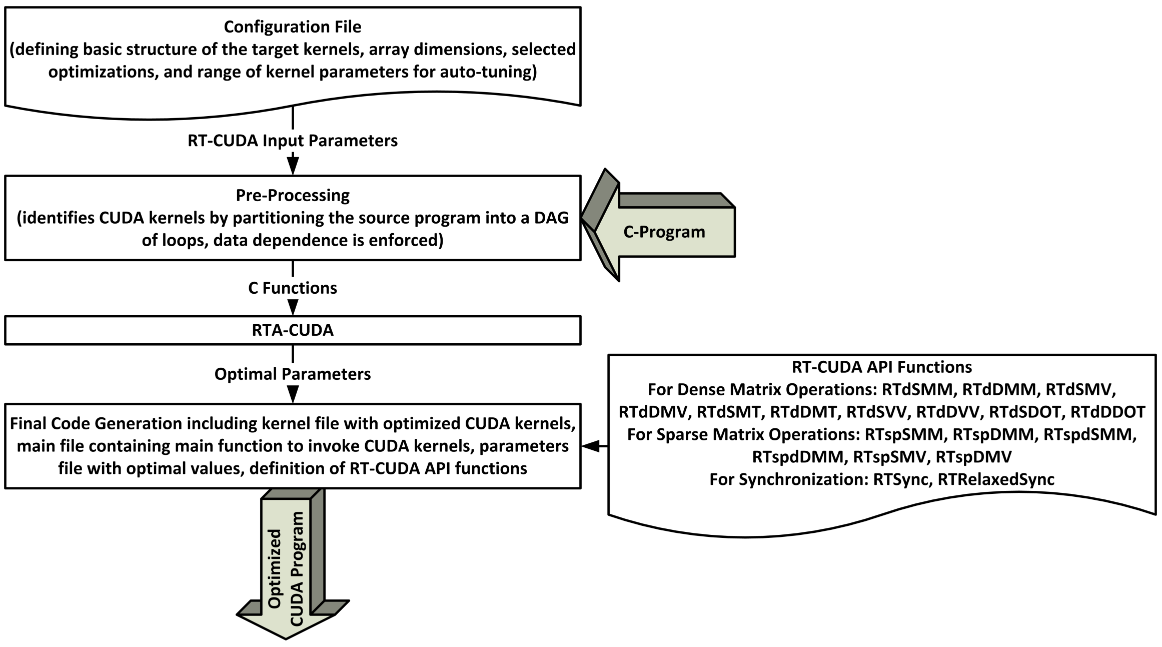

Configuration File
(defining basic structure of the target kernels, array dimensions, selected optimizations, and range of kernel parameters for auto-tuning)
RT-CUDA Input Parameters
Pre-Processing
(identifies CUDA kernels by partitioning the source program into a DAG of loops, data dependence is enforced)
C-Program
C Functions
RTA-CUDA
Optimal Parameters
Final Code Generation including kernel file with optimized CUDA kernels, main file containing main function to invoke CUDA kernels, parameters file with optimal values, definition of RT-CUDA API functions
RT-CUDA API Functions
For Dense Matrix Operations: RTdSMM, RTdDMM, RTdSMV, RTdDMV, RTdSMT, RTdDMT, RTdSVV, RTdDVV, RTdSDOT, RTdDDOT
For Sparse Matrix Operations: RTspSMM, RTspDMM, RTspdSMM, RTspdDMM, RTspSMV, RTspDMV
For Synchronization: RTSync, RTRelaxedSync
Optimized CUDA Program

Figure 5.9: Restructuring tool design

RT-CUDA is a source-to-source transformation tool that is capable to convert a standard C-Program (input) into an Optimized CUDA Program (output). The overall transformation is driven by some user-defined directives and API function calls provided in the tool with automatic kernel optimizations. The user is also able to include/exclude any of the optimizations available in the transformations. The tool follows the steps as shown in Fig. 5.9 to generate an Optimized CUDA Program.

1. **Pre-Processing:** In this step, the source program is partitioned into a DAG (Dynamic Acyclic Graph) of loops identified as candidate CUDA kernels to be executed on GPU while separating the set of scalar segments that will be executed on host. Data dependence among the loops is enforced in the generated code. By examining the DAG, loops that are data independent can be merged together to reduce the exit/entry of kernels, copying data between Global Memory (GM) and Shared Memory (ShM), and to reduce loop overhead.

2. **RTA-CUDA:** This step will take each of the functions generated in the pre-processing step based on the loops identified as candidate CUDA kernels and apply RTA-CUDA algorithm as explained in section 5.2 to generate the optimized CUDA kernels.

3. **Final Code Generation:** At the end, generate the optimized CUDA program including all the optimized CUDA kernels obtained in RTA-CUDA step.

In addition to this, RT-CUDA also provides the functionality of calling external library functions such CUBLAS and cuSparse for some of the dense and sparse matrix operations respectively. These can be included by calling a related API function defined in the tool. Table 5.1 shows the list of available functions.

API Function	Data Precision	Matrix Operation
RTdSMM	Single	Dense-Dense Matrix Multiplication
RTdDMM	Double	
RTdSMV	Single	Dense-Dense Matrix Vector Multiplication
RTdDMV	Double	
RTdSMT	Single	Dense Matrix Transposition
RTdDMT	Double	
RTdSVV	Single	Dense-Dense Vector Multiplication
RTdDVV	Double	
RTdSDOT	Single	Dot Product of Two Dense Vectors
RTdDDOT	Double	
RTspSMM	Single	Sparse-Sparse Matrix Multiplication
RTspDMM	Double	
RTspdSMM	Single	Sparse-Dense Matrix Multiplication
RTspdDMM	Double	
RTspdSMV	Single	Sparse-Dense Matrix Vector Multiplication
RTspdDMV	Double	

Table 5.1: Available API Functions in RT-CUDA for Dense and Sparse Matrix Operations

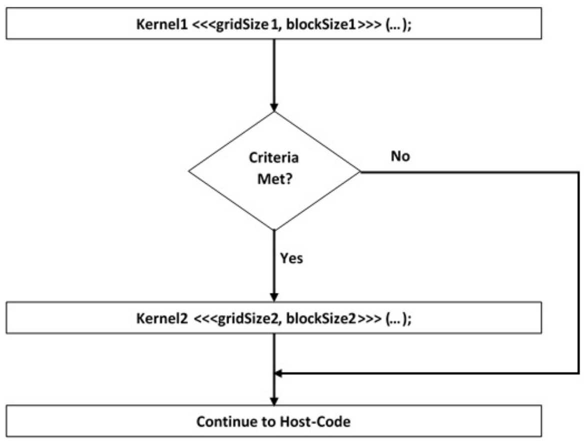

Kernel1 <<<gridSize1, blockSize1>>> (...);
Criteria Met?
No
Yes
Kernel2 <<<gridSize2, blockSize2>>> (...);
Continue to Host-Code

Figure 5.10: CPU-based Synchronization

RT-CUDA also supports inter-block synchronization in three ways:

1. **CPU Synchronization:** This is the simplest approach recommended by Nvidia [51] for inter-block synchronization by exiting and re-entering the kernel that is considered as an implicit synchronization. This is done by defining separate CUDA kernels for each of the dependent loops and calling them in sequence from host. Fig. 5.10 shows the flowchart of the CPU-based

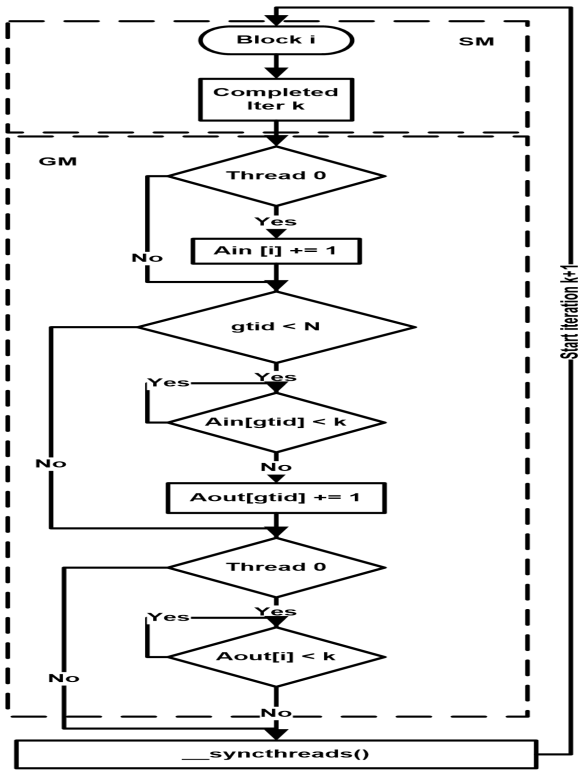


Figure 5.11: Lock-free Synchronization

synchronization.

2. **Lock-Free Synchronization:** This synchronization primitive is based on the work in [50]. Fig. 5.11 shows the main idea of the lock-free synchronization. It uses two arrays, named Ain and Aout, of length N for synchronization. When all threads of a block finish their work, the first thread of each block increments its location in the Ain array. Then, the first N threads of the first block in parallel check whether all blocks have written to their corresponding location in the Ain array. If so, these N threads write in parallel to the Aout array to inform other threads that the threads of this block have reached the synchronization point. Meanwhile, the first thread of each block continuously checks its location in the Aout array until the

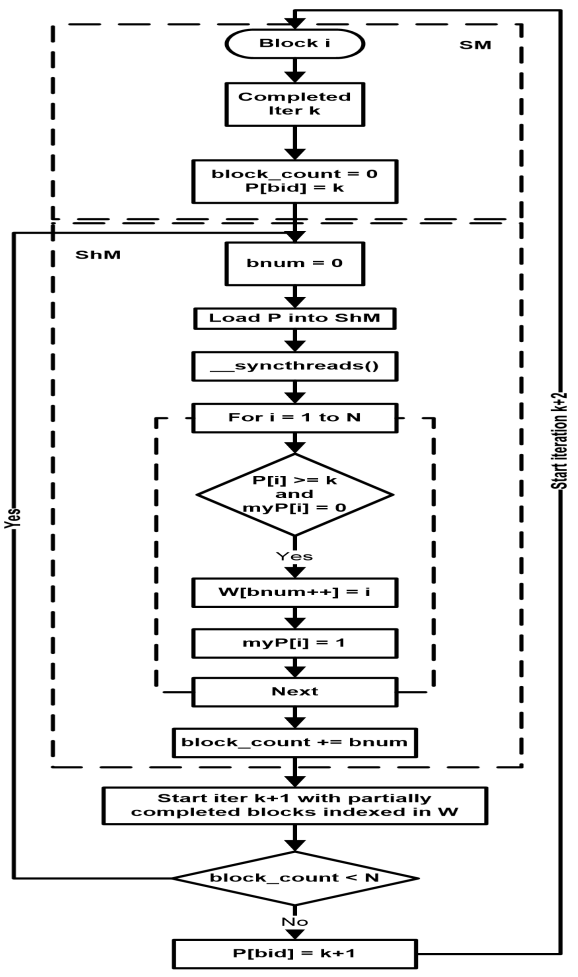


Figure 5.12: Relaxed Synchronization

value is set to k where k is the iteration number.

3. **Relaxed Synchronization:** We have developed a new synchronization primitive that can be useful in implementing iterative solvers with block dependencies among each iterations. Fig. 5.12 shows the flowchart of the relaxed synchronization. This approach overlaps the computation of two consecutive iterations. After completion of iteration 'k', each block start the computation of the iteration 'k+1' using the completed blocks of dependent array by the previous iteration. Each block updates its designated

element in presence vector 'P' in global memory with the iteration number at the end of each iteration. So for the next iteration, it will call the relaxed synchronization primitive that will check the presence vector for the completed blocks of the previous iteration and return a work vector 'W' and the number of completed blocks 'bnum' that can be used to start the computation of the next iteration using the completed blocks of the previous iteration. The presence vector will be first loaded into shared memory from global memory with coalesced access to reduce global memory loads.

5.4 RT-CUDA Implementation

To implement RT-CUDA transformations, we have used an innovative approach of source-to-source transformation of a computer program, will be helpful in fast development of source code translators to convert programs written in one language into another program in another language. Using ANTLR, a parser is generated based on a defined grammar, which describes a parse tree that consists of all possible nodes (rules) with their entry and exit. The parser takes a source code (S) and generates its corresponding parse tree PT(S). Also the parser produces a generated parse tree walker which traverses PT(S) and applies the grammar rules. To carry out the code transformation, a Parse Tree Walker traverses PT(S) with overloaded Listener class instance. The method in the listener class modifies the nodes or add/delete payloads. An event method for adding space after every type specifier is implemented so the parse tree can be generated from the transformed code by updating the spaces with the new transformation. Modifying the parse tree is much easier than putting actions into the grammar because every grammar has different structure and needs to be understood completely before putting actions into it. While, in our approach the modification is done at the tree structure, which is commonly used data structure in programming languages and known to every programmer.

The method of code transformation includes following steps, as shown in Fig. 5.13:

1. **Parser Generator:** In this initial step, a parser code is generated based on the given C grammar (see Appendix A) using ANTLR providing functions for enter and exit of all possible nodes in a parse tree for the grammar. This also generates a Parse Tree Walker class that can traverse a parse tree of any given code following the rules in the grammar. The Parse Tree Walker raise related node events at the entry and exit of a particular node in the parse tree.

2. **Parse Tree Generation:** the parser takes the source code to be transformed as input and creates a parse tree.

3. **Parse Tree Traversal:** A Listener class is implemented for the generated parse tree with overloaded events from the base listener of Parse Tree Walker. Traverse the parse tree using Parse Tree Walker with overloaded Listener class instance. This calls the related event at each node entry and exit of the parse tree.

4. **Transformations:** implement the event methods defined in the Listener class to perform the code transformations of RT-CUDA by modifying the related node payload or add/delete node payloads. In addition to the required transformations, an event method for variable type specifier needs to be implemented to add a space after every type specifier so the parse tree can be generated from the transformed code (see Appendix B and C).

5. **Code Generation:** at the end, final payload of the modified parse tree is used to produce the formatted code for better readability. This generates the final transformed CUDA code.

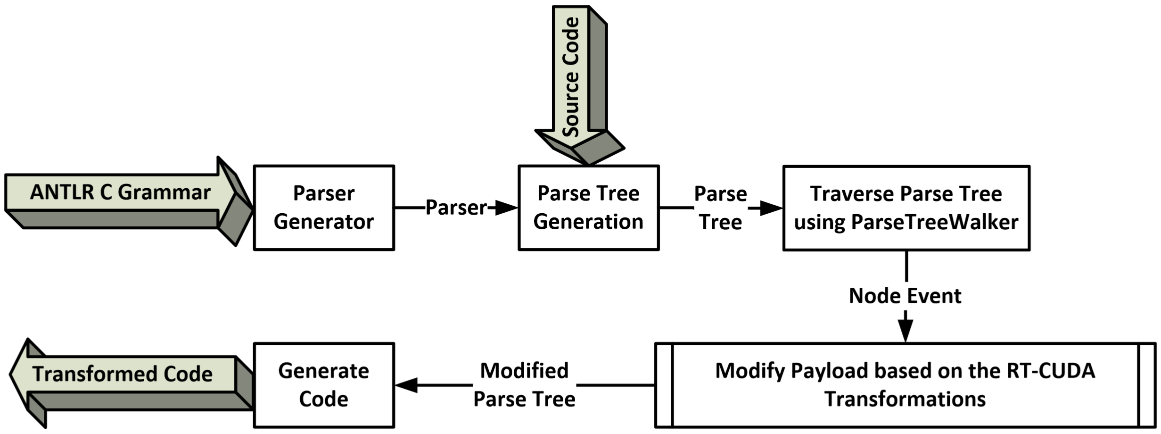


Figure 5.13: RT-CUDA Implementation Strategy Based on ANTLR Compiler Framework

5.5 RT-CUDA API Definitions

5.5.1 RTAPIInit()

Syntax

void RTAPIInit()

Description

This functions initialized the embedded library and creates a handle to an opaque structure, defined within the api, holding the embedded library context. It allocates hardware resources on the host and device and must be called prior to making any other API function calls. Because RTAPIInit allocates some internal resources and the release of those resources by calling RTAPIFinalize will implicitly call cudaDeviceSynchronize, it is recommended to minimize the number of RTAPIInit/RTAPIFinalize occurrences.

5.5.2 RTAPIFinalize()

Syntax

void RTAPIFinalize()

Description

This function releases hardware resources used by the embedded library. This function is usually the last call in the sequence of other API function calls. Because RTAPIInit allocates some internal resources and the release of those resources by calling RTAPIFinalize will implicitly call cudaDeviceSynchronize, it is recommended to minimize the number of RTAPIInit/RTAPIFinalize occurrences.

5.5.3 RTdSMM()

Syntax

void RTdSMM(float *C, const float *A, const float *B, int m, int n, int k)

Description

This function performs the single-precision matrix-matrix multiplication: $C = A \times B$, where A, B and C are dense matrices with dimensions m x k, k x n, and m x n respectively.

5.5.4 RTdDMM()

Syntax

void RTdDMM(double *C, const double *A, const double *B, int m, int n, int k)

Description

This function performs the double-precision matrix-matrix multiplication: $C = A \times B$, where A, B and C are dense matrices with dimensions m x k, k x n, and m x n respectively.

5.5.5 RTdSMV()

Syntax

void RTdSMV(float *C, const float *A, const float *B, int m, int n)

Description

This function performs the single-precision matrix-vector multiplication: $C = A \times B$, where A is a m x n dense matrix, B and C are dense vectors of length n and m respectively.

5.5.6 RTdDMV()

Syntax

void RTdDMV(double *C, const double *A, const double *B, int m, int n)

Description

This function performs the double-precision matrix-vector multiplication: $C = A \times B$, where A is a m x n dense matrix, B and C are dense vectors of length n and m respectively.

5.5.7 RTdSMT()

Syntax

void RTdSMT(float *C, const float *A, int m, int n)

Description

This function performs the single-precision out-of-place matrix transposition: $C = A^T$, where A and C are dense matrices with dimensions n x m and m x n respectively.

5.5.8 RTdDMT()

Syntax

void RTdDMT(double *C, const double *A, int m, int n)

Description

This function performs the double-precision out-of-place matrix transposition: $C = A^T$, where A and C are dense matrices with dimensions n x m and m x n respectively.

5.5.9 RTdSVV()

Syntax

void RTdSVV(float *C, const float *A, int m, const float *B, int n)

Description

This function performs the single-precision vector-vector multiplication: $C = A^T \times B$, where A and B are dense vectors with dimensions m and n respectively, C is a dense matrix with dimension m x n.

5.5.10 RTdDVV()

Syntax

void RTdDVV(double *C, const double *A, int m, const double *B, int n)

Description

This function performs the double-precision vector-vector multiplication: $C = A^T \times B$, where A and B are dense vectors with dimensions m and n respectively, C is a dense matrix with dimension m x n.

5.5.11 RTdSDOT()

Syntax

void RTdSDOT(const float *C, const float *A, int n, float *r)

Description

This function computes the single-precision dot product of vectors A and C. Hence, the result (r) is $\sum_{i=1}^{m} C[i] \times A[i]$, where A and C are dense vectors of length m.

5.5.12 RTdDDOT()

Syntax

void RTdDDOT(const double *C, const double *A, int n, double *r)

Description

This function computes the double-precision dot product of vectors A and C. Hence, the result (r) is $\sum_{i=1}^{m} C[i] \times A[i]$, where A and C are dense vectors of length m.

5.5.13 RTspSArrayCreate()

Syntax

void RTspSArrayCreate(float *A, RTspSArray *array, int m, int n)

Description

This function creates single-precision sparse matrix representation defined as RT-spSArray structure from a dense matrix A with dimension m x n. RTspSArray is a row-major single-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.5.14 RTspDArrayCreate()

Syntax

void RTspDArrayCreate(double *A, RTspDArray *array, int m, int n)

Description

This function creates double-precision sparse matrix representation defined as RTspDArray structure from a dense matrix A with dimension m x n. RTspDArray is a row-major double-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.5.15 RTspSArrayLoadFromFile()

Syntax

void RTspSArrayLoadFromFile(char *filename, RTspSArray *array)

Description

This function reads a market matrix file (filename) and store into a single-precision sparse matrix representation defined as RTspSArray structure. RTspSArray is a row-major single-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.5.16 RTspDArrayLoadFromFile()

Syntax

void RTspDArrayLoadFromFile(char *filename, RTspDArray *array)

Description

This function reads a market matrix file (filename) and store into a double-precision sparse matrix representation defined as RTspDArray structure. RTspDArray is a row-major double-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.5.17 RTspSMM()

Syntax

void RTspSMM(RTspSArray *C, const RTspSArray *A, const RTspSArray *B, int m, int n, int k, int format=RTCSR)

Description

This function performs the single-precision matrix-matrix multiplication: $C = A \times B$, where A, B and C are sparse matrices (defined as RTspSArray structure) with dimensions m x k, k x n, and m x n respectively. *format* identifies the sparse matrix format (default=csr, only csr format is supported in this version) to be used by the function for computations. RTspSArray is a row-major single-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.5.18 RTspDMM()

Syntax

void RTspDMM(RTspDArray *C, const RTspDArray *A, const RTspDArray *B, int m, int n, int k, int format=RTCSR)

Description

This function performs the double-precision matrix-matrix multiplication: $C = A \times B$, where A, B and C are sparse matrices (defined as RTspDArray structure) with dimensions m x k, k x n, and m x n respectively. *format* identifies the sparse matrix format (default=csr, only csr format is supported in this version) to be used by the function for computations. RTspDArray is a row-major double-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.5.19 RTspdSMM()

Syntax

void RTspdSMM(float *C, const RTspSArray *A, const float *B, int m, int n, int k, int format=RTCSR)

Description

This function performs the single-precision matrix-matrix multiplication: $C = A \times B$, where A is a sparse matrix (defined as RTspSArray structure) with dimension m x k and B, C are dense matrices with dimensions k x n, and m x n respectively. *format* identifies the sparse matrix format (default=csr, csr and bsr formats are supported in this version) to be used by the function for computations. RTspSArray is a row-major single-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.5.20 RTspdDMM()

Syntax

void RTspdDMM(double *C, const RTspDArray *A, const double *B, int m, int n, int k, int format=RTCSR)

Description

This function performs the double-precision matrix-matrix multiplication: $C = A \times B$, where A is a sparse matrix (defined as RTspSArray structure) with dimension m x k and B, C are dense matrices with dimensions k x n, and m x n respectively. *format* identifies the sparse matrix format (default=csr, csr and bsr formats is supported in this version) to be used by the function for computations. RTspDArray is a row-major double-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.5.21 RTspSMV()

Syntax

void RTspSMV(float *C, const RTspSArray *A, const float *B, int m, int n, int format=RTHYB)

Description

This function performs the single-precision matrix-vector multiplication: $C = A \times B$, where A is a m x n sparse matrix (defined as RTspSArray structure), B and C are dense vectors of length n and m respectively. *format* identifies the sparse matrix format (default=hyb, csr, bsr, and hyb formats are supported in this version) to be used by the function for computations. RTspSArray is a row-major single-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.5.22 RTspDMV()

Syntax

void RTspDMV(float *C, const RTspDArray *A, const float *B, int m, int n, int format=RTHYB)

Description

This function performs the double-precision matrix-vector multiplication: $C = A \times B$, where A is a m x n sparse matrix (defined as RTspDArray structure), B and C are dense vectors of length n and m respectively. *format* identifies the sparse matrix format (default=hyb, csr, bsr, and hyb formats are supported in this version) to be used by the function for computations. RTspDArray is a row-major double-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.5.23 RTspSArrayDestroy()

Syntax

void RTspSArrayDestroy(RTspSArray *array)

Description

This function de-allocates single-precision sparse matrix representation defined as RTspSArray structure. RTspSArray is a row-major single-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.5.24 RTspDArrayDestroy()

Syntax

void RTspDArrayDestroy(RTspDArray *array)

Description

This function de-allocates double-precision sparse matrix representation defined as RTspDArray structure. RTspDArray is a row-major double-precision linear addressing array structure to store sparse matrix containing three arrays for row indices, column indices, and values of the matrix. It also includes the count of non-zero elements in the matrix.

5.6 RT-CUDA Package README

5.6.1 RT-CUDA Installation and Setup

Pre-Requisites

- Java JDK/JRE 1.7 or later, it can be downloaded from the following link: (www.oracle.com/technetwork/java/javase/downloads/index.html)
- NVIDIA CUDA Toolkit, it can be downloaded from the following link: (https://developer.nvidia.com/cuda-toolkit)

Package Extraction

To extract RT-CUDA package, run the following command:

unzip RTCUDACompiler.zip

5.6.2 RT-CUDA Usage

Input

RT-CUDA compiler requires following files to be created in src folder:

- **kernel.c:** This file contains the C function that needs to be converted as CUDA kernel to run on GPU device. The function should follow the ANSI C standard with required parameters and no return type. This function implements the C-Loop structure to be partitioned among multiple CUDA blocks and threads to compute the required results. Following is the syntax of the function definition:

  void func_name(type param, ...){ $< functionbody >$ }

- **main.c:** This file contains the C main function that implements the user input, data allocation, initialization, function call (defined in kernel.c), and output of the program following the ANSI C standard. All the arrays that are going to be used by the kernel function should be allocated dynamically using C malloc() function. To run the kernel on a particular GPU device, user should use the function cudaSetDevice(GPU_ID) before calling the kernel function where GPU_ID is the id of the GPU device available in the system.

- **config.txt:** This file contains the configurations for different parameters used by the compiler for optimizations and final code generation. Following are the list of parameters that need to be defined in configuration:

– **LOOP_COLLAPSING:** Enabled(1)/Disabled(0) loop collapsing optimization. It is only applicable to the kernel with 2D resultant matrix and having two nested loops in the computations.

– **BLOCK_SKEW:** Enabled(1)/Disabled(0) block skewing optimization. It is only applicable to the kernel with 2D resultant matrix. It increases the thread access locality by merging multiple resultant blocks horizontally to one thread block.

– **PREFETCHING:** Enabled(1)/Disabled(0) prefetching optimization. It is only applicable to the kernel having 2D matrix in the computation that need to be tiled to store in shared memory.

– **PREFETCHED_ARRAYS:** List of array variables that need to be tiled. This is only applied if PREFETCHING is enabled.

– **NON_PREFETCHED_ARRAYS:** List of array variables that should be ignored for tiling. This is only applied if PREFTECHING is enabled.

– **DATA_TYPE:** It defines the data type of the arrays in the computation and resultant that need to be stored in GPU memory.

– **KERNEL_NAMES:** List of function names that need to be converted as CUDA kernels that are defined in kernel.c file.

- **2DMATRIX:** It is set to 1 for 2D resultant matrix and 0 for 1D.
- **ROW_DIM:** This is the leading dimension of the matrices used in computation.
- **MAX_BLOCKSIZE:** Upper bound of BLOCKSIZE to be analyzed by RT-CUDA Parameter Tunner. This should be less than or equal to the maxmimum possible thread block size of the underlying GPU compute capability. To check all possible block size based on the underlying GPU architecture automatically, set this value to 0.
- **MAX_MERGE_LEVEL:** Upper bound of MERGE_LEVEL to be analyzed by RT-CUDA Parameter Tunner. This should be less than or equal to the MAX_BLOCKSIZE.
- **MAX_SKEW_LEVEL:** Upper bound of SKEW_LEVEL to be analyzed by RT-CUDA Parameter Tunner. This should be less than or equal to the MAX_BLOCKSIZE.
- **MIN_BLOCKSIZE:** Lower bound of BLOCKSIZE to be analyzed by RT-CUDA Parameter Tunner. This should be greater than or equal to 1 and less than or equal to the MAX_BLOCKSIZE.
- **MIN_MERGE_LEVEL:** Lower bound of MERGE_LEVEL to be analyzed by RT-CUDA Parameter Tunner. This should be greater than or equal to 1 and less than or equal to the MAX_MERGE_LEVEL.
- **MIN_SKEW_LEVEL:** Lower bound of SKEW_LEVEL to be analyzed by RT-CUDA Parameter Tunner. This should be greater than

or equal to 1 and less than or equal to the MAX_SKEW_LEVEL.

Following is an example of a configuration file:

```
LOOP_COLLAPSING=1

BLOCK_SKEW=1

PREFETCHING=0

PREFETCHED_ARRAYS=A

NON_PREFETCHED_ARRAYS=B

DATA_TYPE=float

KERNEL_NAMES=matrix_scale

2DMATRIX=1

ROW_DIM=N

MAX_BLOCKSIZE=0

MAX_MERGE_LEVEL=8

MAX_SKEW_LEVEL=2

MIN_BLOCKSIZE=32

MIN_MERGE_LEVEL=1

MIN_SKEW_LEVEL=1
```

Execution

To run the compiler from the command line, go to the dist folder and type the following:

```
java -jar RTCUDATranslator.jar
```

Output

RT-CUDA generates following files in the output folder:

- **kernel.cu:** This file contains the converted CUDA kernels.
- **main.cu:** This file contains the main program that calls CUDA kernels.
- **Header Files:** The compiler generates three header files params.h, and rcuda.h that are included in the main.cu.
- **Makefile:** For compilation with make program, it generates Makefile and the dependent findcudalib.mk files.

5.7 RT-CUDA Examples

We have applied RT-CUDA on the following applications that served for testing RT-CUDA over linear algebra operators:

1. Matrix-Matrix Multiplication: multiplying two matrices
2. Dense Matrix Operators using RT-CUDA API: perform Matrix-Matrix Multiplication, Matrix-Vector Multiplication, Matrix Transpose, and Inner Product (Vector-Vector Multiplication) by calling RT-CUDA API functions to invoke related CUBLAS library routines

3. Sparse-Matrix Operators using RT-CUDA API: perform Matrix-Matrix Multiplication, Matrix-Vector Multiplication, Matrix Transpose, and Inner Product (Vector-Vector Multiplication) by calling RT-CUDA API functions to invoke related cuSPARSE library routines

4. Matrix Scaling: multiplying a matrix with a scalar

5. Matrix Addition: addition of two matrices

6. Demosaic: a digital image process used to reconstruct a full color image from the incomplete color samples. This is also called CFA (Color Filter Array) interpolation or color reconstruction

7. Histogram: calculate an estimate of the probability distribution of a continuous variable

8. Matrix-Vector Multiplication: multiplying a matrix by a vector

9. Vector-Vector Multiplication: inner product of two vectors

10. AXPY: addition of a vector with an another scaled vector

Following sections show the applications 1, 2, and 3. See Appendix E for the implementation of applications 4, 5, 6, 7, 8, 9, and 10.

5.7.1 Matrix-Matrix Multiplication

Inputs

Listing 5.5: kernel.c

```
void matrix_mul(float *C, float * restrict A, float *
   restrict B, int N)
{
    float sum=0.0;
    for(int i=0; i < N; i++)
        for(int j=0; j < N; j++){
                for(int k=0; k < N; k++){
                float b = B[k][j];
                        sum += A[i][k] * b;
            }
                C[i][j] = sum;
        }
}
```

Listing 5.6: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *A, *B, *C;

    int memsize = N * N * sizeof(float);
    A = (float *)malloc(memsize);
    B = (float *)malloc(memsize);
    C = (float *)malloc(memsize);

    matrix_mul(C, A, B, N);

    free(A);
    free(B);
    free(C);

    exit(0);
}
```

Listing 5.7: config.txt

```
LOOP_COLLAPSING=1
BLOCK_SKEW=0
PREFETCHING=1
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B
DATA_TYPE=float
KERNEL_NAMES=matrix_mul
2DMATRIX=1
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=0
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Outputs

Listing 5.8: kernel.cu

```
__global__ void matrix_mul(float *C, float * __restrict__
    A, float * __restrict__ B, int N)
{
        float sum[MERGE_LEVEL];

        for(int i=0; i < MERGE_LEVEL; i++)
                sum[i] = 0.0;

        int tid = threadIdx.x;
        int bid = blockIdx.x;
        int ij = bid * BLOCKSIZE + tid;
        int i = (ij / N) * MERGE_LEVEL;
        int j = ij % N;

        __shared__ float As[MERGE_LEVEL][BLOCKSIZE];

        int k=0;
        for(int m=0; m < MERGE_LEVEL; m++)
                As[m][tid] = A[(i+m) * N + k + tid];
        __syncthreads();
        for(k = 0; k < N-BLOCKSIZE; k+=BLOCKSIZE){
                for(int t=0; t < BLOCKSIZE; t++){
                        float b = B[(k+t) * N + j];
                        for(int m = 0; m < MERGE_LEVEL; m
                           ++)
                                sum[m] += As[m][t] * b;
                }
                __syncthreads();
                for(int m = 0; m < MERGE_LEVEL; m++)
                        As[m][tid] = A[(i+m) * N + (k+
                           BLOCKSIZE) + tid];
                __syncthreads();
        }
        for(int t=0; t < BLOCKSIZE; t++){
                float b = B[(k+t) * N + j];
                for(int m = 0; m < MERGE_LEVEL; m++)
                        sum[m] += As[m][t] * b;
        }
        for(int m = 0; m < MERGE_LEVEL; m++)
                C[(i + m) * N + j] = sum[m];
}
```

Listing 5.9: main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc,char *argv[]){
        int N=1024;
        int GPU=0;
        if(argc>1)N=atoi(argv[1]);
        if(argc>2)GPU=atoi(argv[2]);

        cudaSetDevice (GPU);

        float *A,*B,*C;
        int memsize=N*N*sizeof(float );
        cudaMallocManaged(&A,memsize);
        cudaMallocManaged(&B,memsize);
        cudaMallocManaged(&C,memsize);

        dim3 threads(BLOCKSIZE,1);
        dim3 grid(N*N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1)
           ;
        matrix_mul<<<grid,threads>>>(C,A,B,N);
        cudaDeviceSynchronize();

        cudaFree (A);
        cudaFree (B);
        cudaFree (C);
        cudaThreadExit();
}
```

Listing 5.10: params.h

```
#define BLOCKSIZE 64
#define MERGE_LEVEL 16
#define SKEW_LEVEL 1
```

5.7.2 Dense Matrix Operators using RT-CUDA API

Inputs

Listing 5.11: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
```

```
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *C, *A, *B, *X, *Y;

    int memsize = N * N * sizeof(float);
    int memsizevec = N * sizeof(float);
    C = (float *)malloc(memsize);
    A = (float *)malloc(memsize);
    B = (float *)malloc(memsize);
    X = (float *)malloc(memsizevec);
    Y = (float *)malloc(memsizevec);

    RTAPIInit();
    RTdSMM(C, A, B, N, N, N);
    RTdSMV(Y, A, X, N, N);
    RTdSMT(C, A, N, N);
    RTdSVV(C, X, Y, N);
    float result;
    RTdSDOT(X, Y, N, &result);
    RTAPIFinalize();

    free(C);
    free(A);
    free(B);
    free(X);
    free(Y);

    exit(0);
}
```

Outputs

Listing 5.12: main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *C, *A, *B, *X, *Y;

    int memsize = N * N * sizeof(float);
    int memsizevec = N * sizeof(float);
    cudaMallocManaged(&C,memsize);
    cudaMallocManaged(&A,memsize);
    cudaMallocManaged(&B,memsize);
    cudaMallocManaged(&X,memsizevec);
    cudaMallocManaged(&Y,memsizevec);

    RTAPIInit();
    RTdSMM(C, A, B, N, N, N);
    RTdSMV(Y, A, X, N, N);
    RTdSMT(C, A, N, N);
    RTdSVV(C, X, Y, N);
    float result;
    RTdSDOT(X, Y, N, &result);
    RTAPIFinalize();

    cudaFree(C);
    cudaFree(A);
    cudaFree(B);
    cudaFree(X);
    cudaFree(Y);
    cudaThreadExit();
}
```

5.7.3 Sparse Matrix Operators using RT-CUDA API

Inputs

Listing 5.13: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *DC, *X, *Y;

    int memsize = N * N * sizeof(float);
    int memsizevec = N * sizeof(float);
    X = (float *)malloc(memsizevec);
    Y = (float *)malloc(memsizevec);
    DC = (float *)malloc(memsize);

    RTAPIInit();
    RTspSArray *A, *B, *C;
    RTspSArrayLoadFromFile(argv[1], A);
    RTspSArrayLoadFromFile(argv[1], B);
    RTspSArrayCreate(DC, C, N, N);

    RTspSMM(C, A, B, N, N, N);
    RTspdSMM(C, A, B, N, N, N);
    RTspdSMM(C, A, B, N, N, N, RTBSR);
    RTspSMV(Y, A, X, N, N);
    RTspSMV(Y, A, X, N, N, RTBSR);
    RTspSMV(Y, A, X, N, N, RTCSR);
    RTspSArrayDestroy(A);
    RTspSArrayDestroy(B);
    RTspSArrayDestroy(C);
    RTAPIFinalize();

    free(X);
    free(Y);
    free(DC);

    exit(0);
}
```

Outputs

Listing 5.14: main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *DC, *X, *Y;
    int memsize = N * N * sizeof(float);
    int memsizevec = N * sizeof(float);
    cudaMallocManaged(&X,memsizevec);
    cudaMallocManaged(&Y,memsizevec);
    cudaMallocManaged(&DC,memsize);

    RTAPIInit();
    RTspSArray *A, *B, *C;
    RTspSArrayLoadFromFile(argv[1], A);
    RTspSArrayLoadFromFile(argv[1], B);
    RTspSArrayCreate(DC, C, N, N);

    RTspSMM(C, A, B, N, N, N);
    RTspdSMM(C, A, B, N, N, N);
    RTspdSMM(C, A, B, N, N, N, RTBSR);
    RTspSMV(Y, A, X, N, N);
    RTspSMV(Y, A, X, N, N, RTBSR);
    RTspSMV(Y, A, X, N, N, RTCSR);
    RTspSArrayDestroy(A);
    RTspSArrayDestroy(B);
    RTspSArrayDestroy(C);
    RTAPIFinalize();

    cudaFree(X);
    cudaFree(Y);
    cudaFree(DC);

    cudaThreadExit();
}
```

CHAPTER 6

PERFORMANCE EVALUATION OF THE RESTRUCTURING TOOL

We have run our experiments on Tesla K20c GPU (see Table-3.1 for specifications) with various applications including Demosaic, Histogram, Matrix Addition (Madd), Matrix Multiplication (MM), Matrix Vector Multiplication (MV), and Vector Vector Multiplication (VV). We have compared the implementations using RTA-CUDA with CUBLAS, GPGPU compiler, and OpenACC (PGI compiler) implementations. We have also evaluated the different inter-block synchronization primitives provided in the tool to be used in Jacobi Iterative Solver. Furthermore, the effects of calling external library functions for basic linear algebra operations and sparse matrices have also been evaluated. The correctness of the results of each converted application using RT-CUDA are guaranteed by comparing with the results of serial version of each application on CPU using the subset of the problem sizes. The result showed that our conversions produce correct resultant values. We have also trace the resultant matrix indices with the mapping of block and thread ids which are also found to be corrected.

6.1 Evaluation of Basic Linear Algebra Operations

This section shows the evaluation of the tool using a set of operators in LAPACK benchmark suite for basic linear algebra operations including Madd, MM, MV, and VV applications to compare with CUBLAS, GPGPU compiler, and OpenACC (PGI compiler). All four kernels are generated by the tool using RTA-CUDA algorithm as explained in section 5.2 and applied different set of transformations/optimizations according to the specific code structure of each application. Table 6.1 shows the applied transformations/optimizations for each application, the table also shows the optimal parameters (<BLOCKSIZE,MERGE_LEVEL,SKEW_LEVEL>) for each application obtained through parameter tuning algorithm (Algorithm 1). The comparisons for different applications and tools have been shown with appropriate space size (N) for each application to show the execution times in a particular range.

Fig. 6.1 shows the execution time in seconds of different applications using CUBLAS and RTA-CUDA. The comparisons have been performed using following

Transformations/Optimizations	Madd	MM	MV	VV
Loop Collapsing	√	√		√
Array Transformations	√	√	√	√
Loop Partitioning	√	√	√	√
Block Merging		√		√
Block Skewing	√			
Prefetching using Shared Memory		√	√	
Remove Redundant Array Access		√		√
Read-Only Data Cache	√	√	√	√
Optimal Parameters	$< 128, 1, 2 >$	$< 64, 16, 1 >$	$< 64, 1, 1 >$	$< 256, 32, 1 >$

Table 6.1: Code Transformation Summary for Each Application

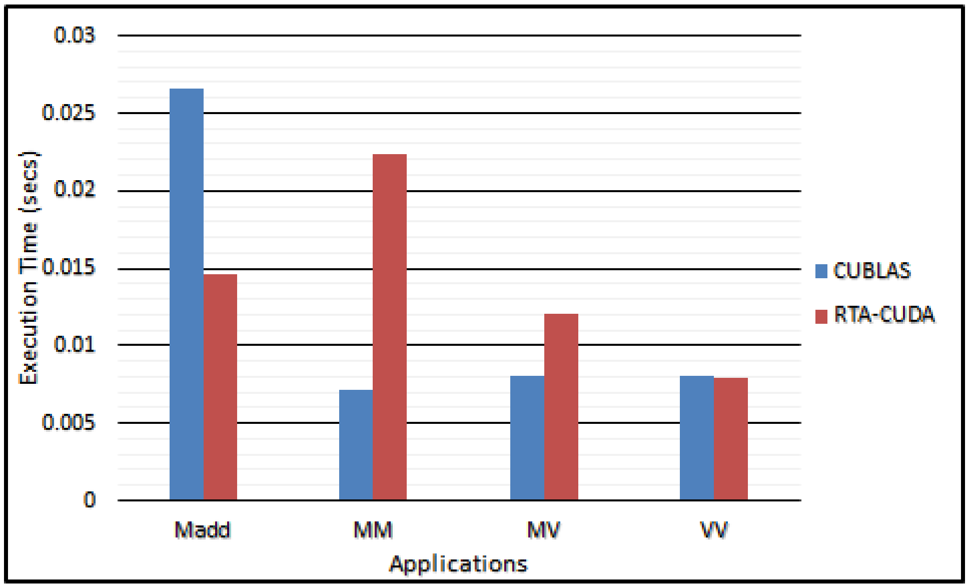


Figure 6.1: Comparing CUBLAS and RTA-CUDA with different applications

CUBLAS functions:

- cublasSgeam for Madd
- cublasSgemm for MM and VV
- cublasSgemv for MV

The results show that RTA-CUDA obtained better performance for Madd and VV. But, for complex applications such as MM and MV, CUBLAS still has significant performance advantage over RTA-CUDA. This is because cublasSgemm and cublasSgemv functions have been developed with complex kernel optimizations at very low level of coding by hand while at this stage, we are focusing on high level CUDA kernel optimizations. However, with the proposed high level kernel optimizations, RTA-CUDA outperforms CUBLAS with 45% improvement in case of Madd and 2% improvement in case of VV.

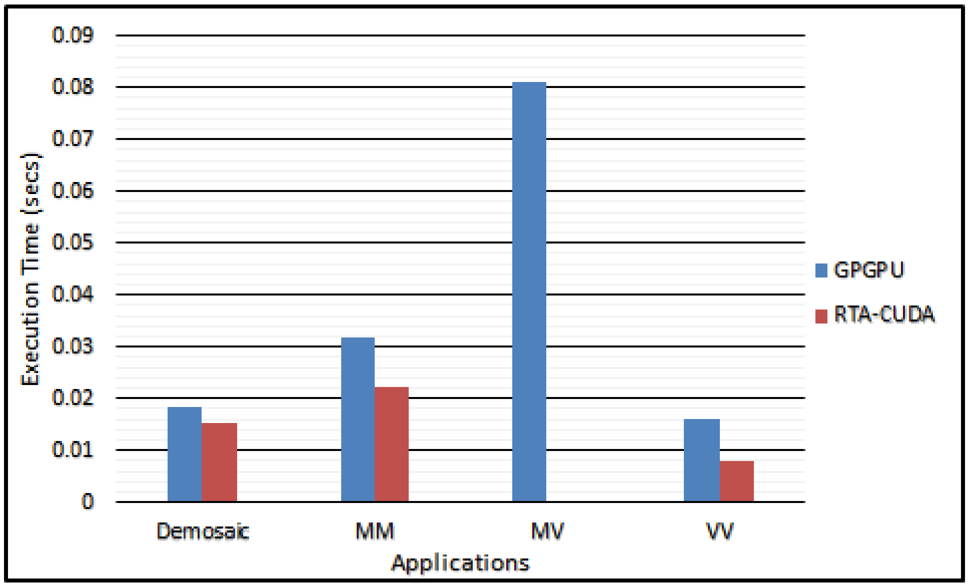


Figure 6.2: Comparing GPGPU Compiler and RTA-CUDA with different applications

Fig. 6.2 shows the execution time in seconds of different applications using GPGPU compiler and RTA-CUDA. The results show that RTA-CUDA outperforms GPGPU compiler with 17% improvement in case of Demosaic, 30% improvement in case of MM, 99% improvement in case of MV and 50% improvement in case of VV. Also, MV implementation in GPGPU compiler gives value errors in case of large space size while RTA-CUDA generates correct values with any space size. So, the optimizations proposed in GPGPU compiler is not likely to be useful in kepler family of GPUs where RTA-CUDA's optimizations obtained better performance.

Fig. 6.3 shows the execution time in seconds of different applications using

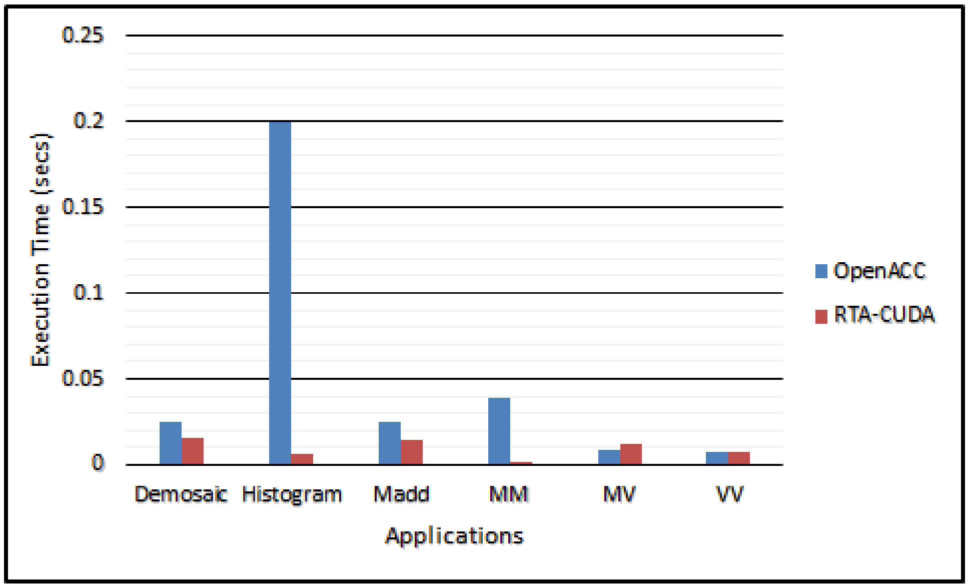
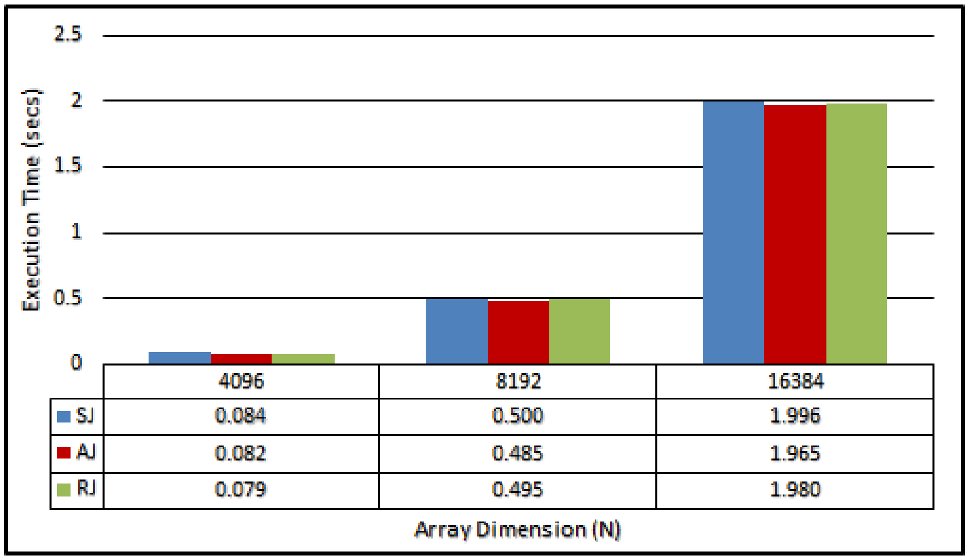


Figure 6.3: Comparing OpenACC (PGI Compiler) and RTA-CUDA with different applications

OpenACC implementation in PGI compiler and RTA-CUDA. The results show that RTA-CUDA outperforms OpenACC implementation of PGI compiler with 37% improvement in case of Demosaic, 97% improvement in case of Histogram, 42% improvement in case of Madd, and 99% improvement in case of MM and approx similar performance in case of VV and MV.

6.2 Evaluation of Inter-Block Synchronization Primitives

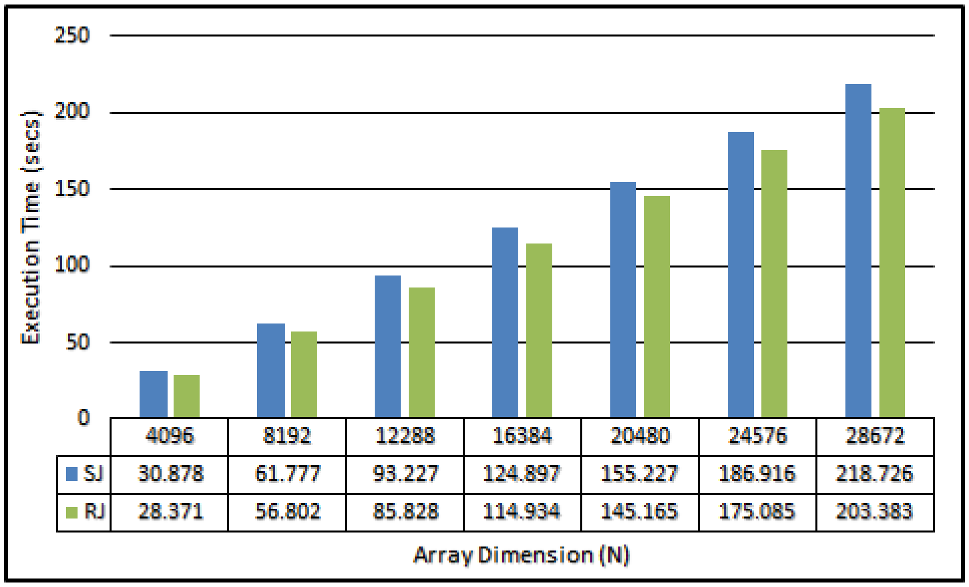
This section shows the evaluation of inter-block synchronization primitives provided in the tool. The execution time of three variants of block Jacobi iterative solver have been calculated and compared: 1) Synchronous Jacobi (SJ), 2) Asynchronous Jacobi (AJ), and 3) Relaxed Jacobi (RJ).



	4096	8192	16384
🟦 SJ	0.084	0.500	1.996
🟥 AJ	0.082	0.485	1.965
🟩 RJ	0.079	0.495	1.980

Figure 6.4: Performance comparison of Single-Precision SJ, AJ, and RJ with different array dimensions

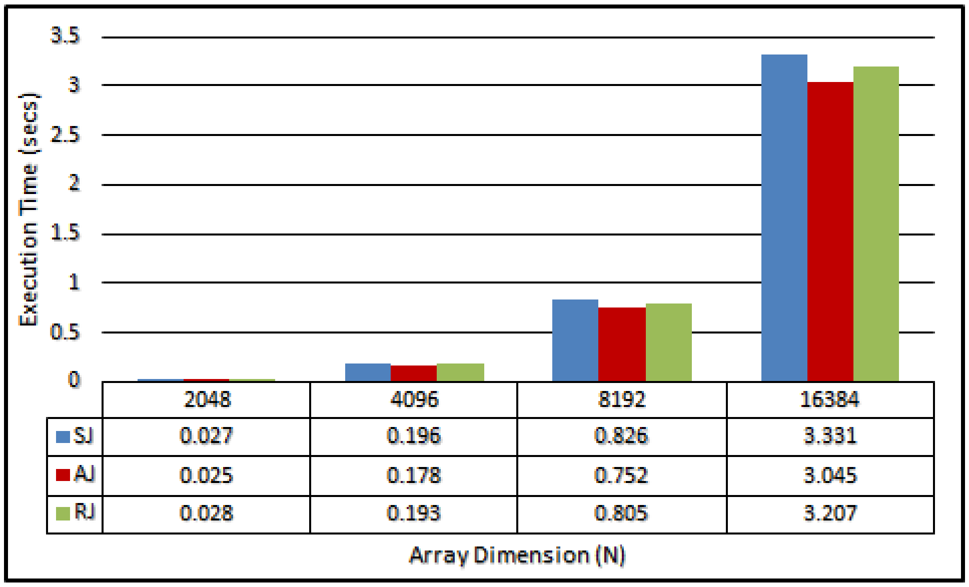
Fig. 6.4 shows the execution time in seconds of SJ, AJ, and RJ implementations with different array dimensions using 128 (maximum concurrent threads blocks possible for these implementations) number of blocks and single-precision floating point operations. Here, SJ implementation shows a little overhead of synchronization among each iteration and reduces with the increase in the array dimension that for N=16384 with 128 thread blocks the synchronization overhead is just about 1.5% over AJ implementation. RJ implementation further obtained little improvement over SJ implementation that is about 1% reduction in execution time than SJ implementation except the case of N=4096 where RJ improvement is about 6%. This shows that all thread blocks complete its execution with little difference in time as the tasks among each thread block is distributed equally. Relaxed synchronization approach is expected to give more performance improvement if the tasks are not evenly distributed among thread blocks on GPU architectures. To analyze the behaviour of RJ in case of unbalanced thread blocks execution, we have performed an experiment of a naïve block-row partitioning in sparse matrix-vector operation used in an iterative solver (such as Sparse Jacobi with MV), which causes some load unbalancing over the iteration space. Fig. 6.5 shows the execution time in seconds of SJ and RJ with unbalanced thread blocks. Here, RJ obtained about 8% performance improvement in terms of execution time over SJ implementation.



	4096	8192	12288	16384	20480	24576	28672
🟦 SJ	30.878	61.777	93.227	124.897	155.227	186.916	218.726
🟩 RJ	28.371	56.802	85.828	114.934	145.165	175.085	203.383

Figure 6.5: Performance comparison of Sparse Jacobi with MV

Fig. 6.6 shows the execution time in seconds of SJ, AJ, and RJ implementations with different array dimensions using 64 number of blocks (optimal number of thread blocks for these implementations) and double-precision floating point operations. Here, SJ implementation shows a high overhead of synchronization among each iteration that is for N=16384 the synchronization overhead is about 12% over AJ implementation. RJ implementation is shown performance improvement of about 4% over SJ implementation. RJ also shows increasing trend in terms of performance improvement over SJ with the increase in the array dimension.



	2048	4096	8192	16384
🟦 SJ	0.027	0.196	0.826	3.331
🟥 AJ	0.025	0.178	0.752	3.045
🟩 RJ	0.028	0.193	0.805	3.207

Figure 6.6: Performance comparison of Double Precision SJ, AJ, and RJ with different array dimensions

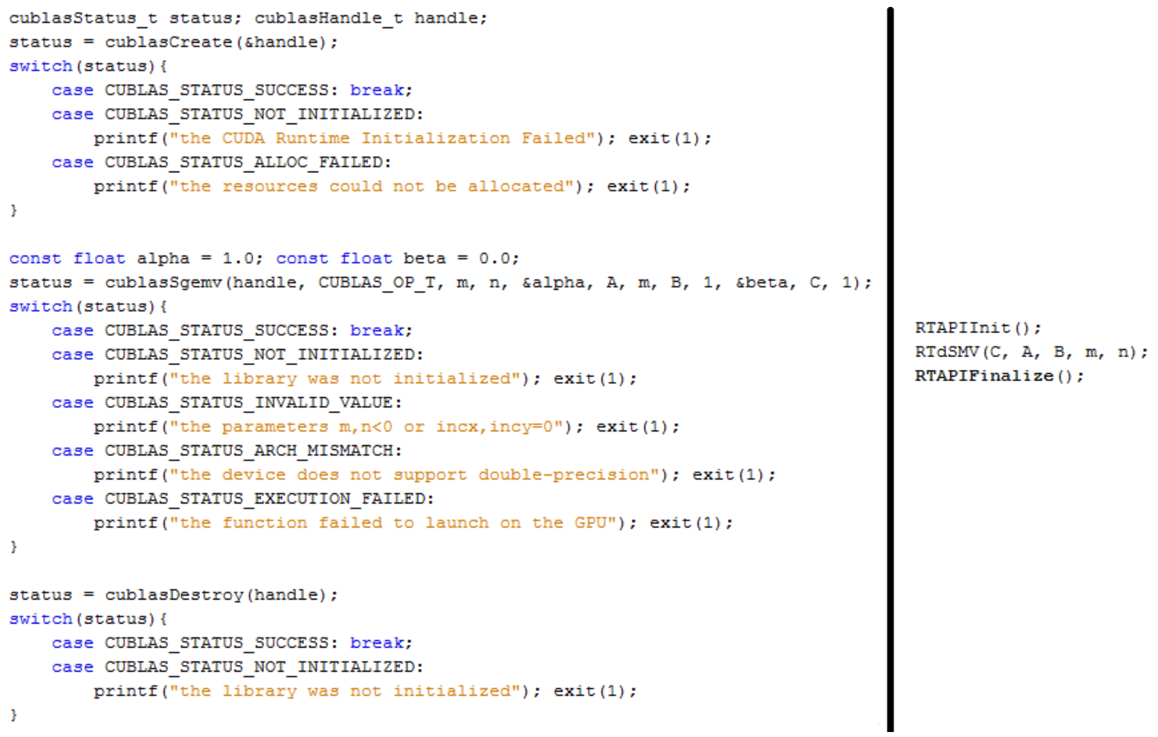
6.3 Effects of Calling External CUBLAS Functions

As shown in section 6.1, in most of the cases CUBLAS library functions obtained the highest performance in comparison to RT-CUDA and other tools. This is because CUBLAS functions have been developed with complex kernel optimizations at very low level of coding by hand. To get the benefit of the existing optimized CUDA libraries, we have provided a feature of calling external library functions as an optimization within the tool.

This section shows the performance evaluation of RT-CUDA using CUBLAS functons as an optimization instead of using RTA-CUDA algorithm. We applied the tool on Matrix Multiplication (MM), Matrix Vector Multiplication (MV), Matrix Transpose (MT), and Vector Vector Multiplication (VV). The execution time of each application has been compared with GPGPU compiler and PGI OpenACC compiler implementations.

RT-CUDA enables transparent invocation of the most optimized external math libraries like cuBLAS and cuSparse libraries. It provides interfacing APIs, error handling interpretation, and user transparent programming. Fig. 6.7 shows the coding comparison of MV by directly using cuBLAS library and simplified invocation of the library using RT-CUDA APIs. It reduces the programming efforts of about 93% in terms of lines of code and hides complex parameters selection by the programmers while giving similar performance in terms of execution time. Fig. 6.8 shows normalized execution time of MM and MV operations using CUBLAS library and RT-CUDA API.

Table 6.2 shows the execution time in milliseconds of RT-CUDA, GPGPU compiler and PGI OpenACC implementations of different applications with different array dimensions. The results show that RT-CUDA significantly outperforms both GPGPU and PGI OpenACC implementations. It obtained performance improvements in terms of execution time of up to 80%, 100%, 4%, and 56% for MM, MV, MT, and VV respectively over GPGPU compiler with N=4096. It obtained performance improvements of up to 100%, 33%, 78%, and 70% for MM, MV, MT,



```
cublasStatus_t status; cublasHandle_t handle;
status = cublasCreate(&handle);
switch(status){
    case CUBLAS_STATUS_SUCCESS: break;
    case CUBLAS_STATUS_NOT_INITIALIZED:
        printf("the CUDA Runtime Initialization Failed"); exit(1);
    case CUBLAS_STATUS_ALLOC_FAILED:
        printf("the resources could not be allocated"); exit(1);
}

const float alpha = 1.0; const float beta = 0.0;
status = cublasSgemv(handle, CUBLAS_OP_T, m, n, &alpha, A, m, B, 1, &beta, C, 1);
switch(status){
    case CUBLAS_STATUS_SUCCESS: break;
    case CUBLAS_STATUS_NOT_INITIALIZED:
        printf("the library was not initialized"); exit(1);
    case CUBLAS_STATUS_INVALID_VALUE:
        printf("the parameters m,n<0 or incx,incy=0"); exit(1);
    case CUBLAS_STATUS_ARCH_MISMATCH:
        printf("the device does not support double-precision"); exit(1);
    case CUBLAS_STATUS_EXECUTION_FAILED:
        printf("the function failed to launch on the GPU"); exit(1);
}

status = cublasDestroy(handle);
switch(status){
    case CUBLAS_STATUS_SUCCESS: break;
    case CUBLAS_STATUS_NOT_INITIALIZED:
        printf("the library was not initialized"); exit(1);
}
```

```
RTAPIInit();
RTdSMV(C, A, B, m, n);
RTAPIFinalize();
```

Figure 6.7: Code snipped of MV using cuBLAS library (left) and RT-CUDA API (right)

and VV respectively over PGI OpenACC implementations with N=4096.

	Matrix Multiplication			Matrix Vector Multiplication			Matrix Transpose			Vector Vector Multiplication		
N	**RT-CUDA**	**GPGPU**	**OpenACC**	**RT-CUDA**	**GPGPU**	**OpenACC**	**RT-CUDA**	**GPGPU**	**OpenACC**	**RT-CUDA**	**GPGPU**	**OpenACC**
256	0.05	0.14	6.19	0.01	0.99	0.05	0.01	0.01	0.05	0.01	0.01	0.12
512	0.20	0.70	38.59	0.01	7.72	0.06	0.02	0.02	0.10	0.01	0.01	0.20
1024	1.10	3.80	542.95	0.04	80.93	0.10	0.07	0.06	0.30	0.03	0.07	0.37
2048	7.12	31.96	5361.62	0.14	659.42	0.21	0.25	0.25	1.11	0.11	0.26	0.71
4096	52.32	258.83	41468.49	0.51	5760.39	0.76	0.97	1.01	4.36	0.44	0.99	1.45

Table 6.2: Comparing RT-CUDA with GPGPU compiler and PGI OpenACC compiler using different applications

6.4 Effects of Sparse Matrix Operations using CUDA Sparse Library Routines

We have evaluated various sparse matrix formats available in cuSparse library for Sparse-Sparse Matrix Multiplication (spMM), Sparse-Dense Matrix Multipli-

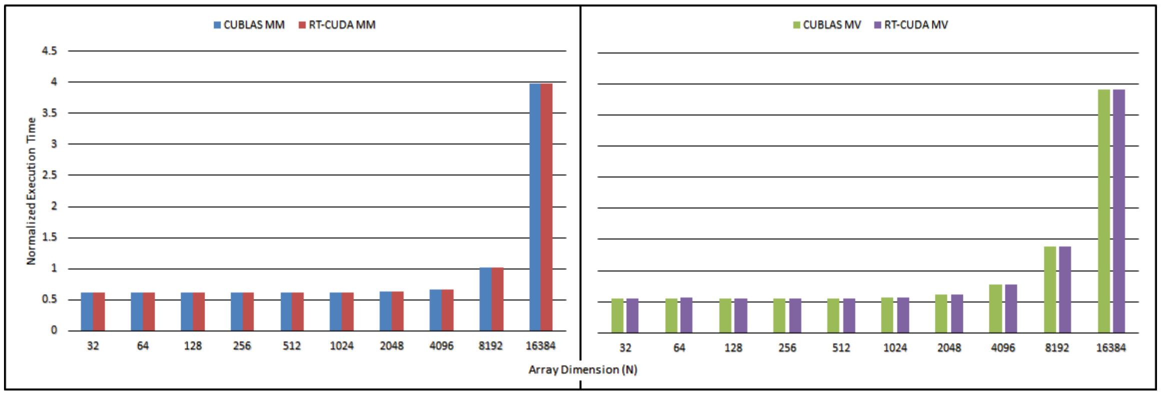


Figure 6.8: Normalized Execution Time of MM (left) and MV (right) using cuBLAS library and RT-CUDA API

cation (spdMM), and Sparse-Dense Matrix Vector Multiplication (spdMV) (see Example 5.7.3 for evaluation of different storage formats using RT-CUDA API). The objective of these evaluations is to access the performance of various matrix operators and storage schemes available in cuSparse library to select the best storages as standard in RT-CUDA. The evaluation results show that the sparse matrix multiplication (both spMM and spdMM) is only profitable in terms of memory allocations but not for execution time for computations as the dense matrix multiplication in CUBLAS is highly optimized and provide the best performance independent on the sparsity (percentage of number of zeros in the matrix) of the matrix. Whereas, in case of matrix-vector multiplication (spdMV), the sparse implementations obtain better performance both in terms of memory allocations and execution time in comparison to dense implementation.

Table 6.3 shows the memory allocations in MB for the sparse matrix in Dense, CSR, BSR (with 256 x 256 block dimensions), and HYB formats. Here, CSR and HYB formats show minimum memory requirements for matrix storage that is

N	Dense	CSR	BSR	HYB
1024	4	1	4	2
2048	16	4	16	6
4096	64	14	64	17
8192	256	51	257	54

Table 6.3: Matrix Storage Requirements in Different Formats with 90% Sparsity

about 50-80% less than the dense and BSR formats. Table 6.4 shows the execution time in seconds for the computations of spMM, spdMM, and spdMV in Dense, CSR, BSR (with 256 x 256 block dimensions), and HYB formats. For matrix multiplication, sparse operations show significant overhead of computations due to irregular memory access patterns of the randomly initialized sparse matrices. For matrix-vector multiplication, sparse operations obtained the speedup of about 4 and 2.5 over dense operations for N=8192 in CSR and HYB formats respectively. Furthermore, spdMV in CSR format is more efficient in terms of performance for $N <= 13312$ and spdMV in HYB format obtained more speedup for $N >= 14336$ as shown in Fig. 6.9.

	Dense		Sparse					
			CSR			BSR		HYB
N	MM	MV	spMM	spdMM	spdMV	spdMM	spdMV	spdMV
1024	0.00117	0.00019	0.00865	0.00258	0.00013	0.01508	0.00156	0.00010
2048	0.00729	0.00026	0.04826	0.01831	0.00017	0.10783	0.00293	0.00014
4096	0.05251	0.00073	0.38818	0.17090	0.00013	0.84739	0.00622	0.00049
8192	0.41254	0.00176	3.17630	1.54086	0.00044	6.73315	0.01231	0.00072

Table 6.4: Execution time in seconds for different applications with available sparse formats in cuSparse library

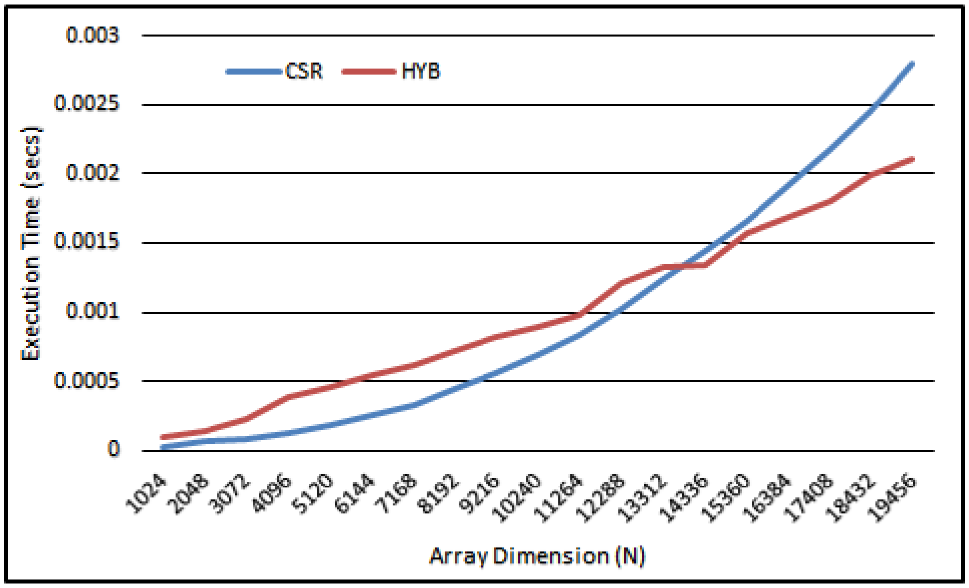


Figure 6.9: Execution time in seconds of spdMV with CSR and HYB matrix formats

6.5 Generation of API Functions for Efficient Calling of cuSparse Library Routines

Since CSR and HYB sparse matrix formats have shown optimized storages (section 6.4), we decided to use them as implicit storage schemes in RT-CUDA. We have implemented API functions to call cuSparse library routines with CSR matrix format for spMM and spdMM operations, and with HYB matrix format for spdMV operation. Tables 6.5 show the execution time in seconds for all three operations spMM, spdMM, and spdMV implemented in RT-CUDA with different matrix sparsity for N=10240.

Furthermore, RT-CUDA provides ability to load standard sparse matrices available in a matrix market file format (an ASCII-based file format designed to facilitate the exchange of matrix data) [86] into a sparse matrix structure to

	Single-Precision			Double-Precision		
Sparsity	spMM	spdMM	spdMV	spMM	spdMM	spdMV
25%	382.7293	20.0087	0.0062	577.9461	33.6707	0.0072
50%	152.2969	13.9083	0.0042	230.9889	22.6241	0.0049
75%	36.6174	7.1978	0.0021	55.1729	11.4095	0.0023
90%	6.2449	3.0196	0.0009	9.0411	4.6379	0.0010

Table 6.5: Execution time in seconds for spMM, spdMM, and spdMV implemented in RT-CUDA for N=10240

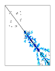
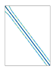
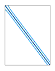
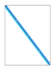
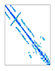
Matrix	**Plot**	**Dimension**	**Non-Zeros**	**Sparsity**
bcsstm13		2003	11973	99.70%
cavity10		2597	76367	98.87%
cavity17		4562	138187	99.34%
cdde1		961	4681	99.49%
coater1		1348	19457	98.93%

Table 6.6: Properties of the Sparse Matrices

be used in RT-CUDA API functions for sparse matrix operations (see Example 5.7.3). We have evaluated the sparse operations of RT-CUDA using various standard sparse matrices in the domain of computational fluid dynamics available in the repository of University of Florida [87] and extracted from the real applications. Table 6.6 shows the properties of these matrices. All of the selected matrices has about 99% sparsity and are bend diagonal in nature. Fig. 6.10 shows the obtained speedup of RTspDMM, RTspdDMM, and RTspdDMV API functions of RT-CUDA (see Table 5.1 for details) over Dense equivalent operation. For RTspDMM, the sparse operation obtained more speedup if the non-zero elements are closed to diagonal as in the case of matrices cavity17 and cdde1 while in the case of bcsstm13, cavity10, and coater1 the speedup is relatively less due to scattered non-zero elements. For RTspdMM, the obtained speedup is seem to be dependent on the sparsity of the matrix. The matrices having more sparsity show more speedup than the matrices having less sparsity. For RTspdMV, the sparse operation obtained more speedup if the non-zero elements are closed to diagonal but with large dimension as in the case of cavity17 but the speedup drops significantly for small dimension as in the case of cdde1.

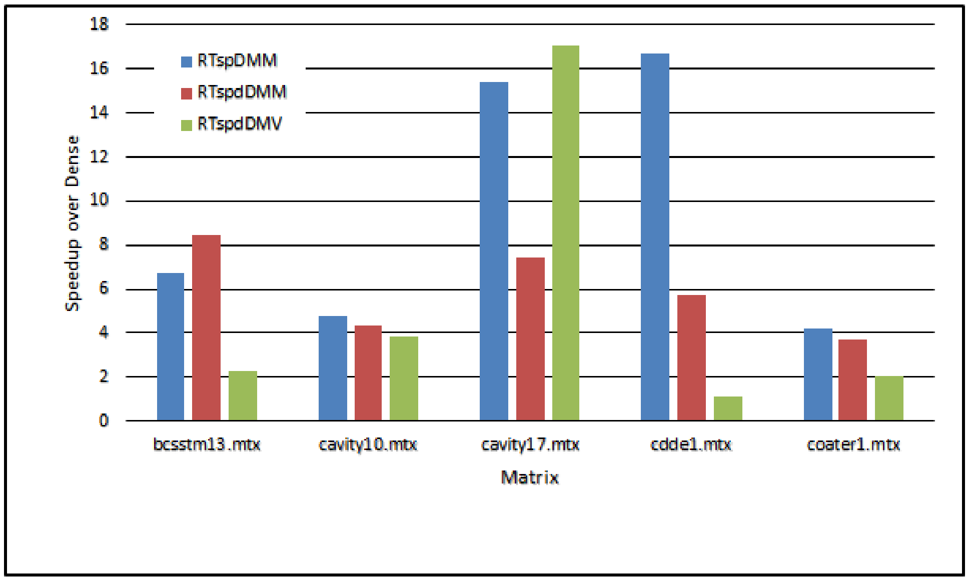


Figure 6.10: Speedup of sparse MM and MV operations in RT-CUDA over Dense operations

CHAPTER 7

CONCLUSION AND FUTURE WORK

Modern GPUs use multiple streaming multiprocessors (SMs) with potentially hundreds of cores, fast context switching, and high memory bandwidth to tolerate ever-increasing latencies to main memory by overlapping long-latency loads in stalled threads with useful computation in other threads. The Compute Unified Device Architecture (CUDA) is a simple C-like interface proposed for programming NVIDIA GPUs. However, porting applications to CUDA remains a challenge to average programmers. CUDA places on the programmer the burden of packaging GPU code in separate functions, of explicitly managing data transfer between the host and GPU memories, and of manually optimizing the utilization of the GPU resources.

In this work, we have explored the GPU architecture and CUDA programming framework to utilize GPU devices for general purpose computing. We have reviewed several numerical algorithms implementations, code transformations to enhance CUDA kernel performance, CUDA kernel optimizations, performance models, auto-tuning frameworks, micro-benchmarking of GPU devices (see Chapter 2). We have studied in detail about the execution model, programming, and synchronization mechanisms of latest GPU architectures including Fermi and Kepler (see Chapter 3).

Based on the study of GPU architectures, CUDA programming framework, and various kernel optimizations, a 3-step algorithm has been proposed to tune the CUDA kernel parameters and enhance GPU resource utilization (see Section 4.1). A detailed analysis of the existing GPGPU frameworks/compilers have been presented (see Section 4.3) including CUDA-lite, hiCUDA, OpenMPC, PGI, OpenACC, HMPP, R-Stream, and CUDA-CHiLL.

A Restructuring Tool Algorithm (RTA-CUDA) has also been presented to generate an optimized CUDA parallel program from a given sequential C program (see Section 5.2). The algorithm generates a parametric CUDA kernel with three parameters that are BLOCK_SIZE, MERGE_LEVEL, and SKEW_LEVEL. A parameter tuning algorithm has also been presented to find an optimal set of CUDA kernel parameters generated by RTA-CUDA (see Section 5.2.5). Based on RTA-CUDA and the parametric tuning algorithm, a Restructuring Tool (RT-CUDA) has been developed with an additional set of API functions to call highly optimized library routines for dense and sparse matrices (cuBLAS and cuSPARSE) and synchronization primitives for inter-block synchronization (see Section 5.3).

RT-CUDA is a software compiler with best possible kernel optimizations to bridge the gap between high-level languages machine dependent CUDA and GPUs. RT-CUDA enables transparent invocation of the most optimized external math libraries like cuSparse, and cuBLAS. RT-CUDA facilitates the design of efficient parallel software for developing parallel simulators (reservoir simulators, molecular dynamics, etc.) which are critical for Aramco and Oil and Gas industry in KSA.

Performance evaluation of the tool has been performed using basic linear algebra operations including Lapack BLAS benchmark, Jacobi iterative solver with different inter-block synchronization primitives, dense and sparse matrix operations (see Chapter 6). RT-CUDA obtained significant speedup over other compilers like PGI OpenACC implementation and GPGPU compiler. Testing of the tool has been performed by some graduate students based on a set of 10 testing cases (see Appendix 7) with progressive difficulties ranging from simple vector matrix operations to full solver of linear system of equations.

The RT-CUDA implementation currently supports single kernel conversion at a time that can be enhanced to provide support for multiple kernels development in a single run that ease the development of complex programs. Also, it can be extended to add support for kernel specific configurations and auto-tuning. Some new optimizations can be implemented targeting emerging GPU architectures such as Maxwell. Futhermore, additional API functions can be added from cuBLAS and cuSparse libraries with different sparse matrix formats.

List of Publications

Journal Publications

- **Ayaz ul Hassan Khan**,Mayez Al-Mouhamed, and Allam Fatayer, "Optimizing the Matrix Multiply Using Strassen and Winograd Algorithms with Limited Recursions on Many Core", Submitted to International Journal of Parallel Programming (IJPP), 2014.
- **Ayaz ul Hassan Khan**, Mayez Al-Mouhamed, Allam Fatayer, Anas Almousa, Abdulrahman Baqais, and Mohammad Assayony, "Padding Free Bank Conflict Resolution for CUDA-Based Matrix Transpose Algorithm", Accepted in International Journal of Networked and Distributed Computing (IJNDC), Vol 2., No. 3, 2014.
- Mayez Al-Mouhamed, and **Ayaz ul Hassan Khan**,"Exploration of Automatic Optimization for CUDA Programming", International Journal of Parallel, Emergent, and Distributed Systems (IJPEDS), 2014. DOI:10.1080/17445760.2014.953158

Conference Publications

- **A. H. Khan**, M. A. Al-Mouhamed, A. Almousa, A. Fatayar, A. R. Ibrahim, and A. J. Siddiqui, AES-128 ECB Encryption on GPUs and Effects of Input

Plaintext Patterns on Performance, 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2014).

- **A. H. Khan**, M. A. Al-Mouhamed, A. Almousa, A. Fatayar, A. Baqais, and M. Assayony, Padding Free Bank Conict Resolution for CUDA-Based Matrix Transpose Algorithm, 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2014).

- Mayez Al-Mouhamed,and **Ayaz ul Hassan Khan**, Exploration of Automatic Optimization for CUDA Programming, 2nd IEEE International Conference on Parallel, Distributed and Grid Computing, India, PDGC2012, 2012.[link].(Second Best Conference Paper Award)

Appendices

Appendix A
ANTLR C Grammar

Listing 1: C Grammar

```
grammar C_RCUDA;

primaryExpression
    :   Identifier
    |   Constant
    |   StringLiteral+
    |   '(' expression ')'
    |   genericSelection
    |   '__extension__'? '(' compoundStatement ')' //
        Blocks (GCC extension)
    |   '__builtin_va_arg' '(' unaryExpression ','
        typeName ')'
    |   '__builtin_offsetof' '(' typeName ','
        unaryExpression ')'
    ;

genericSelection
    :   '_Generic' '(' assignmentExpression ','
        genericAssocList ')'
    ;

genericAssocList
    :   genericAssociation
    |   genericAssocList ',' genericAssociation
    ;

genericAssociation
    :   typeName ':' assignmentExpression
    |   'default' ':' assignmentExpression
    ;

postfixExpression
    :   primaryExpression
    |   postfixExpression '[' expression ']'
    |   postfixExpression '(' argumentExpressionList? ')'
    |   postfixExpression '.' Identifier
    |   postfixExpression '->' Identifier
    |   postfixExpression '++'
    |   postfixExpression '--'
    |   '(' typeName ')' '{' initializerList '}'
    |   '(' typeName ')' '{' initializerList ',' '}'
    |   '__extension__' '(' typeName ')' '{'
       initializerList '}'
    |   '__extension__' '(' typeName ')' '{'
       initializerList ',' '}'
    ;

argumentExpressionList
    :   assignmentExpression
    |   argumentExpressionList ',' assignmentExpression
    ;

unaryExpression
    :   postfixExpression
    |   '++' unaryExpression
    |   '--' unaryExpression
    |   unaryOperator castExpression
    |   'sizeof' unaryExpression
    |   'sizeof' '(' typeName ')'
    |   '_Alignof' '(' typeName ')'
    |   '&&' Identifier // GCC extension address of label
    ;

unaryOperator
    :   '&' | '*' | '+' | '-' | '~' | '!'
    ;

castExpression
    :   unaryExpression
    |   '(' typeName ')' castExpression
    |   '__extension__' '(' typeName ')' castExpression
    ;

multiplicativeExpression
    :   castExpression
    |   multiplicativeExpression '*' castExpression
    |   multiplicativeExpression '/' castExpression
    |   multiplicativeExpression '%' castExpression
    ;

additiveExpression
    :   multiplicativeExpression
    |   additiveExpression '+' multiplicativeExpression
    |   additiveExpression '-' multiplicativeExpression
    ;

shiftExpression
    :   additiveExpression
    |   shiftExpression '<<' additiveExpression
    |   shiftExpression '>>' additiveExpression
    ;

relationalExpression
    :   shiftExpression
    |   relationalExpression '<' shiftExpression
    |   relationalExpression '>' shiftExpression
    |   relationalExpression '<=' shiftExpression
    |   relationalExpression '>=' shiftExpression
    ;

equalityExpression
    :   relationalExpression
    |   equalityExpression '==' relationalExpression
    |   equalityExpression '!=' relationalExpression
    ;

andExpression
    :   equalityExpression
    |   andExpression '&' equalityExpression
    ;

exclusiveOrExpression
    :   andExpression
    |   exclusiveOrExpression '^' andExpression
    ;

inclusiveOrExpression
    :   exclusiveOrExpression
    |   inclusiveOrExpression '|' exclusiveOrExpression
    ;

logicalAndExpression
    :   inclusiveOrExpression
    |   logicalAndExpression '&&' inclusiveOrExpression
    ;

logicalOrExpression
    :   logicalAndExpression
    |   logicalOrExpression '||' logicalAndExpression
    ;
```

```
conditionalExpression
    :   logicalOrExpression ('?' expression ':'
       conditionalExpression)?
    ;

assignmentExpression
    :   conditionalExpression
    |   unaryExpression assignmentOperator
       assignmentExpression
    ;

assignmentOperator
    :   '=' | '*=' | '/=' | '%=' | '+=' | '-=' | '<<=' |
       '>>=' | '&=' | '^=' | '|='
    ;

expression
    :   assignmentExpression
    |   expression ',' assignmentExpression
    ;

constantExpression
    :   conditionalExpression
    ;

declaration
    :   declarationSpecifiers initDeclaratorList? ';'
    |   staticAssertDeclaration
    ;

declarationSpecifiers
    :   declarationSpecifier+
    ;

declarationSpecifiers2
    :   declarationSpecifier+
    ;

declarationSpecifier
    :   storageClassSpecifier
    |   typeSpecifier
    |   typeQualifier
    |   functionSpecifier
    |   alignmentSpecifier
    ;
```

```
initDeclaratorList
    :   initDeclarator
    |   initDeclaratorList ',' initDeclarator
    ;

initDeclarator
    :   declarator
    |   declarator '=' initializer
    ;

storageClassSpecifier
    :   'typedef'
    |   'extern'
    |   'static'
    |   '_Thread_local'
    |   'auto'
    |   'register'
    ;

typeSpecifier
    :   ('void'
    |   'char'
    |   'short'
    |   'int'
    |   'long'
    |   'float'
    |   'double'
    |   'signed'
    |   'unsigned'
    |   '_Bool'
    |   '_Complex'
    |   '__m128'
    |   '__m128d'
    |   '__m128i')
    |   '__extension__' '(' ('__m128' | '__m128d' | '
       __m128i') ')'
    |   atomicTypeSpecifier
    |   structOrUnionSpecifier
    |   enumSpecifier
    |   typedefName
    |   '__typeof__' '(' constantExpression ')' // GCC
       extension
    |   'dim3'
    ;

structOrUnionSpecifier
    :   structOrUnion Identifier? '{'
```

```
        structDeclarationList '}'
    |   structOrUnion Identifier
    ;

structOrUnion
    :   'struct'
    |   'union'
    ;

structDeclarationList
    :   structDeclaration
    |   structDeclarationList structDeclaration
    ;

structDeclaration
    :   specifierQualifierList structDeclaratorList? ';'
    |   staticAssertDeclaration
    ;

specifierQualifierList
    :   typeSpecifier specifierQualifierList?
    |   typeQualifier specifierQualifierList?
    ;

structDeclaratorList
    :   structDeclarator
    |   structDeclaratorList ',' structDeclarator
    ;

structDeclarator
    :   declarator
    |   declarator? ':' constantExpression
    ;

enumSpecifier
    :   'enum' Identifier? '{' enumeratorList '}'
    |   'enum' Identifier? '{' enumeratorList ',' '}'
    |   'enum' Identifier
    ;

enumeratorList
    :   enumerator
    |   enumeratorList ',' enumerator
    ;

enumerator
    :   enumerationConstant
    |   enumerationConstant '=' constantExpression
    ;

enumerationConstant
    :   Identifier
    ;

atomicTypeSpecifier
    :   '_Atomic' '(' typeName ')'
    ;

typeQualifier
    :   'const'
    |   'restrict'
    |   '__restrict__'
    |   'volatile'
    |   '_Atomic'
    ;

functionSpecifier
    :   ('inline'
    |   '_Noreturn'
    |   '__inline__' // GCC extension
    |   '__stdcall')
    |   gccAttributeSpecifier
    |   '__declspec' '(' Identifier ')'
    ;

alignmentSpecifier
    :   '_Alignas' '(' typeName ')'
    |   '_Alignas' '(' constantExpression ')'
    ;

declarator
    :   pointer? directDeclarator gccDeclaratorExtension*
    ;

directDeclarator
    :   Identifier
    |   '(' declarator ')'
    |   directDeclarator '[' typeQualifierList?
       assignmentExpression? ']'
    |   directDeclarator '[' 'static' typeQualifierList?
       assignmentExpression ']'
    |   directDeclarator '[' typeQualifierList 'static'
       assignmentExpression ']'
    |   directDeclarator '[' typeQualifierList? '*' ']'
    |   directDeclarator '(' parameterTypeList ')'
    |   directDeclarator '(' identifierList? ')'
    ;

gccDeclaratorExtension
    :   '__asm' '(' StringLiteral+ ')'
    |   gccAttributeSpecifier
    ;

gccAttributeSpecifier
    :   '__attribute__' '(' '(' gccAttributeList ')' ')'
    ;

gccAttributeList
    :   gccAttribute (',' gccAttribute)*
    |   // empty
    ;

gccAttribute
    :   ~(',' | '(' | ')') // relaxed def for "identifier
        or reserved word"
        ('(' argumentExpressionList? ')')?
    |   // empty
    ;

nestedParenthesesBlock
    :   (   ~('(' | ')')
        |   '(' nestedParenthesesBlock ')'
        )*
    ;

pointer
    :   '*' typeQualifierList?
    |   '*' typeQualifierList? pointer
    |   '^' typeQualifierList? // Blocks language
       extension
    |   '^' typeQualifierList? pointer // Blocks language
        extension
    ;

typeQualifierList
    :   typeQualifier
    |   typeQualifierList typeQualifier
    ;

parameterTypeList
    :   parameterList
    |   parameterList ',' '...'
    ;

parameterList
    :   parameterDeclaration
    |   parameterList ',' parameterDeclaration
    ;

parameterDeclaration
    :   declarationSpecifiers declarator
    |   declarationSpecifiers2 abstractDeclarator?
    ;

identifierList
    :   Identifier
    |   identifierList ',' Identifier
    ;

typeName
    :   specifierQualifierList abstractDeclarator?
    ;

abstractDeclarator
    :   pointer
    |   pointer? directAbstractDeclarator
       gccDeclaratorExtension*
    ;

directAbstractDeclarator
    :   '(' abstractDeclarator ')' gccDeclaratorExtension
       *
    |   '[' typeQualifierList? assignmentExpression? ']'
    |   '[' 'static' typeQualifierList?
       assignmentExpression ']'
    |   '[' typeQualifierList 'static'
       assignmentExpression ']'
    |   '[' '*' ']'
    |   '(' parameterTypeList? ')' gccDeclaratorExtension
       *
    |   directAbstractDeclarator '[' typeQualifierList?
       assignmentExpression? ']'
    |   directAbstractDeclarator '[' 'static'
       typeQualifierList? assignmentExpression ']'
    |   directAbstractDeclarator '[' typeQualifierList '
       static' assignmentExpression ']'
    |   directAbstractDeclarator '[' '*' ']'
    |   directAbstractDeclarator '(' parameterTypeList? '
        )' gccDeclaratorExtension*
    ;

typedefName
    :   Identifier
    ;

initializer
    :   assignmentExpression
    |   '{' initializerList '}'
    |   '{' initializerList ',' '}'
    ;

initializerList
    :   designation? initializer
    |   initializerList ',' designation? initializer
    ;

designation
    :   designatorList '='
    ;

designatorList
    :   designator
    |   designatorList designator
    ;

designator
    :   '[' constantExpression ']'
    |   '.' Identifier
    ;

staticAssertDeclaration
    :   '_Static_assert' '(' constantExpression ','
       StringLiteral+ ')' ';'
    ;

statement
    :   labeledStatement
    |   compoundStatement
    |   expressionStatement
    |   selectionStatement
    |   iterationStatement
    |   jumpStatement
    |   dim3Statement
    |   cudaKernelInvocation
    |   ('__asm' | '__asm__') ('volatile' | '__volatile__
        ') '(' (logicalOrExpression (','
        logicalOrExpression)*)? (':' (logicalOrExpression
        (',' logicalOrExpression)*)?)* ')' ';'
    ;

dim3Statement
    :   typeSpecifier Identifier expression ';'
    ;

labeledStatement
    :   Identifier ':' statement
    |   'case' constantExpression ':' statement
    |   'default' ':' statement
    ;

compoundStatement
    :   '{' blockItemList? '}'
    ;

blockItemList
    :   blockItem
    |   blockItemList blockItem
    ;

blockItem
    :   declaration
    |   statement
    ;

expressionStatement
    :   expression? ';'
    ;

selectionStatement
    :   'if' '(' expression ')' statement ('else'
        statement)?
    |   'switch' '(' expression ')' statement
    ;

iterationStatement
    :   'while' '(' expression ')' statement
    |   'do' statement 'while' '(' expression ')' ';'
    |   'for' '(' expression? ';' expression? ';'
        expression? ')' statement
    |   'for' '(' declaration expression? ';' expression?
        ')' statement
    ;
```

```
jumpStatement
    :   'goto' Identifier ';'
    |   'continue' ';'
    |   'break' ';'
    |   'return' expression? ';'
    |   'goto' unaryExpression ';' // GCC extension
    ;

compilationUnit
    :   translationUnit? EOF
    ;

translationUnit
    :   externalDeclaration
    |   translationUnit externalDeclaration
    ;

externalDeclaration
    :   functionDefinition
    |   declaration
    |   ';' // stray ;
    ;

functionDefinition
    :   declarationSpecifiers? declarator declarationList
       ? compoundStatement
    ;

declarationList
    :   declaration
    |   declarationList declaration
    ;

cudaKernelInvocation
    :   primaryExpression '<<<' expression '>>>' '('
       expression? ')' ';'
    ;

Auto : 'auto';
Break : 'break';
Case : 'case';
Char : 'char';
Const : 'const';
Continue : 'continue';
Default : 'default';
Do : 'do';
```

```
Double : 'double';
Else : 'else';
Enum : 'enum';
Extern : 'extern';
Float : 'float';
For : 'for';
Goto : 'goto';
If : 'if';
Inline : 'inline';
Int : 'int';
Long : 'long';
Register : 'register';
Restrict : 'restrict';
CUDARestrict : '__restrict__';
Return : 'return';
Short : 'short';
Signed : 'signed';
Sizeof : 'sizeof';
Static : 'static';
Struct : 'struct';
Switch : 'switch';
Typedef : 'typedef';
Union : 'union';
Unsigned : 'unsigned';
Void : 'void';
Volatile : 'volatile';
While : 'while';
Dim3 : 'dim3';

Alignas : '_Alignas';
Alignof : '_Alignof';
Atomic : '_Atomic';
Bool : '_Bool';
Complex : '_Complex';
Generic : '_Generic';
Imaginary : '_Imaginary';
Noreturn : '_Noreturn';
StaticAssert : '_Static_assert';
ThreadLocal : '_Thread_local';

LeftParen : '(';
RightParen : ')';
LeftBracket : '[';
RightBracket : ']';
LeftBrace : '{';
RightBrace : '}';
```

```
Less : '<';
LessEqual : '<=';
Greater : '>';
GreaterEqual : '>=';
LeftShift : '<<';
RightShift : '>>';

Plus : '+';
PlusPlus : '++';
Minus : '-';
MinusMinus : '--';
Star : '*';
Div : '/';
Mod : '%';

And : '&';
Or : '|';
AndAnd : '&&';
OrOr : '||';
Caret : '^';
Not : '!';
Tilde : '~';

Question : '?';
Colon : ':';
Semi : ';';
Comma : ',';

Assign : '=';
StarAssign : '*=';
DivAssign : '/=';
ModAssign : '%=';
PlusAssign : '+=';
MinusAssign : '-=';
LeftShiftAssign : '<<=';
RightShiftAssign : '>>=';
AndAssign : '&=';
XorAssign : '^=';
OrAssign : '|=';
CUDAFunctionCallInit : '<<<';
CUDAFunctionCallEnd : '>>>';

Equal : '==';
NotEqual : '!=';

Arrow : '->';
Dot : '.';
```

```
Ellipsis : '...';

Identifier
    :   IdentifierNondigit
        (   IdentifierNondigit
        |   Digit
        )*
    ;

fragment
IdentifierNondigit
    :   Nondigit
    |   UniversalCharacterName
    ;

fragment
Nondigit
    :   [a-zA-Z_]
    ;

fragment
Digit
    :   [0-9]
    ;

fragment
UniversalCharacterName
    :   '\\u' HexQuad
    |   '\\U' HexQuad HexQuad
    ;

fragment
HexQuad
    :   HexadecimalDigit HexadecimalDigit
       HexadecimalDigit HexadecimalDigit
    ;

Constant
    :   IntegerConstant
    |   FloatingConstant
    |   CharacterConstant
    ;

fragment
IntegerConstant
    :   DecimalConstant IntegerSuffix?
    |   OctalConstant IntegerSuffix?
    |   HexadecimalConstant IntegerSuffix?
    ;

fragment
DecimalConstant
    :   NonzeroDigit Digit*
    ;

fragment
OctalConstant
    :   '0' OctalDigit*
    ;

fragment
HexadecimalConstant
    :   HexadecimalPrefix HexadecimalDigit+
    ;

fragment
HexadecimalPrefix
    :   '0' [xX]
    ;

fragment
NonzeroDigit
    :   [1-9]
    ;

fragment
OctalDigit
    :   [0-7]
    ;

fragment
HexadecimalDigit
    :   [0-9a-fA-F]
    ;

fragment
IntegerSuffix
    :   UnsignedSuffix LongSuffix?
    |   UnsignedSuffix LongLongSuffix
    |   LongSuffix UnsignedSuffix?
    |   LongLongSuffix UnsignedSuffix?
    ;

fragment
```

```
UnsignedSuffix
    :   [uU]
    ;

fragment
LongSuffix
    :   [lL]
    ;

fragment
LongLongSuffix
    :   'll' | 'LL'
    ;

fragment
FloatingConstant
    :   DecimalFloatingConstant
    |   HexadecimalFloatingConstant
    ;

fragment
DecimalFloatingConstant
    :   FractionalConstant ExponentPart? FloatingSuffix?
    |   DigitSequence ExponentPart FloatingSuffix?
    ;

fragment
HexadecimalFloatingConstant
    :   HexadecimalPrefix HexadecimalFractionalConstant
       BinaryExponentPart FloatingSuffix?
    |   HexadecimalPrefix HexadecimalDigitSequence
       BinaryExponentPart FloatingSuffix?
    ;

fragment
FractionalConstant
    :   DigitSequence? '.' DigitSequence
    |   DigitSequence '.'
    ;

fragment
ExponentPart
    :   'e' Sign? DigitSequence
    |   'E' Sign? DigitSequence
    ;

fragment
```

```
Sign
    :   '+' | '-'
    ;

fragment
DigitSequence
    :   Digit+
    ;

fragment
HexadecimalFractionalConstant
    :   HexadecimalDigitSequence? '.'
       HexadecimalDigitSequence
    |   HexadecimalDigitSequence '.'
    ;

fragment
BinaryExponentPart
    :   'p' Sign? DigitSequence
    |   'P' Sign? DigitSequence
    ;

fragment
HexadecimalDigitSequence
    :   HexadecimalDigit+
    ;

fragment
FloatingSuffix
    :   'f' | 'l' | 'F' | 'L'
    ;

fragment
CharacterConstant
    :   '\'' CCharSequence '\''
    |   'L\'' CCharSequence '\''
    |   'u\'' CCharSequence '\''
    |   'U\'' CCharSequence '\''
    ;

fragment
CCharSequence
    :   CChar+
    ;

fragment
CChar
```

```
    :   ~['\\\r\n]
    |   EscapeSequence
    ;

fragment
EscapeSequence
    :   SimpleEscapeSequence
    |   OctalEscapeSequence
    |   HexadecimalEscapeSequence
    |   UniversalCharacterName
    ;

fragment
SimpleEscapeSequence
    :   '\\' ['"?abfnrtv\\]
    ;

fragment
OctalEscapeSequence
    :   '\\' OctalDigit
    |   '\\' OctalDigit OctalDigit
    |   '\\' OctalDigit OctalDigit OctalDigit
    ;

fragment
HexadecimalEscapeSequence
    :   '\\x' HexadecimalDigit+
    ;

StringLiteral
    :   EncodingPrefix? '"' SCharSequence? '"'
    ;

fragment
EncodingPrefix
    :   'u8'
    |   'u'
    |   'U'
    |   'L'
    ;

fragment
SCharSequence
    :   SChar+
    ;

fragment
```

```
SChar
    :   ~["\\\r\n]
    |   EscapeSequence
    ;

LineDirective
    :   '#' Whitespace? DecimalConstant Whitespace?
       StringLiteral ~[\r\n]*
        -> skip
    ;

PragmaDirective
    :   '#' Whitespace? 'pragma' Whitespace ~[\r\n]*
        -> skip
    ;

Whitespace
    :   [ \t]+
        -> skip
    ;

Newline
    :   (   '\r' '\n'?
        |   '\n'
        )
        -> skip
    ;

BlockComment
    :   '/*' .*? '*/'
        -> skip
    ;

LineComment
    :   '//' ~[\r\n]*
        -> skip
    ;
```

Appendix B

RT-CUDA Transformations

Listing 2: GlobalValues.java

```
public class GlobalValues {
    static int loopCount=0;
    static boolean nestLoopFound = false;
    static String BlockMergeIndex = "";
    static boolean enteredExpressionStatement = false;
    static ArrayList<String>
       BlockMergeIndexLValueIdentifier = new ArrayList
       <>();
    static boolean applyBlockSkew = false;
    static String BlockSkewIndex = "";
    static ArrayList<String>
       BlockSkewIndexLValueIdentifier = new ArrayList<>()
       ;
    static boolean applyPrefetching = false;
    static ArrayList<String> prefetchedArrayIdentifiers =
        new ArrayList<>();
    static ArrayList<String>
       nonPrefetchedArrayIdentifiers = new ArrayList<>();
    static String typeSpecifier = "float";
    static boolean sharedMemoryDeclared = false;
    static String prefetchingLoopIndex = "";
    static String prefetchingArrayExpression = "";
    static String nonPrefetchingArrayExpression = "";
    static String lastLoopIndex = "";
    static int prefetchingLoopLineNumber = 0;
    static boolean prefetechedLoopEntered = false;
    static boolean prefetchedArrayExpressionFound = false
       ;
    static boolean nonPrefetchedArrayExpressionFound =
       false;
    static boolean applyLoopCollapsing = true;
    static ArrayList<String> kernelNameList = new
       ArrayList<>();
    static boolean _2DMatrix = false;
    static String rowDim = "N";
    static int tabCount = 0;
    static boolean iterationStatment = false;
    static ArrayList<Integer> endIterationDecs = new
       ArrayList<>();

    public static ArrayList<Integer> getEndIterationDecs
       () {
        return endIterationDecs;
    }

    public static void setEndIterationDecs(int
       endIterationDecs) {
        GlobalValues.endIterationDecs.add(
           endIterationDecs);
    }

    public static boolean isIterationStatment() {
        return iterationStatment;
    }

    public static void setIterationStatment(boolean
       iterationStatment) {
        GlobalValues.iterationStatment =
           iterationStatment;
    }

    public static int getTabCount() {
        return tabCount;
    }

    public static void setTabCount(int tabCount) {
        GlobalValues.tabCount = tabCount;
    }

    public static void incTabCount() {
        GlobalValues.tabCount++;
    }
    public static void decTabCount() {
        GlobalValues.tabCount--;
    }

    public static String getRowDim() {
        return rowDim;
    }

    public static void setRowDim(String rowDim) {
        GlobalValues.rowDim = rowDim;
    }
```

```
    public static boolean is2DMatrix() {
        return _2DMatrix;
    }

    public static void set2DMatrix(boolean _2DMatrix) {
        GlobalValues._2DMatrix = _2DMatrix;
    }

    public static ArrayList<String> getKernelNameList() {
        return kernelNameList;
    }

    public static void setKernelNameList(String
       kernelName) {
        GlobalValues.kernelNameList.add(kernelName);
    }

    public static boolean isApplyLoopCollapsing() {
        return applyLoopCollapsing;
    }

    public static void setApplyLoopCollapsing(boolean
       applyLoopCollapsing) {
        GlobalValues.applyLoopCollapsing =
           applyLoopCollapsing;
    }

    public static boolean
       isNonPrefetchedArrayExpressionFound() {
        return nonPrefetchedArrayExpressionFound;
    }

    public static void
       setNonPrefetchedArrayExpressionFound(boolean
       nonPrefetchedArrayExpressionFound) {
        GlobalValues.nonPrefetchedArrayExpressionFound =
           nonPrefetchedArrayExpressionFound;
    }

    public static ArrayList<String>
       getNonPrefetchedArrayIdentifiers() {
        return nonPrefetchedArrayIdentifiers;
    }

    public static void setNonPrefetchedArrayIdentifiers(
       String nonPrefetchedArrayIdentifiers) {
         GlobalValues.nonPrefetchedArrayIdentifiers.add(
            nonPrefetchedArrayIdentifiers);
    }

    public static String getNonPrefetchingArrayExpression
       () {
         return nonPrefetchingArrayExpression;
    }

    public static void setNonPrefetchingArrayExpression(
       String nonPrefetchingArrayExpression) {
         GlobalValues.nonPrefetchingArrayExpression =
            nonPrefetchingArrayExpression;
    }

    public static boolean
       isPrefetchedArrayExpressionFound() {
         return prefetchedArrayExpressionFound;
    }

    public static void setPrefetchedArrayExpressionFound(
       boolean prefetchedArrayExpressionFound) {
         GlobalValues.prefetchedArrayExpressionFound =
            prefetchedArrayExpressionFound;
    }

    public static boolean isPrefetechedLoopEntered() {
         return prefetechedLoopEntered;
    }

    public static void setPrefetechedLoopEntered(boolean
       prefetechedLoopEntered) {
         GlobalValues.prefetechedLoopEntered =
            prefetechedLoopEntered;
    }

    public static int getPrefetchingLoopLineNumber() {
         return prefetchingLoopLineNumber;
    }

    public static void setPrefetchingLoopLineNumber(int
       prefetchingLoopLineNumber) {
         GlobalValues.prefetchingLoopLineNumber =
            prefetchingLoopLineNumber;
    }

    public static String getLastLoopIndex() {
         return lastLoopIndex;
    }

    public static void setLastLoopIndex(String
       loopIndices) {
         GlobalValues.lastLoopIndex = loopIndices;

    }

    public static String getPrefetchingArrayExpression()
       {
         return prefetchingArrayExpression;
    }

    public static void setPrefetchingArrayExpression(
       String prefetchingArrayExpression) {
         GlobalValues.prefetchingArrayExpression =
            prefetchingArrayExpression;
    }

    public static String getPrefetchingLoopIndex() {
         return prefetchingLoopIndex;
    }

    public static void setPrefetchingLoopIndex(String
       prefetchingLoopIndex) {
         GlobalValues.prefetchingLoopIndex =
            prefetchingLoopIndex;
    }

    public static boolean isSharedMemoryDeclared() {
         return sharedMemoryDeclared;
    }

    public static void setSharedMemoryDeclared(boolean
       sharedMemoryDeclared) {
         GlobalValues.sharedMemoryDeclared =
            sharedMemoryDeclared;
    }

    public static String getTypeSpecifier() {
         return typeSpecifier;
    }

    public static void setTypeSpecifier(String
       typeSpecifier) {
         GlobalValues.typeSpecifier = typeSpecifier;
```

```
    }

    public static ArrayList<String>
       getPrefetchedArrayIdentifiers() {
        return prefetchedArrayIdentifiers;
    }

    public static void setPrefetchedArrayIdentifiers(
       String prefetchedArrayIdentifiers) {
        GlobalValues.prefetchedArrayIdentifiers.add(
           prefetchedArrayIdentifiers);
    }

    public static boolean isApplyPrefetching() {
        return applyPrefetching;
    }

    public static void setApplyPrefetching(boolean
       applyPrefetching) {
        GlobalValues.applyPrefetching = applyPrefetching;
    }

    public static boolean isApplyBlockSkew() {
        return applyBlockSkew;
    }

    public static void setApplyBlockSkew(boolean
       applyBlockSkew) {
        GlobalValues.applyBlockSkew = applyBlockSkew;
    }

    public static String getBlockSkewIndex() {
        return BlockSkewIndex;
    }

    public static void setBlockSkewIndex(String
       BlockSkewIndex) {
        GlobalValues.BlockSkewIndex = BlockSkewIndex;
    }

    public static ArrayList<String>
       getBlockSkewIndexLValueIdentifier() {
        return BlockSkewIndexLValueIdentifier;
    }

    public static void setBlockSkewIndexLValueIdentifier(
       String BlockSkewIndexLValueIdentifier) {
        GlobalValues.BlockSkewIndexLValueIdentifier.add(
            BlockSkewIndexLValueIdentifier);
    }

    public static ArrayList<String>
        getBlockMergeIndexLValueIdentifier() {
        return BlockMergeIndexLValueIdentifier;
    }

    public static void setBlockMergeIndexLValueIdentifier
        (String BlockMergeIndexLValueIdentifier) {
        GlobalValues.BlockMergeIndexLValueIdentifier.add(
            BlockMergeIndexLValueIdentifier);
    }

    public static boolean isEnteredExpressionStatement()
        {
        return enteredExpressionStatement;
    }

    public static void setEnteredExpressionStatement(
        boolean enteredExpressionStatement) {
        GlobalValues.enteredExpressionStatement =
            enteredExpressionStatement;
    }

    public static String getBlockMergeIndex() {
        return BlockMergeIndex;
    }

    public static void setBlockMergeIndex(String
        BlockMergeIndex) {
        GlobalValues.BlockMergeIndex = BlockMergeIndex;
    }

    public static boolean isNestLoopFound() {
        return nestLoopFound;
    }

    public static void setNestLoopFound(boolean
        nestLoopFound) {
        GlobalValues.nestLoopFound = nestLoopFound;
    }
    static IterationStatementContext firstLoop = null;

    public static IterationStatementContext getFirstLoop
        () {
```

```
        return firstLoop;
    }

    public static void setFirstLoop(
       IterationStatementContext firstLoop) {
        GlobalValues.firstLoop = firstLoop;
    }

    public static int getLoopCount() {
        return loopCount;
    }

    public static void setLoopCount(int loopCount) {
        GlobalValues.loopCount = loopCount;
    }

    public static void incrementLoopCount(){
        loopCount++;
    }
    public static void decrementLoopCount(){
        loopCount--;
    }

    public static void resetLoopCount(){
        setLoopCount(0);
    }
}
```

Listing 3: LoopCollapsingListener.java

```
public class LoopCollapsingListener extends
   C_RCUDABaseListener{
        Override public void enterIterationStatement(NotNull
           C_RCUDAParser.IterationStatementContext ctx) {
            GlobalValues.incrementLoopCount();
            int loop_count = GlobalValues.getLoopCount();
            switch(loop_count){
                case 1:
                    GlobalValues.setFirstLoop(ctx);
                    break;
                case 2:
                    GlobalValues.setNestLoopFound(true);
                    C_RCUDAParser.
                       IterationStatementContext
                       first_loop = GlobalValues.
                       getFirstLoop();
```

```
first_loop.removeLastChild();
System.out.println(first_loop.getText
   ());
for(int i=0; i < ctx.getChildCount();
    i++){
    if(ctx.getChild(i).getClass().
       getSimpleName().equals("
       StatementContext")){
        String first_dec = first_loop
           .declaration().
           initDeclaratorList().
           initDeclarator().
           declarator().getText();
        String second_dec = ctx.
           declaration().
           initDeclaratorList().
           initDeclarator().
           declarator().getText();
        String dec = first_dec +
           second_dec;
        String first_limit =
           first_loop.expression(0).
           assignmentExpression().
           conditionalExpression().
           logicalOrExpression().
           logicalAndExpression().
           inclusiveOrExpression().
           exclusiveOrExpression().
           andExpression().
           equalityExpression().
           relationalExpression().
           shiftExpression().
           additiveExpression().
           multiplicativeExpression()
           .castExpression().
           unaryExpression().
           postfixExpression().
           primaryExpression().
           getText();
        String second_limit = ctx.
           expression(0).
           assignmentExpression().
           conditionalExpression().
           logicalOrExpression().
           logicalAndExpression().
           inclusiveOrExpression().
           exclusiveOrExpression().
```

```
            andExpression().
            equalityExpression().
            relationalExpression().
            shiftExpression().
            additiveExpression().
            multiplicativeExpression()
            .castExpression().
            unaryExpression().
            postfixExpression().
            primaryExpression().
            getText();
        String limit = first_limit +
            "*" + second_limit;
        first_loop.declaration().
            initDeclaratorList().
            initDeclarator().
            declarator().
            removeLastChild();
        first_loop.declaration().
            initDeclaratorList().
            initDeclarator().
            declarator().addChild(new
            CommonToken(0, dec));
        first_loop.expression(0).
            assignmentExpression().
            conditionalExpression().
            logicalOrExpression().
            logicalAndExpression().
            inclusiveOrExpression().
            exclusiveOrExpression().
            andExpression().
            equalityExpression().
            relationalExpression().
            relationalExpression().
            shiftExpression().
            additiveExpression().
            multiplicativeExpression()
            .castExpression().
            unaryExpression().
            postfixExpression().
            primaryExpression().
            removeLastChild();
        first_loop.expression(0).
            assignmentExpression().
            conditionalExpression().
            logicalOrExpression().
            logicalAndExpression().
```

```
                                        inclusiveOrExpression().
                                        exclusiveOrExpression().
                                        andExpression().
                                        equalityExpression().
                                        relationalExpression().
                                        relationalExpression().
                                        shiftExpression().
                                        additiveExpression().
                                        multiplicativeExpression()
                                        .castExpression().
                                        unaryExpression().
                                        postfixExpression().
                                        primaryExpression().
                                        addChild(new CommonToken
                                        (0, dec));
                                   first_loop.expression(1).
                                        assignmentExpression().
                                        conditionalExpression().
                                        logicalOrExpression().
                                        logicalAndExpression().
                                        inclusiveOrExpression().
                                        exclusiveOrExpression().
                                        andExpression().
                                        equalityExpression().
                                        relationalExpression().
                                        shiftExpression().
                                        additiveExpression().
                                        multiplicativeExpression()
                                        .castExpression().
                                        unaryExpression().
                                        postfixExpression().
                                        postfixExpression().
                                        primaryExpression().
                                        removeLastChild();
                                   first_loop.expression(1).
                                        assignmentExpression().
                                        conditionalExpression().
                                        logicalOrExpression().
                                        logicalAndExpression().
                                        inclusiveOrExpression().
                                        exclusiveOrExpression().
                                        andExpression().
                                        equalityExpression().
                                        relationalExpression().
                                        shiftExpression().
                                        additiveExpression().
                                        multiplicativeExpression()
```

```
         .castExpression().
         unaryExpression().
         postfixExpression().
         postfixExpression().
         primaryExpression().
         addChild(new CommonToken
         (0, dec));
      first_loop.expression(0).
         assignmentExpression().
         conditionalExpression().
         logicalOrExpression().
         logicalAndExpression().
         inclusiveOrExpression().
         exclusiveOrExpression().
         andExpression().
         equalityExpression().
         relationalExpression().
         shiftExpression().
         additiveExpression().
         multiplicativeExpression()
         .castExpression().
         unaryExpression().
         postfixExpression().
         primaryExpression().
         removeLastChild();
      first_loop.expression(0).
         assignmentExpression().
         conditionalExpression().
         logicalOrExpression().
         logicalAndExpression().
         inclusiveOrExpression().
         exclusiveOrExpression().
         andExpression().
         equalityExpression().
         relationalExpression().
         shiftExpression().
         additiveExpression().
         multiplicativeExpression()
         .castExpression().
         unaryExpression().
         postfixExpression().
         primaryExpression().
         addChild(new CommonToken
         (0, limit));
      String s1 = "int " +
         first_dec + "=(" + dec + "
         /" + first_limit + ")*
```

```
                                    MERGE_LEVEL;";
                                 GlobalValues.
                                    setBlockMergeIndex(
                                    first_dec);
                                 String s2 = "";
                                 if(GlobalValues.
                                    isApplyBlockSkew()){
                                     s2 = "int " + second_dec
                                        + "=(" + dec + "%" +
                                        second_limit + ")*
                                        SKEW_LEVEL;";
                                     GlobalValues.
                                        setBlockSkewIndex(
                                        second_dec);
                                 }
                                 else
                                     s2 = "int " + second_dec
                                        + "=(" + dec + "%" +
                                        second_limit + ");";
                                 first_loop.addChild(new
                                    TerminalNodeImpl(new
                                    CommonToken(0, "{")));
                                 first_loop.addChild(new
                                    TerminalNodeImpl(new
                                    CommonToken(0, s1+s2)));
                                 first_loop.addChild((
                                    RuleContext)ctx.getChild(i
                                    ).getPayload());
                                 first_loop.addChild(new
                                    TerminalNodeImpl(new
                                    CommonToken(0, "}")));
                             }
                         }
                         break;
             }
         }

         Override public void exitIterationStatement(NotNull
            C_RCUDAParser.IterationStatementContext ctx) {
             GlobalValues.decrementLoopCount();
         }

         Override public void exitTypeSpecifier(NotNull
            C_RCUDAParser.TypeSpecifierContext ctx){
             ctx.addChild(new CommonToken(ctx.
                getChildCount(), " "));
         }
```

```
}
```

Listing 4: ArrayTransformationListener.java

```
public class ArrayTransformationListener extends
   C_RCUDABaseListener {
    Override public void enterUnaryExpression(NotNull
       C_RCUDAParser.UnaryExpressionContext ctx){
        String expr = ctx.getText();
        if(expr.contains("][")){
            String strs[] = expr.split("\\[");
            //transformed array creation, DIM is the
               dimension of each matrix assuming square
               matrices of same dimensions
            String texpr = strs[0] + "[(" + strs[1].split
               ("]")[0] + ")*" + GlobalValues.getRowDim()
                + "+(" + strs[2].split("]")[0] + ")]";
            ctx.removeLastChild();
            ctx.addChild(new TerminalNodeImpl(new
               CommonToken(0, texpr)));
        }
    }

    Override public void exitTypeSpecifier(NotNull C_RCUDAParser
       .TypeSpecifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
            " "));
    }
}
```

Listing 5: LoopPartitioning.java

```
public class LoopPartitioning extends C_RCUDABaseListener
    {
    Override public void enterIterationStatement(NotNull
       C_RCUDAParser.IterationStatementContext ctx) {
        GlobalValues.incrementLoopCount();
        if(GlobalValues.getLoopCount() == 1){
            String dec = ctx.declaration().
               initDeclaratorList().initDeclarator().
               declarator().getText();
            String tid = "int tid=threadIdx.x;";
            String bid = "int bid=blockIdx.x;";
            String index;
            if(GlobalValues.isNestLoopFound())
                index = "int " + dec + "=bid*BLOCKSIZE+
                   tid;";
            else{
                index = "int " + dec + "=(bid*BLOCKSIZE+
                   tid)*MERGE_LEVEL;";
                GlobalValues.setBlockMergeIndex(dec);
            }

            RuleContext r = (RuleContext)ctx.getChild(ctx
               .getChildCount()-1).getPayload();
            StatementContext p = (StatementContext)ctx.
               getParent();
            p.removeLastChild();
            p.addChild(new TerminalNodeImpl(new
               CommonToken(0,tid)));
            p.addChild(new TerminalNodeImpl(new
               CommonToken(1,bid)));
            p.addChild(new TerminalNodeImpl(new
               CommonToken(2,index)));
            p.addChild(r);
        }
    }

    Override public void exitIterationStatement(NotNull
       C_RCUDAParser.IterationStatementContext ctx) {
        GlobalValues.decrementLoopCount();
    }

    Override public void exitTypeSpecifier(NotNull C_RCUDAParser
       .TypeSpecifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
            " "));
    }

    Override public void enterPrimaryExpression(NotNull
       C_RCUDAParser.PrimaryExpressionContext ctx){
        if(ctx.getText().equals(GlobalValues.
           getBlockMergeIndex())){
            ParserRuleContext p = ctx.getParent();
            while(!(p.getClass().getSimpleName().equals("
               AssignmentExpressionContext") && p.
               getChildCount() > 1)){
                p = p.getParent();
                if(p == null) return;
            }
            String operator = p.getChild(1).getText();
            while(!p.getClass().getSimpleName().equals("
               ExpressionStatementContext")){
                p = p.getParent();
            }
            GlobalValues.
               setBlockMergeIndexLValueIdentifier(p.
               getText().substring(0, p.getText().indexOf
               (operator)));
        }
        if(GlobalValues.isApplyBlockSkew() && ctx.getText
           ().equals(GlobalValues.getBlockSkewIndex())){
            ParserRuleContext p = ctx.getParent();
            while(!(p.getClass().getSimpleName().equals("
               AssignmentExpressionContext") && p.
               getChildCount() > 1)){
                p = p.getParent();
                if(p == null) return;
            }
            String operator = p.getChild(1).getText();
            while(!p.getClass().getSimpleName().equals("
               ExpressionStatementContext")){
                p = p.getParent();
            }
            GlobalValues.
               setBlockSkewIndexLValueIdentifier(p.
               getText().substring(0, p.getText().indexOf
               (operator)));
        }
    }

        Override public void exitFunctionDefinition(NotNull
           C_RCUDAParser.FunctionDefinitionContext ctx) {
            String funcDef = ctx.getText();
            String rc = ctx.getText();
            for(int i=0; i < ctx.getChildCount(); i++){
                ctx.removeLastChild();
            }
            ctx.removeLastChild();

            ctx.addChild(new CommonToken(0, "__global__ "
               ));
            ctx.addChild(new CommonToken(0, rc));
        }
}
```

Listing 6: BlockMerging.java

```
public class BlockMerging extends C_RCUDABaseListener {
    Override public void enterPrimaryExpression(NotNull
       C_RCUDAParser.PrimaryExpressionContext ctx){
        if(ctx.getText().equals(GlobalValues.
           getBlockMergeIndex())){
            ctx.removeLastChild();
            ctx.addChild(new CommonToken(0, "("+
               GlobalValues.getBlockMergeIndex()+"+m)"));
            GlobalValues.setEnteredExpressionStatement(
               true);
        }
        boolean lValueFound = false;
        ArrayList<String> lValueList = GlobalValues.
           getBlockMergeIndexLValueIdentifier();
        for(int i=0; i < lValueList.size(); i++)
            if(ctx.getText().equals(lValueList.get(i)))
                lValueFound = true;
        if(lValueFound){
            String lvalue = ctx.getText();
            ctx.removeLastChild();
            ctx.addChild(new TerminalNodeImpl(new
               CommonToken(0, lvalue+"[m]")));

            ParserRuleContext p = ctx.getParent();
            while(!(p.getClass().getSimpleName().equals("
               AssignmentExpressionContext") && p.
               getChildCount() == 3))
                p = p.getParent();
            boolean uniTree = true;
            p = (ParserRuleContext)p.getChild(2);

            while(p != null){
                if(p.getChildCount() > 1){
                    uniTree = false;
                    break;
                }
                try{
                    p = (ParserRuleContext)p.getChild(0);
                }
                catch(ClassCastException e){
                    break;
                }
            }

            if(uniTree){
                p = ctx.getParent();
                while(!p.getClass().getSimpleName().
                   equals("StatementContext"))
                     p = p.getParent();
                String temp = p.getText();
                p.removeLastChild();
                p.addChild(new TerminalNodeImpl(new
                   CommonToken(0, "for(int m=0; m <
                   MERGE_LEVEL; m++)")));
                p.addChild(new TerminalNodeImpl(new
                   CommonToken(0, temp)));
                GlobalValues.
                   setEnteredExpressionStatement(false);
            }

        }
    }

    Override public void exitTypeSpecifier(NotNull C_RCUDAParser
       .TypeSpecifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
            " "));
    }

    Override public void exitExpressionStatement(NotNull
       C_RCUDAParser.ExpressionStatementContext ctx) {
        if(GlobalValues.isEnteredExpressionStatement()){
                StatementContext p = (StatementContext)
                   ctx.getParent();
                String child = p.getChild(0).getText();
                p.removeLastChild();
                p.addChild(new TerminalNodeImpl(new
                   CommonToken(0, "for(int m=0;m<
                   MERGE_LEVEL;m++)")));
                p.addChild(new TerminalNodeImpl(new
                   CommonToken(1, child)));
                GlobalValues.
                   setEnteredExpressionStatement(false);
        }
    }

    Override public void enterDirectDeclarator(NotNull
       C_RCUDAParser.DirectDeclaratorContext ctx) {
        boolean lValueFound = false;
        ArrayList<String> lValueList = GlobalValues.
           getBlockMergeIndexLValueIdentifier();
        for(int i=0; i < lValueList.size(); i++)
```

```
                if(ctx.getText().equals(lValueList.get(i)))
                    lValueFound = true;
            if(lValueFound){
                String lvalue = ctx.getText();
                ParserRuleContext p = ctx.getParent();
                while(!p.getClass().getSimpleName().equals("
                   InitDeclaratorContext"))
                    p = p.getParent();
                int childs = p.getChildCount();
                for(int i=0; i < childs; i++){
                    p.removeLastChild();
                }
                p.addChild(new TerminalNodeImpl(new
                   CommonToken(0, lvalue+"[MERGE_LEVEL]")));
                for(int i=1; i < childs; i++)
                    p.addChild(new TerminalNodeImpl(new
                       CommonToken(i, "")));
            }
        }

        Override public void enterTypedefName(NotNull C_RCUDAParser.
           TypedefNameContext ctx) {
            boolean lValueFound = false;
            ArrayList<String> lValueList = GlobalValues.
               getBlockMergeIndexLValueIdentifier();
            for(int i=0; i < lValueList.size(); i++)
                if(ctx.getText().equals(lValueList.get(i)))
                    lValueFound = true;
            if(lValueFound){
                ctx.addChild(new TerminalNodeImpl(new
                   CommonToken(0, "[MERGE_LEVEL]")));
            }
        }
}
```

Listing 7: BlockSkewing.java

```
public class BlockSkewing extends C_RCUDABaseListener {
    Override public void exitTypeSpecifier(NotNull C_RCUDAParser
       .TypeSpecifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
           " "));
    }

    Override public void enterPrimaryExpression(NotNull
       C_RCUDAParser.PrimaryExpressionContext ctx){
        if(ctx.getText().equals(GlobalValues.
           getBlockSkewIndex())){
            ctx.removeLastChild();
            ctx.addChild(new CommonToken(0, "("+
               GlobalValues.getBlockSkewIndex()+"+n)"));
            GlobalValues.setEnteredExpressionStatement(
               true);
        }
        boolean lValueFound = false;
        ArrayList<String> lValueList = GlobalValues.
           getBlockSkewIndexLValueIdentifier();
        for(int i=0; i < lValueList.size(); i++)
            if(ctx.getText().equals(lValueList.get(i)))
                lValueFound = true;
        if(lValueFound){
            String lvalue = ctx.getText();
            ctx.removeLastChild();
            ctx.addChild(new TerminalNodeImpl(new
               CommonToken(0, lvalue+"[n]")));

            ParserRuleContext p = ctx.getParent();
            while(!(p.getClass().getSimpleName().equals("
               AssignmentExpressionContext") && p.
               getChildCount() == 3))
                p = p.getParent();
            boolean uniTree = true;
            p = (ParserRuleContext)p.getChild(2);

            while(p != null){
                if(p.getChildCount() > 1){
                    uniTree = false;
                    break;
                }
                try{
                    p = (ParserRuleContext)p.getChild(0);
                }
                catch(ClassCastException e){
                    break;
                }
            }

            if(uniTree){
                p = ctx.getParent();
                while(!p.getClass().getSimpleName().
                   equals("StatementContext"))
                    p = p.getParent();
                String temp = p.getText();
                p.removeLastChild();
                p.addChild(new TerminalNodeImpl(new
                   CommonToken(0, "for(int n=0; n <
                   SKEW_LEVEL; n++)")));
                p.addChild(new TerminalNodeImpl(new
                   CommonToken(0, temp)));
            }

        }
    }

    Override public void exitExpressionStatement(NotNull
       C_RCUDAParser.ExpressionStatementContext ctx) {
        if(GlobalValues.isEnteredExpressionStatement()){
                C_RCUDAParser.StatementContext p = (
                   C_RCUDAParser.StatementContext)ctx.
                   getParent();
                String child = p.getChild(0).getText();
                p.removeLastChild();
                p.addChild(new TerminalNodeImpl(new
                   CommonToken(0, "for(int n=0;n<
                   SKEW_LEVEL;n++)")));
                p.addChild(new TerminalNodeImpl(new
                   CommonToken(1, child)));
                GlobalValues.
                   setEnteredExpressionStatement(false);
        }
    }

    Override public void enterDirectDeclarator(NotNull
       C_RCUDAParser.DirectDeclaratorContext ctx) {
        boolean lValueFound = false;
        boolean sameMergeLValue = false;
        ArrayList<String> lValueList = GlobalValues.
           getBlockSkewIndexLValueIdentifier();
        for(int i=0; i < lValueList.size(); i++)
            if(ctx.getText().equals(lValueList.get(i)))
                lValueFound = true;
        ArrayList<String> mlValueList = GlobalValues.
           getBlockMergeIndexLValueIdentifier();
        for(int i=0; i < mlValueList.size(); i++)
            if(ctx.getText().equals(mlValueList.get(i)))
                sameMergeLValue = true;
        if(lValueFound){
            if(sameMergeLValue){
                ctx.addChild(new TerminalNodeImpl(new
                   CommonToken(0, "[SKEW_LEVEL]")));
            }
            else{
                String lvalue = ctx.getText();
                ParserRuleContext p = ctx.getParent();
                while(!p.getClass().getSimpleName().
                   equals("InitDeclaratorContext"))
                    p = p.getParent();
                int childs = p.getChildCount();
                for(int i=0; i < childs; i++){
                    p.removeLastChild();
                }
                p.addChild(new TerminalNodeImpl(new
                   CommonToken(0, lvalue+"[SKEW_LEVEL]"))
                   );
                for(int i=1; i < childs; i++)
                    p.addChild(new TerminalNodeImpl(new
                       CommonToken(i, "")));
            }
        }
    }

    Override public void enterTypedefName(NotNull C_RCUDAParser.
       TypedefNameContext ctx) {
        boolean lValueFound = false;
        ArrayList<String> lValueList = GlobalValues.
           getBlockSkewIndexLValueIdentifier();
        for(int i=0; i < lValueList.size(); i++)
            if(ctx.getText().equals(lValueList.get(i)))
                lValueFound = true;
        if(lValueFound){
                ctx.addChild(new TerminalNodeImpl(new
                   CommonToken(0, "[SKEW_LEVEL]")));
        }
    }
}
```

Listing 8: PrefetchingPhase1.java

```
public class PrefetchingPhase1 extends
   C_RCUDABaseListener {
    Override public void exitTypeSpecifier(NotNull C_RCUDAParser
       .TypeSpecifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
           " "));
    }
```

```
Override public void exitFunctionDefinition(NotNull
   C_RCUDAParser.FunctionDefinitionContext ctx) {
    int splitIndex = ctx.getText().indexOf("{");

    String temp1 = ctx.getText().substring(0,
       splitIndex+1);
    String temp2 = ctx.getText().substring(splitIndex
       +1);

    for(int i=0; i < ctx.getChildCount(); i++)
        ctx.removeLastChild();

        ctx.removeLastChild();
    ctx.addChild(new TerminalNodeImpl(new CommonToken
       (0, temp1)));

    ArrayList<String> prefetchedIds = GlobalValues.
       getPrefetchedArrayIdentifiers();
    for(int i=0; i<prefetchedIds.size(); i++)
        ctx.addChild(new TerminalNodeImpl(new
           CommonToken(0, "__shared__ " +
           GlobalValues.getTypeSpecifier() + " " +
           prefetchedIds.get(i) + "s[MERGE_LEVEL][
           BLOCKSIZE];")));

    ctx.addChild(new TerminalNodeImpl(new CommonToken
       (0, temp2)));
}

    Override public void enterPrimaryExpression(NotNull
       C_RCUDAParser.PrimaryExpressionContext ctx) {
        if(GlobalValues.getPrefetchedArrayIdentifiers
           ().contains(ctx.getText())){
            ParserRuleContext p = ctx.getParent();
            while(p != null && !p.getClass().
               getSimpleName().equals("
               MultiplicativeExpressionContext"))
                p = p.getParent();

            String prefetchExp = p.getText().split("]
               ")[0] + "+tid]";

            GlobalValues.
               setPrefetchingArrayExpression(
               prefetchExp);
            GlobalValues.setPrefetchingLoopIndex(
               GlobalValues.getLastLoopIndex());
```

```
                p.removeLastChild();
                p.addChild(new TerminalNodeImpl(new
                   CommonToken(0, ctx.getText()+"s[m][t]"
                   )));
            }
            else if(GlobalValues.
               getNonPrefetchedArrayIdentifiers().
               contains(ctx.getText())){
                ParserRuleContext p = ctx.getParent();
                while(p != null && !p.getClass().
                   getSimpleName().equals("
                   UnaryExpressionContext"))
                    p = p.getParent();

                String nonPrefetchExp = p.getText();

                GlobalValues.
                   setNonPrefetchingArrayExpression(
                   nonPrefetchExp);
            }
        }

        Override public void enterIterationStatement(NotNull
           C_RCUDAParser.IterationStatementContext ctx) {
            String loopIndex = ctx.declaration().
               initDeclaratorList().initDeclarator().
               declarator().getText();
            if(!loopIndex.equals("m")){
                GlobalValues.setLastLoopIndex(loopIndex);
                GlobalValues.setPrefetchingLoopLineNumber
                   (ctx.start.getStartIndex());
            }
        }

        Override public void exitIterationStatement(NotNull
           C_RCUDAParser.IterationStatementContext ctx) {
            String loopIndex = ctx.declaration().
               initDeclaratorList().initDeclarator().
               declarator().getText();
            if(!loopIndex.equals("m")){
                GlobalValues.setLastLoopIndex("");
            }
        }
}
```

Listing 9: PrefetchingPhase2.java

```
public class PrefetchingPhase2 extends
   C_RCUDABaseListener {
    Override public void enterIterationStatement(NotNull
       C_RCUDAParser.IterationStatementContext ctx) {
        String loopDecVar = ctx.declaration().
           initDeclaratorList().initDeclarator().
           declarator().getText();
        if(loopDecVar.equals(GlobalValues.
           getPrefetchingLoopIndex())){
            System.out.println("prefetching loop found");
            System.out.println(ctx.getText());
            String prefetching_loop = "for(int m=0;m<
               MERGE_LEVEL;m++){";
            prefetching_loop += GlobalValues.
               getPrefetchedArrayIdentifiers().get(0) + "
               s[m][tid]=" + GlobalValues.
               getPrefetchingArrayExpression() + ";";
            prefetching_loop += "}__syncthreads();";
            String loop = "int " + loopDecVar + ";" +
               prefetching_loop;
            loop += "for(" + loopDecVar + "=0;" +
               loopDecVar + "<w-BLOCKSIZE;" + loopDecVar
               + "+=BLOCKSIZE){";
            String tloop = "for(int t=0; t < BLOCKSIZE; t
               ++)";
            RuleContext loopbodyctx = (RuleContext)ctx.
               getChild(ctx.getChildCount()-1).getPayload
               ();
            ParserRuleContext p = ctx.getParent();
            p.removeLastChild();
            p.addChild(new TerminalNodeImpl(new
               CommonToken(0, loop+tloop)));
            p.addChild(loopbodyctx);
            p.addChild(new TerminalNodeImpl(new
               CommonToken(0, "__syncthreads();")));
            p.addChild(new TerminalNodeImpl(new
               CommonToken(0, prefetching_loop)));
            p.addChild(new TerminalNodeImpl(new
               CommonToken(0, "}")));
            p.addChild(new TerminalNodeImpl(new
               CommonToken(0, tloop)));
            p.addChild(loopbodyctx);
        }
    }
    Override public void exitTypeSpecifier(NotNull C_RCUDAParser
       .TypeSpecifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
            " "));
    }
}
```

Listing 10: PrefetchingPhase3.java

```
public class PrefetchingPhase3 extends
   C_RCUDABaseListener {
    Override public void exitTypeSpecifier(NotNull C_RCUDAParser
       .TypeSpecifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
            " "));
    }

    Override public void enterIterationStatement(NotNull
       C_RCUDAParser.IterationStatementContext ctx) {
        String loopIndex;
        if(ctx.declaration() != null)
            loopIndex = ctx.declaration().
               initDeclaratorList().initDeclarator().
               declarator().getText();
        else{
            String assignExp = ctx.expression(0).getText
               ();
            String assignOp = ctx.expression(0).
               assignmentExpression().assignmentOperator
               ().getText();
            loopIndex = assignExp.substring(0, assignExp.
               indexOf(assignOp));
        }
        if(loopIndex.equals(GlobalValues.
           getPrefetchingLoopIndex())){
            GlobalValues.setPrefetechedLoopEntered(true);
        }
    }

    Override public void exitIterationStatement(NotNull
       C_RCUDAParser.IterationStatementContext ctx) {
        String loopIndex;
        if(ctx.declaration() != null)
            loopIndex = ctx.declaration().
               initDeclaratorList().initDeclarator().
               declarator().getText();
        else{
            String assignExp = ctx.expression(0).getText
                ();
            String assignOp = ctx.expression(0).
                assignmentExpression().assignmentOperator
                ().getText();
            loopIndex = assignExp.substring(0, assignExp.
                indexOf(assignOp));
        }
        if(loopIndex.equals(GlobalValues.
           getPrefetchingLoopIndex())){
            GlobalValues.setPrefetechedLoopEntered(false)
                ;
        }
    }

        Override public void
           enterMultiplicativeExpression(NotNull C_RCUDAParser
           .MultiplicativeExpressionContext ctx) {
            if(ctx.getText().equals(GlobalValues.
                getPrefetchingArrayExpression()))
                 GlobalValues.
                     setPrefetchedArrayExpressionFound(true
                     );
        }

        Override public void
           exitMultiplicativeExpression(NotNull C_RCUDAParser.
           MultiplicativeExpressionContext ctx) {
            if(ctx.getText().equals(GlobalValues.
                getPrefetchingArrayExpression()))
                 GlobalValues.
                     setPrefetchedArrayExpressionFound(
                     false);
        }

        Override public void enterUnaryExpression(NotNull
           C_RCUDAParser.UnaryExpressionContext ctx) {
            if(ctx.getText().equals(GlobalValues.
                getNonPrefetchingArrayExpression()))
                 GlobalValues.
                     setNonPrefetchedArrayExpressionFound(
                     true);
        }

        Override public void exitUnaryExpression(NotNull
           C_RCUDAParser.UnaryExpressionContext ctx) {
            if(ctx.getText().equals(GlobalValues.
                getNonPrefetchingArrayExpression()))
                 GlobalValues.
                    setNonPrefetchedArrayExpressionFound(
                    false);
        }

    Override public void enterPrimaryExpression(NotNull
       C_RCUDAParser.PrimaryExpressionContext ctx) {
        if(GlobalValues.isPrefetechedLoopEntered() &&
           GlobalValues.isPrefetchedArrayExpressionFound
           () && ctx.getText().equals(GlobalValues.
           getPrefetchingLoopIndex())){
            ctx.removeLastChild();
            ctx.addChild(new TerminalNodeImpl(new
               CommonToken(0, GlobalValues.
               getPrefetchingLoopIndex()+"+BLOCKSIZE")));
        }
        else if(/*GlobalValues.isPrefetechedLoopEntered()
            &&*/ GlobalValues.
           isNonPrefetchedArrayExpressionFound() && ctx.
           getText().equals(GlobalValues.
           getPrefetchingLoopIndex())){
            ctx.removeLastChild();
            ctx.addChild(new TerminalNodeImpl(new
               CommonToken(0, GlobalValues.
               getPrefetchingLoopIndex()+"+t")));
        }
    }
}
```

Listing 11: RemoveRedundantArrayAccess.java

```
public class RemoveRedundantArrayAccess extends
   C_RCUDABaseListener {
    Override public void exitTypeSpecifier(NotNull C_RCUDAParser
       .TypeSpecifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
            " "));
    }

    Override public void enterIterationStatement(NotNull
       C_RCUDAParser.IterationStatementContext ctx) {
        String loopDecVar = ctx.declaration().
           initDeclaratorList().initDeclarator().
           declarator().getText();
        GlobalValues.setLastLoopIndex(loopDecVar);
    }

    Override public void exitIterationStatement(NotNull
       C_RCUDAParser.IterationStatementContext ctx) {
        GlobalValues.setLastLoopIndex("");
    }
}
```

Listing 12: ApplyRestrictClause.java

```
public class ApplyRestrictClause extends rcudaantlrparser
   .C_RCUDABaseListener {
        Override public void
           exitParameterDeclaration(NotNull C_RCUDAParser.
           ParameterDeclarationContext ctx) {
            if(ctx.getText().contains("restrict")){
                String temp = ctx.getText();
                for(int i=0; i < ctx.getChildCount(); i
                   ++)
                    ctx.removeLastChild();
                ctx.removeLastChild();
                String resstr = "restrict";
                temp = temp.substring(0, temp.indexOf(
                   resstr)-1) + " const* __restrict__ " +
                    temp.substring(temp.indexOf(resstr)+
                   resstr.length());
                ctx.addChild(new CommonToken(0, temp));
            }
        }

    Override public void exitTypeSpecifier(NotNull C_RCUDAParser
       .TypeSpecifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
            " "));
    }
}
```

Listing 13: CodeFormatter.java

```
public class CodeFormatter extends C_RCUDABaseListener {
        Override public void
           exitExpressionStatement(NotNull C_RCUDAParser.
           ExpressionStatementContext ctx) {
            String text = "\n";
            for(int i=0; i<GlobalValues.getTabCount(); i
               ++)
                text += "\t";
            ctx.addChild(new TerminalNodeImpl(new
               CommonToken(0, text)));
        }

        Override public void exitBlockItem(NotNull
           C_RCUDAParser.BlockItemContext ctx) {
            String text = "\n";
            for(int i=0; i<GlobalValues.getTabCount(); i
               ++)
                text += "\t";
            ctx.addChild(new TerminalNodeImpl(new
               CommonToken(0, text)));
        }

    Override public void exitTypeSpecifier(NotNull C_RCUDAParser
       .TypeSpecifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
           " "));
    }

    Override public void exitTypeQualifier(NotNull C_RCUDAParser
       .TypeQualifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
           " "));
    }

    Override public void
       visitTerminal(NotNull TerminalNode node) {
        String text = node.getText();
        if(text.equals("{"))
            GlobalValues.incTabCount();
        if(text.equals("}"))
            GlobalValues.decTabCount();
    }

        Override public void enterIterationStatement(NotNull
           C_RCUDAParser.IterationStatementContext ctx) {
            GlobalValues.setIterationStatment(true);
            GlobalValues.incTabCount();
            RuleContext temp = (RuleContext)ctx.getChild(
               ctx.getChildCount()-1).getPayload();
            ctx.removeLastChild();
            String text = "\n";
            for(int i=0; i<GlobalValues.getTabCount(); i
               ++)
```

```
                text += "\t";
            ctx.addChild(new TerminalNodeImpl(new
               CommonToken(0, text)));
            ctx.addChild(temp);
        }

        Override public void exitIterationStatement(NotNull
           C_RCUDAParser.IterationStatementContext ctx) {
            GlobalValues.setIterationStatment(false);
            GlobalValues.decTabCount();
        }

        Override public void enterStatement(NotNull
           C_RCUDAParser.StatementContext ctx) {
            GlobalValues.setIterationStatment(false);
        }
}
```

Listing 14: MainFile.java

```
public class MainFile extends C_RCUDABaseListener {
        Override public void
           enterAssignmentExpression(NotNull C_RCUDAParser.
           AssignmentExpressionContext ctx) {
            if(ctx.getText().contains("malloc")){
                String sttext = ctx.getText();
                String lvalue = sttext.substring(0,
                   sttext.indexOf("=")).trim();
                String malloc_str = sttext.substring(
                   sttext.indexOf("malloc")+6);

                for(int i=0; i < ctx.getChildCount(); i
                   ++)
                    ctx.removeLastChild();
                ctx.removeLastChild();
                ctx.addChild(new TerminalNodeImpl(new
                   CommonToken(0, "cudaMallocManaged(&" +
                    lvalue + "," + malloc_str.substring(
                   malloc_str.indexOf("(")+1))));
            }
        }

        Override public void exitTypedefName(NotNull
           C_RCUDAParser.TypedefNameContext ctx) {
            if(ctx.getText().equals("free")){
                ctx.removeLastChild();
                ctx.addChild(new TerminalNodeImpl(new
                    CommonToken(0, "cudaFree")));
            }
        }

Override public void exitTypeSpecifier(NotNull C_RCUDAParser
    .TypeSpecifierContext ctx){
        ctx.addChild(new CommonToken(ctx.getChildCount(),
            " "));
}

        Override public void enterPrimaryExpression(NotNull
            C_RCUDAParser.PrimaryExpressionContext ctx) {
             if(GlobalValues.getKernelNameList().contains(
                 ctx.getText())){
                 String kname = ctx.getText();
                 ctx.removeLastChild();
                 if(GlobalValues.is2DMatrix())
                      ctx.addChild(new TerminalNodeImpl(new
                           CommonToken(0, "dim3 threads(
                          BLOCKSIZE,1);dim3 grid(N*N/
                          BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL
                          ,1);")));
                 else
                      ctx.addChild(new TerminalNodeImpl(new
                           CommonToken(0, "dim3 threads(
                          BLOCKSIZE,1);dim3 grid(N/BLOCKSIZE
                          /MERGE_LEVEL/SKEW_LEVEL,1);")));
                 ctx.addChild(new TerminalNodeImpl(new
                     CommonToken(0, kname+"<<<grid, threads
                     >>>")));

                 ParserRuleContext p = ctx.getParent();
                 while(!(p.getClass().getSimpleName().
                     equals("StatementContext"))){
                      p = p.getParent();
                      if(p == null) return;
                 }

                 p.addChild(new TerminalNodeImpl(new
                     CommonToken(0, "cudaDeviceSynchronize
                     ();")));

            }
        }

        Override public void
            exitExpressionStatement(NotNull C_RCUDAParser.
            ExpressionStatementContext ctx) {
             if(ctx.getText().startsWith("exit(")){
                 for(int i=0; i < ctx.getChildCount(); i
                    ++)
                     ctx.removeLastChild();
                 ctx.removeLastChild();
                 ctx.addChild(new TerminalNodeImpl(new
                    CommonToken(0, "cudaThreadExit();")));
             }
        }
}
```

Listing 15: RCUDATranslator.java

```
public class RCUDATranslator {
    public static void main(String[] args) {
        GlobalValues.setLoopCount(0);
        try{
            FileReader fr = new FileReader(System.
               getProperty("user.dir") + "/src/config.txt
               ");
            BufferedReader br = new BufferedReader(fr);

            String temp;
            while((temp = br.readLine()) != null){
                String lvalue = temp.substring(0, temp.
                   indexOf("="));
                String rvalue = temp.substring(temp.
                   indexOf("=")+1);
                switch(lvalue){
                    case "LOOP_COLLAPSING":
                        if(rvalue.equals("1"))
                            GlobalValues.
                               setApplyLoopCollapsing(
                               true);
                        else
                            GlobalValues.
                               setApplyLoopCollapsing(
                               false);
                        break;
                    case "BLOCK_SKEW":
                        if(rvalue.equals("1"))
                            GlobalValues.
                               setApplyBlockSkew(true);
                        else
                            GlobalValues.
                               setApplyBlockSkew(false);
                        break;
                    case "PREFETCHING":
                        if(rvalue.equals("1"))
                            GlobalValues.
                               setApplyPrefetching(true);
                        else
                            GlobalValues.
                               setApplyPrefetching(false)
                               ;
                        break;
                    case "PREFETCHED_ARRAYS":
                        String pstr[] = rvalue.split(",")
                           ;
                        for(int i=0; i < pstr.length; i
                           ++)
                            GlobalValues.
                               setPrefetchedArrayIdentifiers
                               (pstr[i]);
                        break;
                    case "NON_PREFETCHED_ARRAYS":
                        String npstr[] = rvalue.split(","
                           );
                        for(int i=0; i < npstr.length; i
                           ++)
                            GlobalValues.
                               setNonPrefetchedArrayIdentifiers
                               (npstr[i]);
                        break;
                    case "DATA_TYPE":
                        GlobalValues.setTypeSpecifier(
                           rvalue);
                        break;
                    case "KERNEL_NAMES":
                        String kstr[] = rvalue.split(",")
                           ;
                        for(int i=0; i < kstr.length; i
                           ++)
                            GlobalValues.
                               setKernelNameList(kstr[i])
                               ;
                        break;
                    case "2DMATRIX":
                        if(rvalue.equals("1"))
                            GlobalValues.set2DMatrix(true
                               );
                        else
```

```
                    GlobalValues.set2DMatrix(
                       false);
                break;
            case "ROW_DIM":
                GlobalValues.setRowDim(rvalue);
        }
    }

    br.close();
    fr.close();

        ANTLRInputStream input = new
           ANTLRFileStream(System.getProperty("
           user.dir") + "/src/kernel.c");
        C_RCUDALexer lexer = new C_RCUDALexer(
           input);
        CommonTokenStream tokens = new
           CommonTokenStream(lexer);
        C_RCUDAParser parser = new C_RCUDAParser(
           tokens);
        parser.setBuildParseTree(true);
        ParserRuleContext tree = parser.
           functionDefinition();
        String output = "";

        //Applying Loop Collapsing
        if(GlobalValues.isApplyLoopCollapsing()){
            ParseTreeWalker.DEFAULT.walk(new
               LoopCollapsingListener(), tree);
            output = tree.getText();
            if(GlobalValues.getLoopCount() == 0
               && GlobalValues.isNestLoopFound())
               {
                System.out.println("Loop
                   collapsing applied");
            }
            System.out.println(output);

            //Applying Array Transformations 2D
               -> 1D
            input = new ANTLRInputStream(new
               ByteArrayInputStream(output.
               getBytes()));
            lexer = new C_RCUDALexer(input);
            tokens = new CommonTokenStream(lexer)
               ;
            parser = new C_RCUDAParser(tokens);
            parser.setBuildParseTree(true);
            tree = parser.functionDefinition();
        }

        ParseTreeWalker.DEFAULT.walk(new
           ArrayTransformationListener(), tree);
        output = tree.getText();
        System.out.println("Array transformation
           applied");
        System.out.println(output);

        //Applying Loop Partitioning (Naive CUDA
           Kernel)
        input = new ANTLRInputStream(new
           ByteArrayInputStream(output.getBytes()
           ));
        lexer = new C_RCUDALexer(input);
        tokens = new CommonTokenStream(lexer);
        parser = new C_RCUDAParser(tokens);
        parser.setBuildParseTree(true);
        tree = parser.functionDefinition();

        ParseTreeWalker.DEFAULT.walk(new
           LoopPartitioning(), tree);
        output = tree.getText();
        System.out.println("Loop partitioning
           applied");
        System.out.println(output);

        //Applying Block Merging
        input = new ANTLRInputStream(new
           ByteArrayInputStream(output.getBytes()
           ));
        lexer = new C_RCUDALexer(input);
        tokens = new CommonTokenStream(lexer);
        parser = new C_RCUDAParser(tokens);
        parser.setBuildParseTree(true);
        tree = parser.functionDefinition();


        ParseTreeWalker.DEFAULT.walk(new
           BlockMerging(), tree);
        output = tree.getText();
        System.out.println("Block merging applied
           ");
        System.out.println(output);
```

```
//Applying Block Skewing
if(GlobalValues.isApplyBlockSkew()){
    input = new ANTLRInputStream(new
       ByteArrayInputStream(output.
       getBytes()));
    lexer = new C_RCUDALexer(input);
    tokens = new CommonTokenStream(lexer)
       ;
    parser = new C_RCUDAParser(tokens);
    parser.setBuildParseTree(true);
    tree = parser.functionDefinition();

    ParseTreeWalker.DEFAULT.walk(new
       BlockSkewing(), tree);
    output = tree.getText();
    System.out.println("Block skewing
       applied");
    System.out.println(output);
}

//Applying Prefetching
if(GlobalValues.isApplyPrefetching()){
    input = new ANTLRInputStream(new
       ByteArrayInputStream(output.
       getBytes()));
    lexer = new C_RCUDALexer(input);
    tokens = new CommonTokenStream(lexer)
       ;
    parser = new C_RCUDAParser(tokens);
    parser.setBuildParseTree(true);
    tree = parser.functionDefinition();

    ParseTreeWalker.DEFAULT.walk(new
       PrefetchingPhase1(), tree);
    output = tree.getText();
    System.out.println("Prefetching phase
        1 applied");
    System.out.println(GlobalValues.
       getPrefetchingArrayExpression());
    System.out.println(GlobalValues.
       getPrefetchingLoopIndex());
    System.out.println(output);


    input = new ANTLRInputStream(new
       ByteArrayInputStream(output.
       getBytes()));
```

```
            lexer = new C_RCUDALexer(input);
            tokens = new CommonTokenStream(lexer)
               ;
            parser = new C_RCUDAParser(tokens);
            parser.setBuildParseTree(true);
            tree = parser.functionDefinition();

            ParseTreeWalker.DEFAULT.walk(new
               PrefetchingPhase2(), tree);
            output = tree.getText();
            System.out.println("Prefetching phase
                2 applied");
            System.out.println(output);

            input = new ANTLRInputStream(new
               ByteArrayInputStream(output.
               getBytes()));
            lexer = new C_RCUDALexer(input);
            tokens = new CommonTokenStream(lexer)
               ;
            parser = new C_RCUDAParser(tokens);
            parser.setBuildParseTree(true);
            tree = parser.functionDefinition();

            ParseTreeWalker.DEFAULT.walk(new
               PrefetchingPhase3(), tree);
            output = tree.getText();
            System.out.println("Prefetching phase
                3 applied");
            System.out.println(output);

        }

        //Apply Restict Clause for Data Cache
        input = new ANTLRInputStream(new
           ByteArrayInputStream(output.getBytes()
           ));
        lexer = new C_RCUDALexer(input);
        tokens = new CommonTokenStream(lexer);
        parser = new C_RCUDAParser(tokens);
        parser.setBuildParseTree(true);
        tree = parser.functionDefinition();

        ParseTreeWalker.DEFAULT.walk(new
           ApplyRestrictClause(), tree);
        output = tree.getText();
        System.out.println("Restict clause
           applied");
        System.out.println(output);

        //Formatting Kernel File Output
        input = new ANTLRInputStream(new
           ByteArrayInputStream(output.getBytes()
           ));
        lexer = new C_RCUDALexer(input);
        tokens = new CommonTokenStream(lexer);
        parser = new C_RCUDAParser(tokens);
        parser.setBuildParseTree(true);
        tree = parser.functionDefinition();

        ParseTreeWalker.DEFAULT.walk(new
           CodeFormatter(), tree);
        output = tree.getText();

        int i=0;
        i = output.indexOf("{", i)+1;
        String tabs = "";
        while(i > 0){
            tabs += "\t";
            output = output.substring(0, i) + "\n
               " + tabs + output.substring(i);
            i = output.indexOf("{", i)+1;
        }

        i=0;
        i = output.indexOf("}", i);
        while(i > 0){
            output = output.substring(0, i-1) +
               output.substring(i);
            i = output.indexOf("}", i+1);
        }

        System.out.println("Kernel File Formatted
           ");
        System.out.println(output);

        File output_dir = new File(System.
           getProperty("user.dir") + "/output");
        if(!output_dir.exists())
            output_dir.mkdir();

        PrintWriter kfile = new PrintWriter(
           System.getProperty("user.dir") + "/
           output/kernel.cu");
```

```
kfile.println(output);
kfile.close();

//Creating Main File
input = new ANTLRFileStream(System.
   getProperty("user.dir") + "/src/main.c
   ");
lexer = new C_RCUDALexer(input);
tokens = new CommonTokenStream(lexer);
parser = new C_RCUDAParser(tokens);
parser.setBuildParseTree(true);
tree = parser.functionDefinition();

ParseTreeWalker.DEFAULT.walk(new MainFile
   (), tree);
output = tree.getText();
System.out.println("Main File Created");
System.out.println(output);

//Formatting Kernel File Output
input = new ANTLRInputStream(new
   ByteArrayInputStream(output.getBytes()
   ));
lexer = new C_RCUDALexer(input);
tokens = new CommonTokenStream(lexer);
parser = new C_RCUDAParser(tokens);
parser.setBuildParseTree(true);
tree = parser.functionDefinition();

ParseTreeWalker.DEFAULT.walk(new
   CodeFormatter(), tree);
output = tree.getText();
i=0;
i = output.indexOf("{", i)+1;
tabs = "";
while(i > 0){
    tabs += "\t";
    output = output.substring(0, i) + "\n
       " + tabs + output.substring(i);
    i = output.indexOf("{", i)+1;
}

i=0;
i = output.indexOf("}", i);
while(i > 0){
    output = output.substring(0, i-1) +
       output.substring(i);
    i = output.indexOf("}", i+1);
}
System.out.println("Main File Formatted")
   ;
System.out.println(output);

PrintWriter mainfile = new PrintWriter(
   System.getProperty("user.dir") + "/
   output/main.cu");
mainfile.println("#include <stdlib.h>");
mainfile.println("#include <stdio.h>");
mainfile.println("#include <string.h>");
mainfile.println("#include <math.h>");
mainfile.println("#include <time.h>");
mainfile.println("#include <cuda.h>");
mainfile.println("void checkCudaError(
   const char *msg)\n" +
                 "{\n" +
                 "        cudaError_t
                    err =
                    cudaGetLastError()
                    ;\n" +
                 "        if(
                    cudaSuccess != err
                    ){\n" +
                 "
                    printf(\"%s(%i) :
                    CUDA error : %s :
                    (%d) %s\\n\",
                    __FILE__, __LINE__
                    , msg, (int)err,
                    cudaGetErrorString
                    (err));\n" +
                 "                exit
                     (-1);\n" +
                 "        }\n" +
               "}");
mainfile.println("#include \"params.h\"")
   ;
mainfile.println("#include \"rcudacublas.
   h\"");
mainfile.println("#include \"kernel.cu\""
   );
mainfile.println(output);
mainfile.close();

PrintWriter paramfile = new PrintWriter(
   System.getProperty("user.dir") + "/
   output/params.h");
paramfile.println("#define BLOCKSIZE 32")
   ;
paramfile.println("#define MERGE_LEVEL 1"
   );
paramfile.println("#define SKEW_LEVEL 1")
   ;
paramfile.close();

Path headerpath = FileSystems.getDefault
   ().getPath(System.getProperty("user.
   dir") + "/output/rcudacublas.h");
InputStream in = RCUDATranslator.class.
   getClassLoader().getResourceAsStream("
   rcudaapi/rcudacublas.h");
Files.copy(in, headerpath,
   StandardCopyOption.REPLACE_EXISTING);

headerpath = FileSystems.getDefault().
   getPath(System.getProperty("user.dir")
    + "/output/Makefile");
in = RCUDATranslator.class.getClassLoader
   ().getResourceAsStream("rcudaapi/
   Makefile");
Files.copy(in, headerpath,
   StandardCopyOption.REPLACE_EXISTING);

headerpath = FileSystems.getDefault().
   getPath(System.getProperty("user.dir")
    + "/output/findcudalib.mk");
in = RCUDATranslator.class.getClassLoader
   ().getResourceAsStream("rcudaapi/
   findcudalib.mk");
Files.copy(in, headerpath,
   StandardCopyOption.REPLACE_EXISTING);

System.out.println("RCUDA Parameter
   Tuning");
Optimizer opt = new Optimizer();
ArrayList<CalculatedValues> parameters;
int N = 1024;
int GPU = 0;
double cc = 3.5;
Runtime rt = Runtime.getRuntime();
String filespath = "./";
```

```
if(args.length >= 1)
    GPU = Integer.parseInt(args[0]);
if(args.length >= 2)
    filespath = args[1];

Scanner s = new Scanner(System.in);
System.out.print("Enter N: ");
N = s.nextInt();
System.out.print("Enter Compute
   Capability: ");
cc = s.nextDouble();
parameters = opt.findOptimalParameters(N,
    cc, filespath);
File f = new File(filespath + "params.h")
   ;
double min_time = 0.0;
String base_cmd="make -f output/Makefile"
   ;

for(int ii=0; ii < parameters.size(); ii
   ++){
    int bs = parameters.get(ii).
       getBlockSize();
    int ml = parameters.get(ii).
       getMergeLevel();
    int sl = parameters.get(ii).
       getSkewLevel();
    FileWriter w = new FileWriter(f);
    w.write("#define BLOCKSIZE " + bs + "
       \n#define MERGE_LEVEL " + ml + "\n
       #define SKEW_LEVEL " + sl + "\n");
    w.flush();
    w.close();

    System.out.println("Running config =
       (" + bs + "," + ml + "," + sl + ")
       ");

    String cmd = base_cmd;
    System.out.println(cmd);
    Process pr = rt.exec(cmd);

    BufferedReader inputt = new
       BufferedReader(new
       InputStreamReader(pr.
       getInputStream()));
```

```
        String line = null;

        while((line = inputt.readLine()) !=
           null)
            System.out.println(line);

        cmd = filespath + "main " + GPU;
        System.out.println(cmd);
        pr = rt.exec(cmd);

        inputt.close();
        inputt = null;

        inputt = new BufferedReader(new
           InputStreamReader(pr.
           getInputStream()));

        line = null;

        line = inputt.readLine();
        System.out.println("time = " + line);

        if(line == null)
            continue;

        if(Double.parseDouble(line) > 0 && (
           min_time == 0.0 || min_time >
           Double.parseDouble(line))){
            min_time = Double.parseDouble(
               line);
            System.out.println("min time = "
               + min_time);
            OptimalValues.setBlockSize(bs);
            OptimalValues.setMergeLevel(ml);
            OptimalValues.setSkewLevel(sl);
        }
    }

    System.out.println("Optimal Block Size =
       " + OptimalValues.getBlockSize());
    System.out.println("Optimal Merge Level =
        " + OptimalValues.getMergeLevel());
    System.out.println("Optimal Skew Level =
       " + OptimalValues.getSkewLevel());

    FileWriter w = new FileWriter(f);
    w.write("#define BLOCKSIZE " +
                            OptimalValues.getBlockSize() + "\n#
                            define MERGE_LEVEL " + OptimalValues.
                            getMergeLevel() + "\n#define
                            SKEW_LEVEL " + OptimalValues.
                            getSkewLevel() + "\n");
                        w.flush();
                        w.close();
            }
            catch(IOException e){
                System.out.println(e.getMessage());
            }
        }
}
```

Appendix C

RT-CUDA Parameter Tuning

Listing 16: CalculatedValues.java

```
public class CalculatedValues {
    private int BlockSize;
    private int MergeLevel;
    private int SkewLevel;
    private int ActiveBlocksByWarps;
    private int ActiveBlocksBySharedMemory;
    private int ActiveBlocksByRegisters;

    public CalculatedValues(int bs, int ml, int sl, int
       abw, int abs, int abr) {
        BlockSize = bs;
        MergeLevel = ml;
        SkewLevel = sl;
        ActiveBlocksByWarps = abw;
        ActiveBlocksBySharedMemory = abs;
        ActiveBlocksByRegisters = abr;
    }

    public int getBlockSize() {
        return BlockSize;
    }

    public int getMergeLevel() {
        return MergeLevel;
    }

    public int getSkewLevel() {
        return SkewLevel;
    }

    public int getActiveBlocksByWarps() {
        return ActiveBlocksByWarps;
    }

    public int getActiveBlocksBySharedMemory() {
        return ActiveBlocksBySharedMemory;
    }

    public int getActiveBlocksByRegisters() {
        return ActiveBlocksByRegisters;
    }
}
```

Listing 17: GPUData.java

```
public class GPUData {
    ArrayList<GPUDataParameters> gparameters;
    GPUDataParameters param;

    public GPUData() {
        gparameters = new ArrayList<GPUDataParameters>();
        param = new GPUDataParameters();
        param.setComputeCapability(1.0);
        param.setSM_Version("sm_10");
        param.setThreadPerWarp(32);
        param.setWarpsPerSM(24);
        param.setThreadsPerSM(768);
        param.setThreadBlocksPerSM(8);
        param.setMaxSharedMemoryPerSM(16384);
        param.setRegisterFileSize(8192);
        param.setRegisterAllocationUnitSize(256);
        param.setAllocationGranularity("block");
        param.setMaxRegistersPerThread(124);
        param.setSharedMemoryAllocationUnitSize(512);
        param.setWarpAllocationGranularity(2);
        param.setMaxThreadBlockSize(512);
        addParameters(param);
        param = new GPUDataParameters();
        param.setComputeCapability(1.1);
        param.setSM_Version("sm_11");
        param.setThreadPerWarp(32);
        param.setWarpsPerSM(24);
        param.setThreadsPerSM(768);
        param.setThreadBlocksPerSM(8);
        param.setMaxSharedMemoryPerSM(16384);
        param.setRegisterFileSize(8192);
        param.setRegisterAllocationUnitSize(256);
        param.setAllocationGranularity("block");
        param.setMaxRegistersPerThread(124);
        param.setSharedMemoryAllocationUnitSize(512);
        param.setWarpAllocationGranularity(2);
        param.setMaxThreadBlockSize(512);
        addParameters(param);
        param = new GPUDataParameters();
        param.setComputeCapability(1.2);
        param.setSM_Version("sm_12");
        param.setThreadPerWarp(32);
        param.setWarpsPerSM(32);
        param.setThreadsPerSM(1024);
        param.setThreadBlocksPerSM(8);
        param.setMaxSharedMemoryPerSM(16384);
        param.setRegisterFileSize(16384);
        param.setRegisterAllocationUnitSize(512);
        param.setAllocationGranularity("block");
        param.setMaxRegistersPerThread(124);
        param.setSharedMemoryAllocationUnitSize(512);
        param.setWarpAllocationGranularity(2);
        param.setMaxThreadBlockSize(512);
        addParameters(param);
        param = new GPUDataParameters();
        param.setComputeCapability(1.3);
        param.setSM_Version("sm_13");
        param.setThreadPerWarp(32);
        param.setWarpsPerSM(32);
        param.setThreadsPerSM(1024);
        param.setThreadBlocksPerSM(8);
        param.setMaxSharedMemoryPerSM(16384);
        param.setRegisterFileSize(16384);
        param.setRegisterAllocationUnitSize(512);
        param.setAllocationGranularity("block");
        param.setMaxRegistersPerThread(124);
        param.setSharedMemoryAllocationUnitSize(512);
        param.setWarpAllocationGranularity(2);
        param.setMaxThreadBlockSize(512);
        addParameters(param);
        param = new GPUDataParameters();
        param.setComputeCapability(2.0);
        param.setSM_Version("sm_20");
        param.setThreadPerWarp(32);
        param.setWarpsPerSM(48);
        param.setThreadsPerSM(1536);
        param.setThreadBlocksPerSM(8);
        param.setMaxSharedMemoryPerSM(49152);
        param.setRegisterFileSize(32768);
        param.setRegisterAllocationUnitSize(128);
        param.setAllocationGranularity("warp");
        param.setMaxRegistersPerThread(63);
        param.setSharedMemoryAllocationUnitSize(128);
        param.setWarpAllocationGranularity(2);
        param.setMaxThreadBlockSize(1024);
        addParameters(param);
        param = new GPUDataParameters();
        param.setComputeCapability(2.1);
        param.setSM_Version("sm_21");
        param.setThreadPerWarp(32);
        param.setWarpsPerSM(48);
        param.setThreadsPerSM(1536);
        param.setThreadBlocksPerSM(8);
        param.setMaxSharedMemoryPerSM(49152);
        param.setRegisterFileSize(32768);
        param.setRegisterAllocationUnitSize(128);
        param.setAllocationGranularity("warp");
        param.setMaxRegistersPerThread(63);
        param.setSharedMemoryAllocationUnitSize(128);
        param.setWarpAllocationGranularity(2);
        param.setMaxThreadBlockSize(1024);
        addParameters(param);
        param = new GPUDataParameters();
        param.setComputeCapability(3.0);
        param.setSM_Version("sm_30");
        param.setThreadPerWarp(32);
        param.setWarpsPerSM(64);
        param.setThreadsPerSM(2048);
        param.setThreadBlocksPerSM(16);
        param.setMaxSharedMemoryPerSM(49152);
        param.setRegisterFileSize(65536);
        param.setRegisterAllocationUnitSize(256);
        param.setAllocationGranularity("warp");
        param.setMaxRegistersPerThread(63);
        param.setSharedMemoryAllocationUnitSize(256);
        param.setWarpAllocationGranularity(4);
        param.setMaxThreadBlockSize(1024);
        addParameters(param);
        param = new GPUDataParameters();
        param.setComputeCapability(3.5);
        param.setSM_Version("sm_35");
        param.setThreadPerWarp(32);
        param.setWarpsPerSM(64);
        param.setThreadsPerSM(2048);
        param.setThreadBlocksPerSM(16);
        param.setMaxSharedMemoryPerSM(49152);
        param.setRegisterFileSize(65536);
        param.setRegisterAllocationUnitSize(256);
        param.setAllocationGranularity("warp");
        param.setMaxRegistersPerThread(255);
        param.setSharedMemoryAllocationUnitSize(256);
        param.setWarpAllocationGranularity(4);
        param.setMaxThreadBlockSize(1024);
        addParameters(param);
    }

    private void addParameters(GPUDataParameters p){
        gparameters.add(p);
    }

    public GPUDataParameters findParameters(double
       ComputeCapability){
        GPUDataParameters r = null;
        int i;
        for(i=0; i < gparameters.size(); i++){
            r = gparameters.get(i);
            if(r.getComputeCapability() ==
               ComputeCapability) break;
        }
        if(i == gparameters.size()){
            System.out.println("Compute Capability not
               found, returning default");
            r = gparameters.get(0);
        }
        return r;
    }
}
```

Listing 18: GPUDataParameters.java

```
public class GPUDataParameters {
    private double ComputeCapability;
    private String SM_Version;
    private int ThreadPerWarp;
    private int WarpsPerSM;
    private int ThreadsPerSM;
    private int ThreadBlocksPerSM;
    private int MaxSharedMemoryPerSM;
    private int RegisterFileSize;
    private int RegisterAllocationUnitSize;
    private String AllocationGranularity;
    private int MaxRegistersPerThread;
    private int SharedMemoryAllocationUnitSize;
    private int WarpAllocationGranularity;
    private int MaxThreadBlockSize;

    public double getComputeCapability() {
        return ComputeCapability;
    }

    public void setComputeCapability(double
       ComputeCapability) {
        this.ComputeCapability = ComputeCapability;
    }

    public String getSM_Version() {
        return SM_Version;
    }

    public void setSM_Version(String SM_Version) {
        this.SM_Version = SM_Version;
    }

    public int getThreadPerWarp() {
        return ThreadPerWarp;
    }

    public void setThreadPerWarp(int ThreadPerWarp) {
        this.ThreadPerWarp = ThreadPerWarp;
    }

    public int getWarpsPerSM() {
        return WarpsPerSM;
    }

    public void setWarpsPerSM(int WarpsPerSM) {
        this.WarpsPerSM = WarpsPerSM;
    }

    public int getThreadsPerSM() {
        return ThreadsPerSM;
    }

    public void setThreadsPerSM(int ThreadsPerSM) {
        this.ThreadsPerSM = ThreadsPerSM;
    }

    public int getThreadBlocksPerSM() {
        return ThreadBlocksPerSM;
    }

    public void setThreadBlocksPerSM(int
       ThreadBlocksPerSM) {
        this.ThreadBlocksPerSM = ThreadBlocksPerSM;
    }

    public int getMaxSharedMemoryPerSM() {
        return MaxSharedMemoryPerSM;
    }

    public void setMaxSharedMemoryPerSM(int
       MaxSharedMemoryPerSM) {
        this.MaxSharedMemoryPerSM = MaxSharedMemoryPerSM;
    }

    public int getRegisterFileSize() {
        return RegisterFileSize;
    }

    public void setRegisterFileSize(int RegisterFileSize)
        {
        this.RegisterFileSize = RegisterFileSize;
    }

    public int getRegisterAllocationUnitSize() {
        return RegisterAllocationUnitSize;
    }

    public void setRegisterAllocationUnitSize(int
       RegisterAllocationUnitSize) {
        this.RegisterAllocationUnitSize =
           RegisterAllocationUnitSize;
    }

    public String getAllocationGranularity() {
        return AllocationGranularity;
    }

    public void setAllocationGranularity(String
       AllocationGranularity) {
        this.AllocationGranularity =
           AllocationGranularity;
    }

    public int getMaxRegistersPerThread() {
        return MaxRegistersPerThread;
    }

    public void setMaxRegistersPerThread(int
       MaxRegistersPerThread) {
        this.MaxRegistersPerThread =
            MaxRegistersPerThread;
    }

    public int getSharedMemoryAllocationUnitSize() {
        return SharedMemoryAllocationUnitSize;
    }

    public void setSharedMemoryAllocationUnitSize(int
       SharedMemoryAllocationUnitSize) {
        this.SharedMemoryAllocationUnitSize =
           SharedMemoryAllocationUnitSize;
    }

    public int getWarpAllocationGranularity() {
        return WarpAllocationGranularity;
    }

    public void setWarpAllocationGranularity(int
       WarpAllocationGranularity) {
        this.WarpAllocationGranularity =
           WarpAllocationGranularity;
    }

    public int getMaxThreadBlockSize() {
        return MaxThreadBlockSize;
    }

    public void setMaxThreadBlockSize(int
       MaxThreadBlockSize) {
        this.MaxThreadBlockSize = MaxThreadBlockSize;
    }
}
```

Listing 19: OptimalValues.java

```
public class OptimalValues {

    private static int BlockSize = 0;
    private static int MergeLevel = 0;
    private static int SkewLevel = 0;

    public static int getMergeLevel() {
        return MergeLevel;
    }

    public static void setMergeLevel(int MergeLevel) {
        OptimalValues.MergeLevel = MergeLevel;
    }

    public static int getBlockSize() {
        return BlockSize;
    }

    public static void setBlockSize(int BlockSize) {
        OptimalValues.BlockSize = BlockSize;
    }

    public static int getSkewLevel() {
        return SkewLevel;
    }

    public static void setSkewLevel(int SkewLevel) {
        OptimalValues.SkewLevel = SkewLevel;
    }
}
```

Listing 20: Optimizer.java

```
public class Optimizer {
    private GPUData gdata;

    public Optimizer(){
        gdata = new GPUData();
    }

    public ArrayList<CalculatedValues> generateParameters
       (int N, double ComputeCapability, String path){
        ArrayList<CalculatedValues> cv = new ArrayList<
           CalculatedValues>();
        String base_cmd="";
        int maxBlockSize=0;
        int maxMergeLevel=0;
        int maxSkewLevel=0;
        int minBlockSize=32;
        int minMergeLevel=1;
        int minSkewLevel=1;
        try{
            base_cmd = "make -f " + path + "Makefile main
               .o";
            File fc = new File(path + "../src/config.txt"
               );
            BufferedReader reader = new BufferedReader(
                new FileReader(fc));

            String line=null;
            while((line=reader.readLine()) != null){
                String params[] = line.split("=");
                if(params.length >= 1){
                    if(params[0].contains("MAX_BLOCKSIZE"
                       ))
                        maxBlockSize = Integer.parseInt(
                           params[1]);
                    else if(params[0].contains("
                       MAX_MERGE_LEVEL"))
                        maxMergeLevel = Integer.parseInt(
                           params[1]);
                    else if(params[0].contains("
                       MAX_SKEW_LEVEL"))
                        maxSkewLevel = Integer.parseInt(
                           params[1]);
                    else if(params[0].contains("
                       MIN_BLOCKSIZE"))
                        minBlockSize = Integer.parseInt(
                           params[1]);
                    else if(params[0].contains("
                       MIN_MERGE_LEVEL"))
                        minMergeLevel = Integer.parseInt(
                           params[1]);
                    else if(params[0].contains("
                       MIN_SKEW_LEVEL"))
                        minSkewLevel = Integer.parseInt(
                           params[1]);
                }
            }
        }
        catch(IOException e){
            System.out.println(e.getMessage());
        }

        System.out.println("Max Block Size = " +
           maxBlockSize);
        System.out.println("Max Merge Level = " +
           maxMergeLevel);
        System.out.println("Max Skew Level = " +
           maxSkewLevel);
        System.out.println("Min Block Size = " +
           minBlockSize);
        System.out.println("Min Merge Level = " +
           minMergeLevel);
```

```
System.out.println("Min Skew Level = " +
   minSkewLevel);

if(maxBlockSize == 0)
    maxBlockSize = gdata.findParameters(
       ComputeCapability).getMaxThreadBlockSize()
       ;
for(int bs=minBlockSize; bs <= maxBlockSize; bs
   *=2){
    if(N % bs != 0)
        continue;
    int m_limit=bs;
    if(maxMergeLevel > 0)
        m_limit = maxMergeLevel;

    for(int ml=minMergeLevel; ml <= m_limit; ml
       *=2){
        int s_limit=bs;
        if(maxSkewLevel > 0)
            s_limit = maxSkewLevel;
        for(int sl=minSkewLevel; sl <= s_limit;
           sl*=2){
            int KernelBlocks = N/bs/ml/sl;
            if(KernelBlocks == 0)
                continue;
            try{
                File f = new File(path + "params.
                   h");
                FileWriter w = new FileWriter(f);
                w.write("#define BLOCKSIZE " + bs
                    + "\n#define MERGE_LEVEL " +
                   ml + "\n#define SKEW_LEVEL " +
                    sl + "\n");
                w.flush();
                w.close();
                int ShM = 1;
                int RPT = 1;

                System.out.println("Testing
                   config = (" + bs + "," + ml +
                   "," + sl + ")");

                String cmd = base_cmd;
                System.out.println(cmd);

                Runtime rt = Runtime.getRuntime()
                   ;
```

```
Process pr = rt.exec(cmd);
String line = null;
BufferedReader input1 = new
   BufferedReader(new
   InputStreamReader(pr.
   getInputStream()));

String ptx_info = "";
while((line = input1.readLine())
   != null){
    System.out.println(line);
    if(line.contains("ptxas"))
        ptx_info = line;
}

if(ptx_info.isEmpty()){
    BufferedReader input = new
       BufferedReader(new
       InputStreamReader(pr.
       getErrorStream()));
    while((line = input.readLine
       ()) != null){
        System.out.println(line);
        if(line.contains("ptxas")
           )
            ptx_info = line;
    }
}

String strs[] = ptx_info.split(":
   ");
ptx_info = strs[strs.length-1].
   trim();
strs = ptx_info.split(",");
String registersString = strs[0];
String sharedMemoryString = strs
   [1];
RPT = Integer.parseInt(
   registersString.split(" ")[1])
   ;
sharedMemoryString =
   sharedMemoryString.trim();
if(sharedMemoryString.contains("
   smem"))
    ShM = Integer.parseInt(
       sharedMemoryString.split("
        ")[0]);
```

```
                else
                    ShM = 1;

                int r = (RPT * bs) / gdata.
                   findParameters(
                   ComputeCapability).
                   getRegisterAllocationUnitSize
                   () + 1;
                int RPB = r * gdata.
                   findParameters(
                   ComputeCapability).
                   getRegisterAllocationUnitSize
                   ();
                int WarpsPerBlock = (int)Math.
                   ceil((double)bs / (double)
                   gdata.findParameters(
                   ComputeCapability).
                   getThreadPerWarp());
                int ActiveBlocksByWarps = (int)
                   Math.floor(gdata.
                   findParameters(
                   ComputeCapability).
                   getWarpsPerSM() /
                   WarpsPerBlock);
                int ActiveBlocksByShM = (int)Math
                   .floor(gdata.findParameters(
                   ComputeCapability).
                   getMaxSharedMemoryPerSM() /
                   ShM);
                int ActiveBlocksByRegisters = (
                   int)Math.floor(gdata.
                   findParameters(
                   ComputeCapability).
                   getRegisterFileSize() / RPB);

                cv.add(new CalculatedValues(bs,
                   ml, sl, ActiveBlocksByWarps,
                   ActiveBlocksByShM,
                   ActiveBlocksByRegisters));

            }
            catch(IOException e){System.out.
               println(e.getMessage());}

        }
    }
}
```

```
        return cv;
    }

    public ArrayList<CalculatedValues>
       findOptimalParameters(int N, double
       ComputeCapability, String path){
        ArrayList<CalculatedValues> parameterValues =
           generateParameters(N, ComputeCapability, path)
           ;
        ArrayList<CalculatedValues> optimalParameters =
           new ArrayList<CalculatedValues>();

        System.out.println("Generated Parameters Count =
           " + parameterValues.size());

        for(int i=0; i < parameterValues.size(); i++)
            if(parameterValues.get(i).
               getActiveBlocksByWarps() > 0 &&
               parameterValues.get(i).
               getActiveBlocksBySharedMemory() > 0 &&
               parameterValues.get(i).
               getActiveBlocksByRegisters() > 0)
                optimalParameters.add(parameterValues.get
                   (i));

        System.out.println("Optimal Parameters Count = "
           + optimalParameters.size());

        return optimalParameters;
    }
}
```

Appendix D
RT-CUDA API

Listing 21: rcudacublas.h

```
#include <cublas_v2.h>

void RTdSMM(float *C, const float *A, const float *B, int
    m, int n, int k){
    cublasStatus_t status;
    cublasHandle_t handle;
    status = cublasCreate(&handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdSMM error: the CUDA Runtime
               Initialization Failed");
            exit(1);
            break;
        case CUBLAS_STATUS_ALLOC_FAILED:
            printf("RtdSMM error: the resources could not
                be allocated");
            exit(1);
            break;
    }

    const float alpha = 1.0;
    const float beta = 0.0;
    status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N
       , m, n, k, &alpha, B, k, A, m, &beta, C, m);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdSMM error: the library was not
               initialized");
            exit(1);
            break;
        case CUBLAS_STATUS_INVALID_VALUE:
            printf("RTdSMM error: the parameters m,n,k<0"
               );
            exit(1);
            break;
        case CUBLAS_STATUS_ARCH_MISMATCH:
            printf("RTdSMM error: the device does not
               support double-precision");
            exit(1);
            break;
        case CUBLAS_STATUS_EXECUTION_FAILED:
            printf("RTdSMM error: the function failed to
               launch on the GPU");
            exit(1);
            break;
    }

    status = cublasDestroy(handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdSMM error: the library was not
               initialized");
            exit(1);
            break;
    }
}

void RTdDMM(double *C, const double *A, const double *B,
   int m, int n, int k){
    cublasStatus_t status;
    cublasHandle_t handle;
    status = cublasCreate(&handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdDMM error: the CUDA Runtime
               Initialization Failed");
            exit(1);
            break;
        case CUBLAS_STATUS_ALLOC_FAILED:
            printf("RtdDMM error: the resources could not
                be allocated");
            exit(1);
            break;
    }
    const double alpha = 1.0;
    const double beta = 0.0;
    status = cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N
       , m, n, k, &alpha, B, k, A, m, &beta, C, m);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdDMM error: the library was not
               initialized");
            exit(1);
            break;
        case CUBLAS_STATUS_INVALID_VALUE:
            printf("RTdDMM error: the parameters m,n,k<0"
               );
            exit(1);
            break;
        case CUBLAS_STATUS_ARCH_MISMATCH:
            printf("RTdDMM error: the device does not
               support double-precision");
            exit(1);
            break;
        case CUBLAS_STATUS_EXECUTION_FAILED:
            printf("RTdDMM error: the function failed to
               launch on the GPU");
            exit(1);
            break;
    }

    status = cublasDestroy(handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdDMM error: the library was not
               initialized");
            exit(1);
            break;
    }
}

void RTdSMV(float *C, const float *A, const float *B, int
    m, int n){
    cublasStatus_t status;
    cublasHandle_t handle;
    status = cublasCreate(&handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdSMV error: the CUDA Runtime
               Initialization Failed");
            exit(1);
            break;
        case CUBLAS_STATUS_ALLOC_FAILED:
            printf("RtdSMV error: the resources could not
                be allocated");
            exit(1);
            break;
    }

    const float alpha = 1.0;
    const float beta = 0.0;
    status = cublasSgemv(handle, CUBLAS_OP_T, m, n, &
       alpha, A, m, B, 1, &beta, C, 1);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdSMV error: the library was not
               initialized");
            exit(1);
            break;
        case CUBLAS_STATUS_INVALID_VALUE:
            printf("RTdSMV error: the parameters m,n<0 or
                incx,incy=0");
            exit(1);
            break;
        case CUBLAS_STATUS_ARCH_MISMATCH:
            printf("RTdSMV error: the device does not
               support double-precision");
            exit(1);
            break;
        case CUBLAS_STATUS_EXECUTION_FAILED:
            printf("RTdSMV error: the function failed to
               launch on the GPU");
            exit(1);
            break;
    }

    status = cublasDestroy(handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
                break;
            case CUBLAS_STATUS_NOT_INITIALIZED:
                printf("RTdSMV error: the library was not
                   initialized");
                exit(1);
                break;
        }
}

void RTdDMV(double *C, const double *A, const double *B,
    int m, int n){
        cublasStatus_t status;
        cublasHandle_t handle;
        status = cublasCreate(&handle);
        switch(status){
            case CUBLAS_STATUS_SUCCESS:
                break;
            case CUBLAS_STATUS_NOT_INITIALIZED:
                printf("RTdDMV error: the CUDA Runtime
                   Initialization Failed");
                exit(1);
                break;
            case CUBLAS_STATUS_ALLOC_FAILED:
                printf("RtdDMV error: the resources could not
                    be allocated");
                exit(1);
                break;
        }

        const double alpha = 1.0;
        const double beta = 0.0;
        status = cublasDgemv(handle, CUBLAS_OP_T, m, n, &
           alpha, A, m, B, 1, &beta, C, 1);
        switch(status){
            case CUBLAS_STATUS_SUCCESS:
                break;
            case CUBLAS_STATUS_NOT_INITIALIZED:
                printf("RTdDMV error: the library was not
                   initialized");
                exit(1);
                break;
            case CUBLAS_STATUS_INVALID_VALUE:
                printf("RTdDMV error: the parameters m,n<0 or
                    incx,incy=0");
                exit(1);
                break;
            case CUBLAS_STATUS_ARCH_MISMATCH:
```

```
            printf("RTdDMV error: the device does not
               support double-precision");
            exit(1);
            break;
        case CUBLAS_STATUS_EXECUTION_FAILED:
            printf("RTdDMV error: the function failed to
               launch on the GPU");
            exit(1);
            break;
    }

    status = cublasDestroy(handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdDMV error: the library was not
               initialized");
            exit(1);
            break;
    }
}

void RTdSMT(float *C, const float *A, int m, int n){
    cublasStatus_t status;
    cublasHandle_t handle;
    status = cublasCreate(&handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdSMT error: the CUDA Runtime
               Initialization Failed");
            exit(1);
            break;
        case CUBLAS_STATUS_ALLOC_FAILED:
            printf("RtdSMT error: the resources could not
                be allocated");
            exit(1);
            break;
    }

    const float alpha = 1.0;
    const float beta = 0.0;
    status = cublasSgeam(handle, CUBLAS_OP_T, CUBLAS_OP_N
       , m, n, &alpha, A, m, &beta, A, m, C, m);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdSMT error: the library was not
               initialized");
            exit(1);
            break;
        case CUBLAS_STATUS_INVALID_VALUE:
            printf("RTdSMT error: the parameters m,n<0,
               alpha,beta=NULL or improper settings of in
               -place mode");
            exit(1);
            break;
        case CUBLAS_STATUS_ARCH_MISMATCH:
            printf("RTdSMT error: the device does not
               support double-precision");
            exit(1);
            break;
        case CUBLAS_STATUS_EXECUTION_FAILED:
            printf("RTdSMT error: the function failed to
               launch on the GPU");
            exit(1);
            break;
    }

    status = cublasDestroy(handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdSMT error: the library was not
               initialized");
            exit(1);
            break;
    }
}

void RTdDMT(double *C, const double *A, int m, int n){
    cublasStatus_t status;
    cublasHandle_t handle;
    status = cublasCreate(&handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdDMT error: the CUDA Runtime
               Initialization Failed");
            exit(1);
            break;
        case CUBLAS_STATUS_ALLOC_FAILED:
            printf("RtdDMT error: the resources could not
                 be allocated");
            exit(1);
            break;
    }

    const double alpha = 1.0;
    const double beta = 0.0;
    status = cublasDgeam(handle, CUBLAS_OP_T, CUBLAS_OP_N
        , m, n, &alpha, A, m, &beta, A, m, C, m);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdDMT error: the library was not
                initialized");
            exit(1);
            break;
        case CUBLAS_STATUS_INVALID_VALUE:
            printf("RTdDMT error: the parameters m,n<0,
                alpha,beta=NULL or improper settings of in
                -place mode");
            exit(1);
            break;
        case CUBLAS_STATUS_ARCH_MISMATCH:
            printf("RTdDMT error: the device does not
                support double-precision");
            exit(1);
            break;
        case CUBLAS_STATUS_EXECUTION_FAILED:
            printf("RTdDMT error: the function failed to
                launch on the GPU");
            exit(1);
            break;
    }

    status = cublasDestroy(handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdDMT error: the library was not
                initialized");
            exit(1);
            break;
    }
}

void RTdSVV(float *C, const float *A, int m, const float
   *B, int n){
    cublasStatus_t status;
    cublasHandle_t handle;
    status = cublasCreate(&handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdSVV error: the CUDA Runtime
               Initialization Failed");
            exit(1);
            break;
        case CUBLAS_STATUS_ALLOC_FAILED:
            printf("RtdSVV error: the resources could not
                be allocated");
            exit(1);
            break;
    }

    const float alpha = 1.0;
    const float beta = 0.0;
    status = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_T
       , m, n, 1, &alpha, A, m, B, n, &beta, C, m);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdSVV error: the library was not
               initialized");
            exit(1);
            break;
        case CUBLAS_STATUS_INVALID_VALUE:
            printf("RTdSVV error: the parameters m,n,k<0"
               );
            exit(1);
            break;
        case CUBLAS_STATUS_ARCH_MISMATCH:
            printf("RTdSVV error: the device does not
               support double-precision");
            exit(1);
            break;
        case CUBLAS_STATUS_EXECUTION_FAILED:
            printf("RTdSVV error: the function failed to
               launch on the GPU");
            exit(1);
            break;
    }

    status = cublasDestroy(handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdSVV error: the library was not
               initialized");
            exit(1);
            break;
    }
}

void RTdDVV(double *C, const double *A, int m, const
   double *B, int n){
    cublasStatus_t status;
    cublasHandle_t handle;
    status = cublasCreate(&handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdDVV error: the CUDA Runtime
               Initialization Failed");
            exit(1);
            break;
        case CUBLAS_STATUS_ALLOC_FAILED:
            printf("RtdDVV error: the resources could not
                be allocated");
            exit(1);
            break;
    }

    const double alpha = 1.0;
    const double beta = 0.0;
    status = cublasDgemm(handle, CUBLAS_OP_N, CUBLAS_OP_T
       , m, n, 1, &alpha, A, m, B, n, &beta, C, m);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdDVV error: the library was not
                initialized");
            exit(1);
            break;
        case CUBLAS_STATUS_INVALID_VALUE:
            printf("RTdDVV error: the parameters m,n,k<0"
                );
            exit(1);
            break;
        case CUBLAS_STATUS_ARCH_MISMATCH:
            printf("RTdDVV error: the device does not
                support double-precision");
            exit(1);
            break;
        case CUBLAS_STATUS_EXECUTION_FAILED:
            printf("RTdDVV error: the function failed to
                launch on the GPU");
            exit(1);
            break;
    }

    status = cublasDestroy(handle);
    switch(status){
        case CUBLAS_STATUS_SUCCESS:
            break;
        case CUBLAS_STATUS_NOT_INITIALIZED:
            printf("RTdDVV error: the library was not
                initialized");
            exit(1);
            break;
    }
}
```

Appendix E
RT-CUDA Examples

Matrix Scaling

Inputs

Listing 22: kernel.c

```
void matrix_scale(float *C, float * restrict A, int scale
    , int N)
{
    for(int i=0; i < N; i++){
        for(int j=0; j < N; j++)
            C[i][j] = scale * A[i][j];
    }
}
```

Listing 23: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *A, *C;

    int memsize = N * N * sizeof(float);
    A = (float *)malloc(memsize);
    C = (float *)malloc(memsize);
    A[0] = 1;

    matrix_scale(C, A, 3.0, N);

    printf("A[0] = %f, C[0] = %f\n", A[0], C[0]);
    printf("End of Program\n");

    free(A);
    free(C);

    exit(0);
}
```

Listing 24: config.txt

```
LOOP_COLLAPSING=1
BLOCK_SKEW=1
PREFETCHING=0
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B
DATA_TYPE=float
KERNEL_NAMES=matrix_scale
2DMATRIX=1
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=8
MAX_SKEW_LEVEL=2
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Outputs

Listing 25: kernel.cu

```
__global__ void matrix_scale(float *C,float const *
   __restrict__ A,int scale,int N){
        int tid=threadIdx.x;
        int bid=blockIdx.x;
        int ij=bid*BLOCKSIZE+tid;
        {
                int i=(ij/N)*MERGE_LEVEL;
                int j=(ij%N)*SKEW_LEVEL;
                for(int m=0;m<MERGE_LEVEL;m++)
                        for(int n=0;n<SKEW_LEVEL;n++)
                                C[((i+m))*N+((j+n))]=
                                   scale*A[((i+m))*N+((j+
                                   n))];

        }
}
```

Listing 26: main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcudacublas.h"
#include "kernel.cu"
int main(int argc,char *argv[]){
        int N=1024;
        int GPU=0;
        if(argc>1)N=atoi(argv[1]);

        if(argc>2)GPU=atoi(argv[2]);

        cudaSetDevice (GPU);
        float *A,*C;
        int memsize=N*N*sizeof(float );
        cudaMallocManaged(&A,memsize);

        cudaMallocManaged(&C,memsize);
        A[0]=1;

        dim3 threads(BLOCKSIZE,1);
        dim3 grid(N*N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1)
           ;
        matrix_scale<<<grid,threads>>>(C,A,3.0,N);
        cudaDeviceSynchronize();

        printf("A[0] = %f, C[0] = %f\n",A[0],C[0]);

        printf("End of Program\n");

        cudaFree (A);
        cudaFree (C);
        cudaThreadExit();

}
```

Listing 27: params.h

```
#define BLOCKSIZE 32
#define MERGE_LEVEL 1
#define SKEW_LEVEL 1
```

Listing 28: Makefile

```
# OS Name (Linux or Darwin)
OSUPPER = $(shell uname -s 2>/dev/null | tr "[:lower:]" "
   [:upper:]")
OSLOWER = $(shell uname -s 2>/dev/null | tr "[:upper:]" "
   [:lower:]")

# Flags to detect 32-bit or 64-bit OS platform
OS_SIZE = $(shell uname -m | sed -e "s/i.86/32/" -e "s/
   x86_64/64/" -e "s/armv7l/32/")
OS_ARCH = $(shell uname -m | sed -e "s/i386/i686/")

# These flags will override any settings
ifeq ($(i386),1)
        OS_SIZE = 32
        OS_ARCH = i686
endif

ifeq ($(x86_64),1)
```

```
        OS_SIZE = 64
        OS_ARCH = x86_64
endif

ifeq ($(ARMv7),1)
        OS_SIZE = 32
        OS_ARCH = armv7l
endif

# Flags to detect either a Linux system (linux) or Mac
   OSX (darwin)
DARWIN = $(strip $(findstring DARWIN, $(OSUPPER)))

include ./findcudalib.mk

# Location of the CUDA Toolkit binaries and libraries
CUDA_PATH       ?= /usr/local/cuda-6.5
CUDA_INC_PATH   ?= $(CUDA_PATH)/include
CUDA_BIN_PATH   ?= $(CUDA_PATH)/bin
ifneq ($(DARWIN),)
  CUDA_LIB_PATH  ?= $(CUDA_PATH)/lib
else
  ifeq ($(OS_SIZE),32)
    CUDA_LIB_PATH  ?= $(CUDA_PATH)/lib
  else
    CUDA_LIB_PATH  ?= $(CUDA_PATH)/lib64
  endif
endif

# Common binaries
GCC             ?= g++
NVCC            := $(CUDA_BIN_PATH)/nvcc -ccbin $(GCC)

# Extra user flags
EXTRA_NVCCFLAGS ?=
EXTRA_LDFLAGS   ?=

# CUDA code generation flags
ifneq ($(OS_ARCH),armv7l)
GENCODE_SM10    :=
#-gencode arch=compute_10,code=sm_10
endif
GENCODE_SM20    :=
#-gencode arch=compute_20,code=sm_20
GENCODE_SM30    := -gencode arch=compute_30,code=sm_30 -
   gencode arch=compute_35,code=\"sm_35,compute_35\"
GENCODE_FLAGS   := $(GENCODE_SM10) $(GENCODE_SM20) $(
   GENCODE_SM30)

# OS-specific build flags
ifneq ($(DARWIN),)
      LDFLAGS   := -Xlinker -rpath $(CUDA_LIB_PATH) -L$(
         CUDA_LIB_PATH) -lcudart
      CCFLAGS   := -arch $(OS_ARCH)
else
      LDFLAGS   := -L$(CUDA_LIB_PATH) $(CUDALINK) -
         lcudart
  ifeq ($(OS_ARCH),armv7l)
    ifeq ($(abi),gnueabi)
      CCFLAGS   += -mfloat-abi=softfp
    else
      # default to gnueabihf
      override abi := gnueabihf
      LDFLAGS   += -Xlinker --dynamic-linker=/lib/ld-
         linux-armhf.so.3
      CCFLAGS   += -mfloat-abi=hard
    endif
  else
    ifeq ($(OS_SIZE),32)
      CCFLAGS   := -m32
    else
      CCFLAGS   := -m64
    endif
  endif
endif

ifeq ($(ARMv7),1)
ifneq ($(TARGET_FS),)
LDFLAGS += -Xlinker -rpath-link=$(TARGET_FS)/lib
endif
endif

# OS-architecture specific flags
ifeq ($(OS_SIZE),32)
      NVCCFLAGS := -m32
  ifeq ($(ARMv7),1)
      NVCCFLAGS += -target-cpu-arch ARM
  endif
else
      NVCCFLAGS := -m64
endif

# Debug build flags
ifeq ($(dbg),1)
```

```
      CCFLAGS    += -g
      NVCCFLAGS  += -g -G
      TARGET     := debug
else
      TARGET     := release
endif


# Common includes and paths for CUDA
INCLUDES      := -I$(CUDA_INC_PATH) -I. -I.. -I../../
   common/inc -I/usr/local/cuda_sdk-4.2/C/common/inc

# Target rules
all: build

build: main

main.o: main.cu kernel.cu params.h
        $(NVCC) $(NVCCFLAGS) $(EXTRA_NVCCFLAGS) $(
           GENCODE_FLAGS) $(INCLUDES) -Xptxas -v -o $ -c
           < main : main.o(GCC) (CCFLAGS) – o $+ $(LDFLAGS) $(
           EXTRA_LDFLAGS) -lcublas -lcusparse

run: build
        ./main

clean:
        rm -f main main.o
        rm -rf ./main

clobber: clean
```

Matrix Addition

Inputs

Listing 29: kernel.c

```
void matrix_add(float *C, float * restrict A, float *
   restrict B, int N)
{
    for(int i=0; i < N; i++){
        for(int j=0; j < N; j++)
```

```
                C[i][j] = A[i][j] + B[i][j];
    }
}
```

Listing 30: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *A, *B, *C;

    int memsize = N * N * sizeof(float);
    A = (float *)malloc(memsize);
    B = (float *)malloc(memsize);
    C = (float *)malloc(memsize);

    matrix_add(C, A, B, N);

    free(A);
    free(B);
    free(C);

    exit(0);
}
```

Listing 31: config.txt

```
LOOP_COLLAPSING=1
BLOCK_SKEW=1
PREFETCHING=0
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B
DATA_TYPE=float
KERNEL_NAMES=matrix_add
2DMATRIX=1
ROW_DIM=N
```

```
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=0
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Outputs

Listing 32: kernel.cu

```
__global__ void matrix_add(float *C, float * __restrict__
    A, float * __restrict__ B, int N)
{
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    int ij = bid * BLOCKSIZE + tid;
    int i = (ij / n) * MERGE_LEVEL;
    int j = (ij % n) * SKEW_LEVEL;
    for(int s=0; s < SKEW_LEVEL; s++)
        for(int m=0; m < MERGE_LEVEL; m++)
                C[(i + m) * n + j + s] = A[(i + m) * n +
                   j + s] + B[(i + m) * n + j + s];
}
```

Listing 33: main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc,char *argv[]){
        int N=1024;
        int GPU=0;
        if(argc>1)N=atoi(argv[1]);

        if(argc>2)GPU=atoi(argv[2]);

        cudaSetDevice (GPU);

        float *A,*B,*C;
        int memsize=N*N*sizeof(float );
        cudaMallocManaged(&A,memsize);
        cudaMallocManaged(&B,memsize);
        cudaMallocManaged(&C,memsize);

        dim3 threads(BLOCKSIZE,1);
        dim3 grid(N*N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1)
           ;
        matrix_add<<<grid,threads>>>(C,A,B,N);
        cudaDeviceSynchronize();

        cudaFree (A);
        cudaFree (B);
        cudaFree (C);
        cudaThreadExit();
}
```

Listing 34: params.h

```
#define BLOCKSIZE 128
#define MERGE_LEVEL 1
#define SKEW_LEVEL 2
```

Demosaic

Inputs

Listing 35: kernel.c

```
void demosaic(float *r_G, float * restrict GPU_G, int N)
{
    for (int i = 0; i < N-16; ++i){
        for (int j = 0; j < N-16; ++j) {
                r_G[i*(N-16)+j] = GPU_G[(i-1+15)*N+j
                   +15]*0.25
                                + GPU_G[(i+15)*N+j
                                   -1+15]*0.25
                                + GPU_G[(i+15)*N+j+15]
                                + GPU_G[(i+15)*N+j
                                   +1+15]*0.25
                                + GPU_G[(i+1+15)*N+j
                                   +15]*0.25;
        }
    }
}
```

Listing 36: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *A, *C;

    int memsizeA = (N+16) * (N+16) * sizeof(float);
    int memsizeC = N * N * sizeof(float);
    A = (float *)malloc(memsizeA);
    C = (float *)malloc(memsizeC);

    demosaic(C, A, N);

    free(A);
    free(C);

    exit(0);
}
```

Listing 37: config.txt

```
LOOP_COLLAPSING=1
BLOCK_SKEW=1
PREFETCHING=0
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B
DATA_TYPE=float
KERNEL_NAMES=demosaic
2DMATRIX=1
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=0
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Outputs

Listing 38: kernel.cu

```
__global__ void demosaic(float *r_G, float * __restrict__
    GPU_G, int N)
{
    int tid = threadIdx.x;
    int bid = blockIdx.x;
    int ij = bid * BLOCKSIZE + tid;
    int i = (ij / (N-16)) * MERGE_LEVEL;
    int j = (ij % (N-16)) * SKEW_LEVEL;

    for(int s=0; s < SKEW_LEVEL; s++)
        for(int m=0; m < MERGE_LEVEL; m++)
            r_G[(i + m)*(N-16)+j+s] = GPU_G[((i+m)-1+15)*
               N+j+15+s]*0.25 + GPU_G[((i+m)+15)*N+j
               -1+15+s]*0.25 + GPU_G[((i+m)+15)*N+j+15+s]
                + GPU_G[((i+m)+15)*N+j+1+15+s]*0.25 +
               GPU_G[((i+m)+1+15)*N+j+15+s]*0.25;
}
```

Listing 39: main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc,char *argv[]){
        int N=1024;
        int GPU=0;
        if(argc>1)N=atoi(argv[1]);

        if(argc>2)GPU=atoi(argv[2]);

        cudaSetDevice (GPU);

        float *A,*C;
    int memsizeA = (N+16) * (N+16) * sizeof(float);
    int memsizeC = N * N * sizeof(float);
        cudaMallocManaged(&A,memsizeA);
        cudaMallocManaged(&C,memsizeC);

        dim3 threads(BLOCKSIZE,1);
        dim3 grid(N*N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1)
           ;
        demosaic<<<grid,threads>>>(C,A,N);
        cudaDeviceSynchronize();

        cudaFree (A);
        cudaFree (C);
        cudaThreadExit();
}
```

Listing 40: params.h

```
#define BLOCKSIZE 128
#define MERGE_LEVEL 8
#define SKEW_LEVEL 1
```

Histogram

Inputs

Listing 41: kernel.c

```
void histogram(int* histogram, int * restrict A, int N)
{
    for(i = 0;i<N;i++){
        for(j = 0;j<N;j++){
                int b;
                b = A[i][j];
                histogram[b] = histogram[b] + 1;
        }
    }
}
```

Listing 42: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    int *A, *C;

    int memsizeA = N * N * sizeof(int);
    int memsizeC = N * sizeof(int);
    A = (int *)malloc(memsizeA);
    C = (int *)malloc(memsizeC);

    histogram(C, A, N);

    free(A);
    free(C);

    exit(0);
}
```

Listing 43: config.txt

```
LOOP_COLLAPSING=1
BLOCK_SKEW=1
PREFETCHING=0
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B
DATA_TYPE=float
KERNEL_NAMES=histogram
2DMATRIX=1
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=0
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Outputs

Listing 44: kernel.cu

```
__global__ void histogram(int *histogram, int *
   __restrict__ a, int N)
{
        int tid = threadIdx.x;
        int bid = blockIdx.x;
        int ij = bid * BLOCKSIZE + tid;
        int i = (ij / N) * MERGE_LEVEL;
        int j = (ij % N) * SKEW_LEVEL;
        int b;
        for(int s=0; s < SKEW_LEVEL; s++)
        for(int m=0; m < MERGE_LEVEL; m++){
                b = abs((int)a[(i+m) * N + j + s]);
                atomicAdd(&histogram[b],1);
        }
}
```

Listing 45: main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc,char *argv[]){
        int N=1024;
        int GPU=0;
        if(argc>1)N=atoi(argv[1]);

        if(argc>2)GPU=atoi(argv[2]);

        cudaSetDevice (GPU);

        int *A,*C;
    int memsizeA = N * N * sizeof(int);
    int memsizeC = N * sizeof(int);
        cudaMallocManaged(&A,memsizeA);
        cudaMallocManaged(&C,memsizeC);

        dim3 threads(BLOCKSIZE,1);
        dim3 grid(N*N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1)
           ;
        histogram<<<grid,threads>>>(C,A,N);
        cudaDeviceSynchronize();

        cudaFree (A);
        cudaFree (C);
        cudaThreadExit();
}
```

Listing 46: params.h

```
#define BLOCKSIZE 128
#define MERGE_LEVEL 2
#define SKEW_LEVEL 16
```

Matrix-Vector Multiplication

Inputs

Listing 47: kernel.c

```
void matrix_mv(float *C, float * restrict A, float *
   restrict B, int N)
{
    float sum=0.0;
    for(int i=0; i < N; i++){
        for(int k=0; k < N; k++)
                sum += A[i][k] * B[k];
        C[i] = sum;
    }
}
```

Listing 48: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
```

```
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *A, *B, *C;

    int memsize = N * N * sizeof(float);
    int memsizevec = N * sizeof(float);
    A = (float *)malloc(memsize);
    B = (float *)malloc(memsizevec);
    C = (float *)malloc(memsizevec);

    matrix_mv(C, A, B, N);

    free(A);
    free(B);
    free(C);

    exit(0);
}
```

Listing 49: config.txt

```
LOOP_COLLAPSING=0
BLOCK_SKEW=0
PREFETCHING=1
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B
DATA_TYPE=float
KERNEL_NAMES=matrix_mv
2DMATRIX=0
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=0
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Outputs

Listing 50: kernel.cu

```
__global__ void matrix_mv(float *C, float * __restrict__
   A, float * __restrict__ B, int N)
{
    __shared__ float As[MERGE_LEVEL][BLOCKSIZE];
    float sum[MERGE_LEVEL];
        for(int i=0; i < MERGE_LEVEL; i++)
                sum[i] = 0.0;
    int tid=threadIdx.x;
    int bid=blockIdx.x;
    int i=(bid*BLOCKSIZE+tid)*MERGE_LEVEL;
    int k ;
    for(int m=0;m<MERGE_LEVEL;m++){
        As[m][tid]=A[((i+m))*N+(k)+tid];
    }
    __syncthreads();

    for(k=0;k<N-BLOCKSIZE;k+=BLOCKSIZE){
        for(int t=0;t<BLOCKSIZE;t++){
            float b=B[k+t];
            for(int m=0;m<MERGE_LEVEL;m++)
                sum[m]+=As[m][t]*b;
        }
        __syncthreads();

        for(int m=0;m<MERGE_LEVEL;m++){
            As[m][tid]=A[((i+m))*N+(k+BLOCKSIZE)+tid];
        }
        __syncthreads();
    }
    for(int t=0;t<BLOCKSIZE;t++){
        float b=B[k+t];
        for(int m=0;m<MERGE_LEVEL;m++)
            sum[m]+=As[m][t]*b;
        }
    for(int m=0;m<MERGE_LEVEL;m++)
        C[(i+m)]=sum[m];
}
```

Listing 51: main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
```

```
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc,char *argv[]){
        int N=1024;
        int GPU=0;
        if(argc>1)N=atoi(argv[1]);

        if(argc>2)GPU=atoi(argv[2]);

        cudaSetDevice (GPU);

        float *A,*B,*C;
        int memsize=N*N*sizeof(float );
        int memsizevec=N*sizeof(float );
        cudaMallocManaged(&A,memsize);
        cudaMallocManaged(&B,memsizevec);
        cudaMallocManaged(&C,memsizevec);

        dim3 threads(BLOCKSIZE,1);
        dim3 grid(N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1);
        matrix_mv<<<grid,threads>>>(C,A,B,N);
        cudaDeviceSynchronize();

        cudaFree (A);
        cudaFree (B);
        cudaFree (C);
        cudaThreadExit();
}
```

Listing 52: params.h

```
#define BLOCKSIZE 64
#define MERGE_LEVEL 1
#define SKEW_LEVEL 1
```

Vector-Vector Multiplication

Inputs

Listing 53: kernel.c

```
void matrix_vv(float *C, float * restrict A, float *
   restrict B, int N)
{
    for(int i=0; i < N; i++)
        for(int j=0; j < N; j++)
                C[i][j] = A[i] * B[j];
}
```

Listing 54: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *A, *B, *C;

    int memsize = N * N * sizeof(float);
    int memsizevec = N * sizeof(float);
    A = (float *)malloc(memsizevec);
    B = (float *)malloc(memsizevec);
    C = (float *)malloc(memsize);

    matrix_vv(C, A, B, N);

    free(A);
    free(B);
    free(C);

    exit(0);
}
```

Listing 55: config.txt

```
LOOP_COLLAPSING=1
BLOCK_SKEW=1
PREFETCHING=0
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B
DATA_TYPE=float
KERNEL_NAMES=matrix_vv
2DMATRIX=1
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=0
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Outputs

Listing 56: kernel.cu

```
__global__ void matrix_vv(float *C, float * __restrict__
   A, float * __restrict__ B, int N)
{
    int tid=threadIdx.x;
    int bid=blockIdx.x;
    int ij=bid*BLOCKSIZE+tid;

    int i=(ij/N)*MERGE_LEVEL;
    int j=(ij%N)*SKEW_LEVEL;
    for(int n=0;n<SKEW_LEVEL;n++){
        float b=B[(j+n)];
        for(int m=0;m<MERGE_LEVEL;m++)
            C[((i+m))*N+((j+n))]=A[(i+m)]*b;
    }
}
```

Listing 57: main.cu

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc,char *argv[]){
        int N=1024;
        int GPU=0;
        if(argc>1)N=atoi(argv[1]);

        if(argc>2)GPU=atoi(argv[2]);

        cudaSetDevice (GPU);

        float *A,*B,*C;
        int memsize=N*N*sizeof(float );
        int memsizevec=N*sizeof(float );
        cudaMallocManaged(&A,memsizevec);
        cudaMallocManaged(&B,memsizevec);
        cudaMallocManaged(&C,memsize);

        dim3 threads(BLOCKSIZE,1);
        dim3 grid(N*N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1)
           ;
        matrix_vv<<<grid,threads>>>(C,A,B,N);
        cudaDeviceSynchronize();

        cudaFree (A);
        cudaFree (B);
        cudaFree (C);
        cudaThreadExit();
}
```

Listing 58: params.h

```
#define BLOCKSIZE 256
#define MERGE_LEVEL 32
#define SKEW_LEVEL 1
```

AXPY

Inputs

Listing 59: kernel.c

```
void axpy(double *C, double * restrict A, double alpha,
   int N)
{
    for(int i=0; i < N; i++)
        C[i] = C[i] + alpha * A[i];
}
```

Listing 60: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    double *A, *B, *C;

    int memsize = N * sizeof(double);
    A = (double *)malloc(memsize);
    C = (double *)malloc(memsize);

    axpy(C, A, 3.0, N);

    free(A);
    free(C);

    exit(0);
}
```

Listing 61: config.txt

```
LOOP_COLLAPSING=0
BLOCK_SKEW=0
PREFETCHING=0
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B
DATA_TYPE=double
KERNEL_NAMES=axpy
2DMATRIX=0
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=0
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Outputs

Listing 62: kernel.cu

```
__global__ void axpy(double *C,double const *__restrict__
    A,double alpha,int N){
        int tid=threadIdx.x;
        int bid=blockIdx.x;
        int i=(bid*BLOCKSIZE+tid)*MERGE_LEVEL;
        for(int m=0;m<MERGE_LEVEL;m++)
                C[(i+m)]=C[(i+m)]+alpha*A[(i+m)];
}
```

Listing 63: main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

```
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc,char *argv[]){
        int N=1024;
        int GPU=0;
        if(argc>1)N=atoi(argv[1]);

        if(argc>2)GPU=atoi(argv[2]);

        cudaSetDevice (GPU);

        double *A,*C;
        int memsize=N*sizeof(double);
        cudaMallocManaged(&A,memsize);
        cudaMallocManaged(&C,memsize);

        dim3 threads(BLOCKSIZE,1);
        dim3 grid(N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1);
        axpy<<<grid,threads>>>(C,A,3.0,N);
        cudaDeviceSynchronize();

        cudaFree (A);
        cudaFree (C);
        cudaThreadExit();
}
```

Listing 64: params.h

```
#define BLOCKSIZE 128
#define MERGE_LEVEL 1
#define SKEW_LEVEL 1
```

Jacobi Iteration

Inputs

Listing 65: kernel.c

```
void jacobi(double *NX, double * restrict A, double *
   restrict X, double * restrict B, int N)
{
    double a;
    double sum;
    for(int i=0; i < N; i++){
        a = A[i][i];
        sum = -a * X[i];
        for(int j=0; j < N; j++)
                sum += A[i][j] * X[j];
        NX[i] = (B[i] - sum)/a;
    }
}
```

Listing 66: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    double *A, *B, *X, *NX;

    int memsize = N * N * sizeof(double);
    int memsizevec = N * sizeof(double);
    A = (double *)malloc(memsize);
    X = (double *)malloc(memsizevec);
    NX = (double *)malloc(memsizevec);
    B = (double *)malloc(memsizevec);

    for(int iter=0; iter < 100; iter++){
        jacobi(NX, A, X, B, N);
        jacobi(X, A, NX, B, N);
    }

    free(A);
    free(B);
    free(X);
    free(NX);

    exit(0);
}
```

Listing 67: config.txt

```
LOOP_COLLAPSING=0
BLOCK_SKEW=0
PREFETCHING=1
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B,X,NX
DATA_TYPE=double
KERNEL_NAMES=jacobi
2DMATRIX=0
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=1
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Outputs

Listing 68: kernel.cu

```
__global__ void jacobi(double *NX,double const *
   __restrict__ A,double const *__restrict__ X,double
   const *__restrict__ B,int N){
        __shared__ double As[MERGE_LEVEL][BLOCKSIZE];
        double sum[MERGE_LEVEL];
        int tid=threadIdx.x;
        int bid=blockIdx.x;
        int i=(bid*BLOCKSIZE+tid)*MERGE_LEVEL;
        {
                for(int m=0;m<MERGE_LEVEL;m++){
                        a[m] = A[i][i];
                        sum[m]=-a[m]*X[(i+m)];
                }

                int j ;
                for(int m=0;m<MERGE_LEVEL;m++)
                        {
                        As[m][tid]=A[((i+m))*N+(j)+tid];

                        }
                __syncthreads();

                for(j=0;j<N-BLOCKSIZE;j+=BLOCKSIZE)
                        {
                                for(int t=0;t<BLOCKSIZE;t
                                   ++)
                                        for(int m=0;m<
                                           MERGE_LEVEL;m
                                           ++)
                                                sum[m]+=
                                                   As[m][
                                                   t]*X[j
                                                   ];

                                __syncthreads();

                                for(int m=0;m<MERGE_LEVEL
                                   ;m++)
                                        {
                                        As[m][tid]=A[((i+
                                           m))*N+(j+
                                           BLOCKSIZE)+tid
                                           ];

                                        }
                                __syncthreads();

                        }
                for(int t=0;t<BLOCKSIZE;t++)
                        for(int m=0;m<MERGE_LEVEL;m++)
                                sum[m]+=As[m][t]*X[j];

                for(int m=0;m<MERGE_LEVEL;m++)
                        NX[(i+m)]=(B[(i+m)]-sum[m])/a[m];

        }
```

```
}
```

Listing 69: main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc,char *argv[]){
        int N=1024;
        int GPU=0;
        if(argc>1)N=atoi(argv[1]);

        if(argc>2)GPU=atoi(argv[2]);

        cudaSetDevice (GPU);

        double *A,*B,*NX,*X;
        int memsize=N*N*sizeof(double);
        int memsizevec=N*sizeof(double);
        cudaMallocManaged(&A,memsize);
        cudaMallocManaged(&B,memsizevec);
        cudaMallocManaged(&X,memsizevec);
        cudaMallocManaged(&NX,memsizevec);

        dim3 threads(BLOCKSIZE,1);
        dim3 grid(N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1);
        for(int iter=0; iter<100; i++){
        jacobi<<<grid,threads>>>(NX,A,X,B,N);
        cudaDeviceSynchronize();
        jacobi<<<grid,threads>>>(X,A,NX,B,N);
        cudaDeviceSynchronize();
    }

        cudaFree (A);
        cudaFree (B);
        cudaFree (X);
        cudaFree (NX);
        cudaThreadExit();
}
```

Listing 70: params.h

```
#define BLOCKSIZE 1024
#define MERGE_LEVEL 1
#define SKEW_LEVEL 1
```

Conjugate Gradient using RT-CUDA API

Inputs

Listing 71: kernel.c

```
void sub(float *C, float * restrict A, float * restrict B
   , int N)
{
    for(int i=0; i<N; i++)
        C[i] = A[i] - B[i];
}

void copy(float *C, float * restrict A, int N)
{
    for(int i=0; i<N; i++)
        C[i] = A[i];
}

void scaleadd(float *C, float * restrict A, int N, float
   alpha)
{
    for(int i=0; i<N; i++)
        C[i] += alpha * A[i];
}
```

```
void scalesub(float *C, float * restrict A, int N, float
    alpha)
{
    for(int i=0; i<N; i++)
        C[i] -= alpha * A[i];
}

void scaleaddstore(float *C, float * restrict A, float *
    restrict B, int N, float alpha)
{
    for(int i=0; i<N; i++)
        C[i] = alpha * A[i] + B[i];
}
```

Listing 72: main.c

```
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *B, *X, *P, *R, *AP;

    int memsize = N * N * sizeof(float);
    int memsizevec = N * sizeof(float);
    X = (float *)malloc(memsizevec);
    B = (float *)malloc(memsizevec);
    P = (float *)malloc(memsizevec);
    R = (float *)malloc(memsizevec);
    AP = (float *)malloc(memsizevec);

    RTspSArray *A;
    RTspSArrayLoadFromFile(argv[1], A);

    RTspSMV(R, A, X, N, N);
    sub(R, B, R, N);
    copy(P, R, N);
    for(int k=0; k < 100; k++){
        float rr;
```

```
            float pp;
            RTdSDOT(R, R, N, &rr);
            RTspSMV(AP, A, P, N, N);
            RTdSDOT(P, AP, N, &pp);
            float alpha = rr / pp;
            scaleadd(X, P, N, alpha);
            scalesub(R, AP, N, alpha);
            float rrn;
            RTdSDOT(R, R, N, &rrn);
            if(rrn < 1e-10) break;
            float beta = rrn / rr;
            scaleaddstore(P, P, R, N, beta);
    }

    free(A);
    free(B);
    free(X);
    free(P);
    free(R);
    free(AP);

    exit(0);
}
```

Listing 73: config.txt

```
LOOP_COLLAPSING=0
BLOCK_SKEW=0
PREFETCHING=0
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B,X,NX
DATA_TYPE=float
KERNEL_NAMES=sub,copy,scaleadd,scalesub,scaleaddstore
2DMATRIX=0
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=1
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Outputs

Listing 74: kernel.cu

```
__global__ void sub(float *C,float const *__restrict__ A,
   float const *__restrict__ B,int N){
        int tid=threadIdx.x;
        int bid=blockIdx.x;
        int i=(bid*blockDim.x+tid)*MERGE_LEVEL;
        for(int m=0;m<MERGE_LEVEL;m++)
                C[(i+m)]=A[(i+m)]-B[(i+m)];

}

__global__ void copy(float *C,float const *__restrict__ A
   ,int N){
        int tid=threadIdx.x;
        int bid=blockIdx.x;
        int i=(bid*blockDim.x+tid)*MERGE_LEVEL;
        for(int m=0;m<MERGE_LEVEL;m++)
                C[(i+m)]=A[(i+m)];

}

__global__ void scaleadd(float *C,float const *
   __restrict__ A,int N,float alpha){
  int tid=threadIdx.x;
  int bid=blockIdx.x;
  int i=(bid*blockDim.x+tid)*MERGE_LEVEL;
  for(int m=0;m<MERGE_LEVEL;m++)
     C[(i+m)]+=alpha*A[(i+m)];

}

__global__ void scalesub(float *C,float const *
   __restrict__ A,int N,float alpha){
  int tid=threadIdx.x;
  int bid=blockIdx.x;
  int i=(bid*blockDim.x+tid)*MERGE_LEVEL;
  for(int m=0;m<MERGE_LEVEL;m++)
     C[(i+m)]-=alpha*A[(i+m)];

}

__global__ void scaleaddstore(float *C,float const *
   __restrict__ A,float const *__restrict__ B,int N,float
    alpha){
  int tid=threadIdx.x;
  int bid=blockIdx.x;
  int i=(bid*blockDim.x+tid)*MERGE_LEVEL;
  for(int m=0;m<MERGE_LEVEL;m++)
     C[(i+m)]=alpha*A[(i+m)]+B[(i+m)];

}
```

Listing 75: main.cu

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %
                   s\n", __FILE__, __LINE__, msg, (int)
                   err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc, char *argv[])
{
    int N = 1024;
    int GPU = 0;
    if(argc > 1)
        N = atoi(argv[1]);
    if(argc > 2)
        GPU = atoi(argv[2]);

    cudaSetDevice(GPU);

    float *B, *X, *P, *R, *AP;

    int memsize = N * N * sizeof(float);
    int memsizevec = N * sizeof(float);
    cudaMallocManaged(&X,memsizevec);
    cudaMallocManaged(&B,memsizevec);
    cudaMallocManaged(&P,memsizevec);
    cudaMallocManaged(&R,memsizevec);
    cudaMallocManaged(&AP,memsizevec);

    RTspSArray *A;
    RTspSArrayLoadFromFile(argv[1], A);

    dim3 threads(BLOCKSIZE, 1)
    dim3 grid(N/BLOCKSIZE/MERGE_LEVEL,SKEW_LEVEL, 1);

    RTspSMV(R, A, X, N, N);
    sub<<<grid,threads>>>(R, B, R, N);
        cudaDeviceSynchronize();
    copy<<<grid,threads>>>(P, R, N);
        cudaDeviceSynchronize();
    for(int k=0; k < 100; k++){
        float rr;
        float pp;
        RTdSDOT(R, R, N, &rr);
        RTspSMV(AP, A, P, N, N);
        RTdSDOT(P, AP, N, &pp);
        float alpha = rr / pp;
        scaleadd<<<grid,threads>>>(X, P, N, alpha);
        cudaDeviceSynchronize();
        scalesub<<<grid,threads>>>(R, AP, N, alpha);
        cudaDeviceSynchronize();
        float rrn;
        RTdSDOT(R, R, N, &rrn);
        if(rrn < 1e-10) break;
        float beta = rrn / rr;
        scaleaddstore<<<grid,threads>>>(P, P, R, N, beta)
           ;
        cudaDeviceSynchronize();
    }

    cudaFree(A);
    cudaFree(B);
    cudaFree(X);
    cudaFree(P);
    cudaFree(R);
    cudaFree(AP);

    cudaThreadExit();
}
```

Listing 76: params.h

```
#define BLOCKSIZE 512
```

```
#define MERGE_LEVEL 1
#define SKEW_LEVEL 1
```

Appendix F
RT-CUDA Third Party Evaluation

We assigned different applications to a set of four MS/PhD students to analyze and evaluate the compilation, execution, and performance of RT-CUDA as a course assignment. The objective of the assignment is to analyze, experiment, and test RT-CUDA as a newly developed compiler to convert a C-like program into an optimized CUDA program with user directives in a configuration file for guiding the compiler. The tests have been performed using a set of numerical applications including:

1. Scaling Matrix (sM)
2. Matrix Addition (MAdd)
3. Demosaic
4. Histogram
5. Matrix-matrix multiplication (MM)
6. Matrix-vector multiplication (MV)
7. Vector-vector multiplication inner product (VV)
8. AXPY
9. Jacobi iterative solver

10. Conjugate Gradient (CG)

The students are assigned the following applications:

- Student 1: sM, MAdd, Demosaic
- Student 2: Histogram and Matrix-matrix multiplication (MM)
- Student 3: AXPY and Jacobi iterative solver
- Student 4: Vector inner product (VV) and Conjugate Gradient (CG)

For each assigned application, the assignment consists of writing student analysis on the following three aspects:

(a) **Familiarity with RT-CUDA environment:** source code in C, compiler transformation, and, configuration file, compiling process, CUDA generated code, and run-time data of CUDA generated programs. Write about your own comments of the source code (if any), configuration file, and the way the compiler is running, and generated CUDA code.

(b) **Analysis and Inspection of the generated CUDA code:** For each application, carry out the following:

   - Inspect the configuration file and try linking its information with the nature of the application as a user hint to guide the compiler.
   - Inspect the CUDA generated code. For this student tries to (1) identify the mapping of the thread to the result ($Th \rightarrow result$), (2) find whether tilting is used or not and how it is deciding on the tile size, (3) find if

coalesced access is granted, (4) find whether data is copied from GM into ShM prior to computation, (5) your analysis of your choice.

- Carry out some slight modification of the source code and study its effect on the generated CUDA code. We may lightly change the computed equation as above. The student is expected to inspect the CUDA generated code, compare with the original CUDA before the change, try to track changes and analyse the code change whether this is a logical change or not, does this seems to be optimized or not. For example, multiply a result by some constant c such as the equation sum=sum+x(i) is changed to sum=sum+x(i)*c. Observe the generated code: is it sum=sum+x(i)*c or sum=sum+x(i) and later sum = sum*c! Try your own checking for simple optimizations that may impact the generated code. For each application, write down your experiments showing the original code, modified code and your inspection and analysis of the generated code.

(c) **Run-time Analysis of Scalability:** Analysis of problem scalability based on run time data:

- For each application, compile the C source code after inspection of the configuration file, and obtains the corresponding RT-CUDA code. Run the code on GPU and collect the execution time (ET(N)). Vary the problem size as N, 1.4N, 1.8N, 2N , and 2.4N for some reasonable value of N while avoiding small values and too large values. Plot the execution time versus the problem size. The student is expected to show the ET plots versus problem size and provide careful comments on the scalability of ET(N). Is the plotted data scales in way according to the arithmetic complexity of the problem! For example, if the arithmetic complexity of the problem is O(N^2), then ET(2N) must be >= 4*ET(N).

- Carry out some slight modification of the source code and study its effect (execution time scalability) on the run time performance of the generated code, i.e. another test is to modify the C coding at the computed equation. For example, instead of using a vector V we may use 2V and see the impact on performance. The student is expected to show the ET plots versus problem size and link this to the ET of the original problem (before the change) and try see the impact (scalability) of the change by observing the plots.

Author's comments are also included in bold box with blue/bold text under each reported issue.

Student 1: sM, MAdd, Demosaic

a. **Familiarity with RT-CUDA environment**. source code in C, compiler transformation, and, configuration file, compiling process, CUDA generated code, and run-time data of CUDA generated programs. Write about your own comments of the source code (if any), configuration file, and the way the compiler is running, and generated CUDA code.

- Comments on source code in C:

1. Matrix Scale:

   matrix_scale(C, A, 3.0, N) → it's better to make the scale with own variable. Not hard coded. So it will be more readable

   eg: float scale = 3.0; matrix_scale(C, A, scale, N)

It depends on the user input, whatever parameters are provided by user it will be converted as is.

2. Matrix Add: -
3. Demosaic:

   In this code:

   r_G[i*(N-16)+j] = GPU_G[(i-1+15)*N+j+15]*0.25+ GPU_G[(i+15)*N+j-1+15]*0.25+ GPU_G[(i+15)*N+j+15]+ GPU_G[(i+15)*N+j+1+15]*0.25+ GPU_G[(i+1+15)*N+j+15]*0.25;

   Is it correct to put the multiplier as 0.25 or it should be 0.2?

This is the application requirement to use 0.25 as multiplier

4. General:

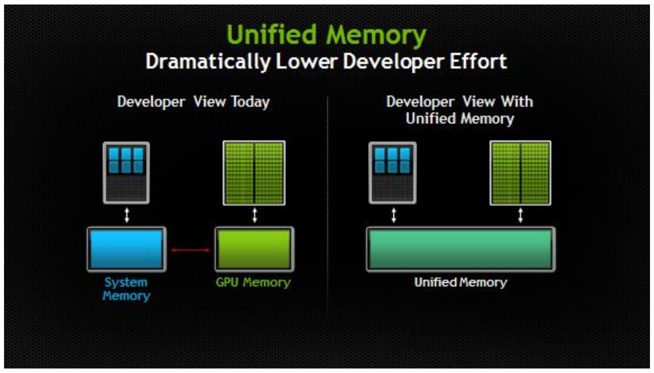
   - cudaSetDevice(GPU); set which cuda device will we use.

This depends on the user whether he want to use specific GPU device or the default.

- The matrix_scale is invoked by using the number of threads for each blok is BLOCKSIZE (one dimensional) and the number of block in grid is N*N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL

This is true. The tool uses one dimensional thread blocks and blocks grid only as it applies loop collapsing to merge two nested loops into a single loop. But, the number of grid to be initialized depends on the total number of elements in the array for computations which is NxN distributed among all threads based on thread block size (BLOCKSIZE) and thread granularity (MERGE_LEVEL and SKEW_LEVEL)

- cudaMallocManaged creates a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer. The key is that the system automatically migrates data allocated in Unified Memory between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU.



It's really simplified the code for memory transfer between CPU and GPU

Yes, this is the reason that we use this in RT-CUDA.

- cudaDeviceSynchronize will blocks until the device has completed all preceding requested tasks.

This is the recommended method in CUDA Programming best practices.

- Configuration File:

    1. I found that the suggested configuration is correct, but in some configuration it didn't configure well. For example, PREFETCHING=0 but the configuration still put PREFETCHED_ARRAYS=A & NON_PREFETCHED_ARRAYS=B, we should put this out.

RT-CUDA ignores other two parameters if PREFETCHING=0. But, these are mandatory in the case of PREFETCHING=1. So, these cannot be removed in the configuration. From the user point of view, we are planning to provide a graphical user interface in future for the configuration and there this will be avoided.

    2. And I also tried to change the DATA_TYPE=float to int, but it doesn't give me any error. Should it affects something?

Same as the above comment in (1).

- Compiling process:

    1. Display warning in several places:

       rcuda.h(605): warning: variable "ret_code" was set but never used

       rcuda.h: In function 'void RTspSArrayLoadFromFile(char*, RTspSArray*)':

       rcuda.h:596:106: warning: format '%lg' expects argument of type 'double*', but argument 5 has type 'float*' [-Wformat=]

       Maybe it's better to fix this.

This has been fixed in v2.3.

    2. While checking for best configuration, I saw the timer only check once. And this value will really varied based on current state machine. I think the result of best configuration could be wrong because sometimes even for the best configuration if executed several times, the result could be lower than other configuration. So we need

to figure out how to measure the time to determine the best configuration. Maybe we could use average execution time.

RT-CUDA uses the average kernel execution times for determining the optimal kernel parameters. So, in order to get stable results, the user can run the target function in main.c for multiple times using loop which will then converted as is by RT-CUDA.

- Run-time data of CUDA generated program:

1. Matrix scale:

   7.55e-04s

   time = 7.55E-4s

   Optimal Block Size = 32

   Optimal Merge Level = 1

   Optimal Skew Level = 1

2. Matrix add:

   1.40e-03s

   time = 0.0014s

   Optimal Block Size = 32

   Optimal Merge Level = 1

   Optimal Skew Level = 1

3. Demosaic: (with new kernel from Ayaz, now it’s working)

   Optimal Block Size = 1024

   Optimal Merge Level = 1

   Optimal Skew Level = 1

b. **Analysis and Inspection of the generated CUDA code.** For each application, carry out the following:

a. Inspect the configuration file and try linking its information with the nature of the application as a user hint to guide the compiler.

Changing the LOOP_COLLAPSING and BLOCK_SKEW:

Result:

if LOOP_COLLAPSING is equals to 0 (disabled) in two nested loop, the code will not be compiled. That's why the result is like this:

time = 0.0

Optimal Block Size = 0

Optimal Merge Level = 0

Optimal Skew Level = 0

Some of the parameters are depended. If LOOP_COLLAPSING is disabled then 2DMATRIX should also be disabled. Then, it will compile and run correctly. It has been checked now in v2.3 of RT-CUDA.

- If BLOCK_SKEW disabled, shows correct code in kernel.cu, but wrong code in main.cu.

  Original code:

  dim3 grid(N*N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1);

  Code after corrected:

  dim3 grid(N*N/BLOCKSIZE/MERGE_LEVEL,1);

It is correct in both cases as the value for BLOCK_SKEW will be equal to 1 if it is disabled. Fixed in RT-CUDA v2.3.

- If BLOCK_SKEW disabled, for this case it will speedup the execution time by: 0.80/0.14

Is there any difference in generated parameters? It may be due to the different state of the underlying machine. In order to get stable results, the user can run the target function in main.c for multiple times using loop which will then converted as is by RT-CUDA.

- On Demosaic, the best configuration block is 1024. If I changed it to 1048, the result is not correct, but also didn't give me error report.

This is configuration error, block size cannot be > 1024. RT-CUDA will add error handling after kernel call to check for such errors. Fixed in RT-CUDA v2.3.

b. Inspect the CUDA generated code. For this student tries to (1) identify the mapping of the thread to the result (Th → result), (2) find whether tilting is used or not and how it is deciding on the tile size, (3) find if coalesced access is granted, (4) find whether data is copied from GM into ShM prior to computation, (5) your analysis of your choice.

(1) each thread responsible for MERGE_LEVEL*SKEW_LEVEL elements. And the mapping of result like this:

RESULT[((i+m))*N+((j+n))] = scale*SOURCE[((i+m))*N+((j+n))]

Correct.

(2) Titling is used by size MERGE_LEVEL*SKEW_LEVEL

Wrong. Tiling is used by size MERGE_LEVEL * BLOCK_SIZE

(3) The access is coallesced, they take consecutive row by using j+n

(4) No, the code is only using General Memory, because no __shared__ variable defined.

c. Carry out some slight modification:

```
__global__ void matrix_scale(float *C,float const *__restrict__ A,int scale,int N){
        int tid=threadIdx.x;
        int bid=blockIdx.x;
        int ij=bid*BLOCKSIZE+tid;
        {
                int i=(ij/N)*MERGE_LEVEL;
                int j=(ij%N);
                for(int m=0;m<MERGE_LEVEL;m++)
                C[((i+m))*N+(j)]=scale*A[((i+m))*N+(j)];

        }
}
```

code without skew will speedup the execution time by: 0.80/0.14

And also I'm trying to add the scale with number of c, and the code converting is correct.

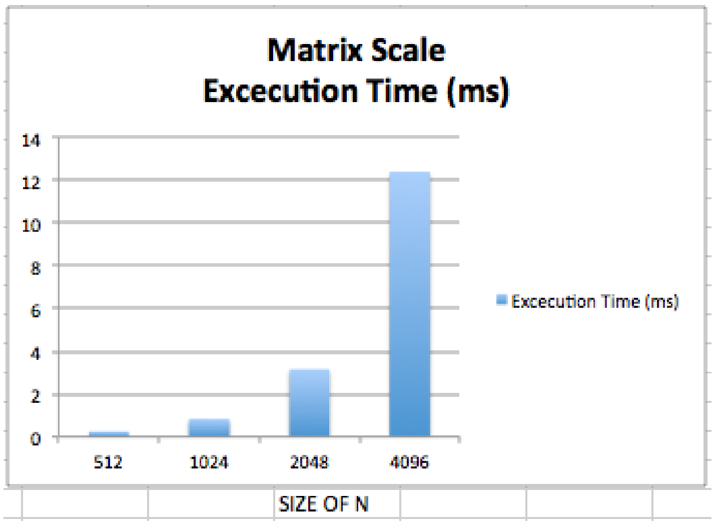
```
__global__ void matrix_scale(float *C,float const *__restrict__ A,int scale,int N){
        int c=2;
        int tid=threadIdx.x;
        int bid=blockIdx.x;
        int ij=bid*BLOCKSIZE+tid;
        {
                int i=(ij/N)*MERGE_LEVEL;
                int j=(ij%N);
                for(int m=0;m<MERGE_LEVEL;m++)
                        C[((i+m))*N+(j)]=scale*A[((i+m))*N+(j)]+c;
        }
}
```

c. **Run-time Analysis of Scalability**. Analysis of problem scalability based on run time data:

   a. For each application, compile the C source code after inspection of the configuration file, and obtains the corresponding RT-CUDA code. Run the code on GPU and collect the execution time (ET(N)). Vary the problem size as N, 1.4N, 1.8N, 2N , and 2.4N for some reasonable value of N while avoiding small values and too large values. Plot the execution time versus the problem size. The student is expected to show the ET plots versus problem size and provide careful comments on the scalability of ET(N).

Is the plotted data scales in way according to the arithmetic complexity of the problem! For example, if the arithmetic complexity of the problem is O(N^2), then ET(2N) must be >= 4*ET(N).

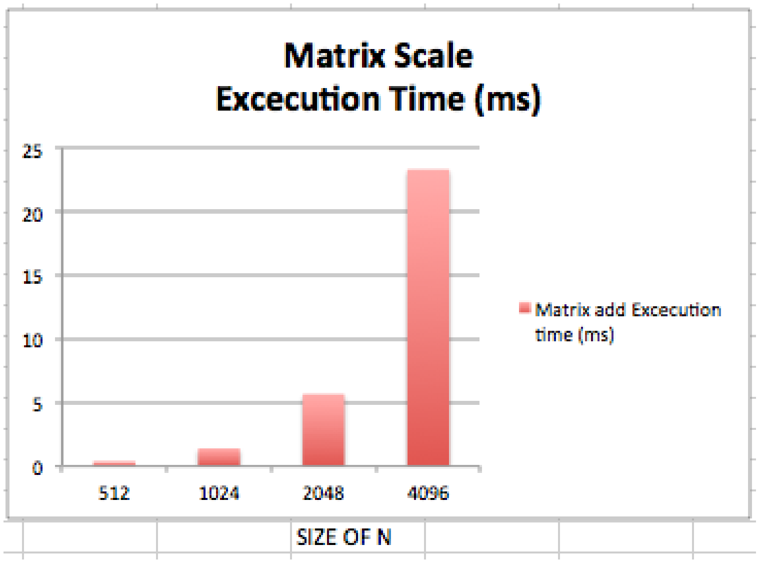
Matrix Scale:



Arithmetic complexity is O(N^2) so the ET of (2N) must be >=4*ET(N)

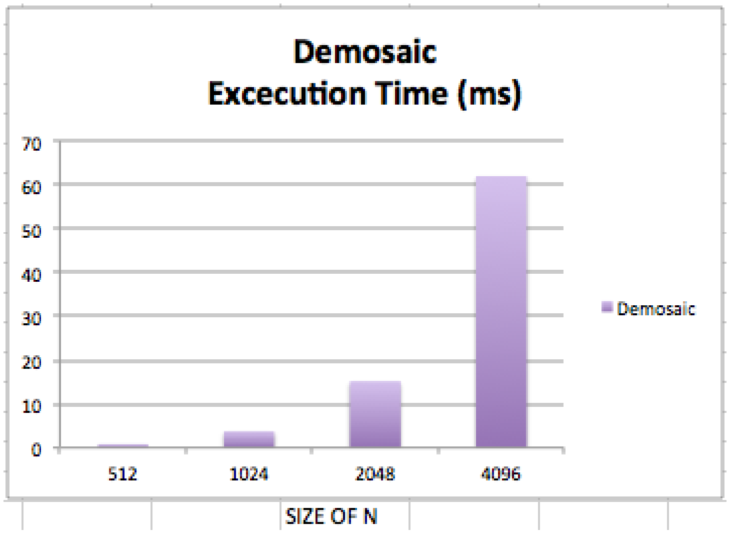
and this result shows that the scalability is good. That the result is near to 4*ET(N) for each problem size.

Matrix add:



Scalability: arithmetic complexity is O(N^2) so the ET of (2N) must be >=4*ET(N) and this result shows that the scalability is good. That the result is also near to 4*ET(N) for each problem size.

Demosaic:



Scalability: arithmetic complexity is O(N^2) so the ET of (2N) must be >=4*ET(N) and this result shows that the scalability is good. That the result is also near to 4*ET(N) for each problem size.

b. Carry out some slight modification of the source code and study its effect (execution time scalability) on the run time performance of the generated code, i.e. another test is to modify the C coding at the computed equation. For example, instead of using a vector V we may use 2V and see the impact on performance. The student is expected to show the ET plots versus problem size and link this to the ET of the original problem (before the change) and try see the impact (scalability) of the change by observing the plots.

After trying to change the code, the time is still the same. So I tried to check if the code give the correct result with this code:

```
for(i=0;i<N;i++){
            for(j=0;i<N;j++){
```

```
				if (C[i*N] != (3.0*A[i*N]))
				{
					error = 1;
					break;
				}
			}
			if (error)
			{
				break;
			}
		}
		printf("testing\n");
		if (error)
		{
			printf("Result is not correct!\n");
		}else{
			printf("Result is correct.\n");
		}
```

And turns out the result is correct.

Student 2: Histogram and Matrix-matrix multiplication (MM)

1. Histogram outputs

- Some comments on the output kernel

A. Each thread workS in some elements of the matrix A and that partition is determined by SKEW_LEVEL and MERGE_LEVEL variables.

Correct.

B. Access to the matrix is coalesced.

C. There is problem here is not taken in the consideration which is the accessing to the critical region. In other words, some threads will access to the same position of matrix A and at the end maybe the wrong result will be produced. To solve this problem it is important to make the threads access sequentially to that critical region. This compiler needs to be modified to add atomic function to handle this issue.

Good point raised. Atomic functions to be included in the future release of RT-CUDA.

Performance Evaluation

In figure (1), the X-axis represents the execution time and the Y-axis represents the size of the histogram matrix. Blue line represents the execution time versus the size of the matrix when we discard the atomic addition in the critical region. In the other hand, the red line represents that time when we used atomic addition. It is clear when we allow the threads to access the same bin sequentially to get the correct result, then the time will increase.

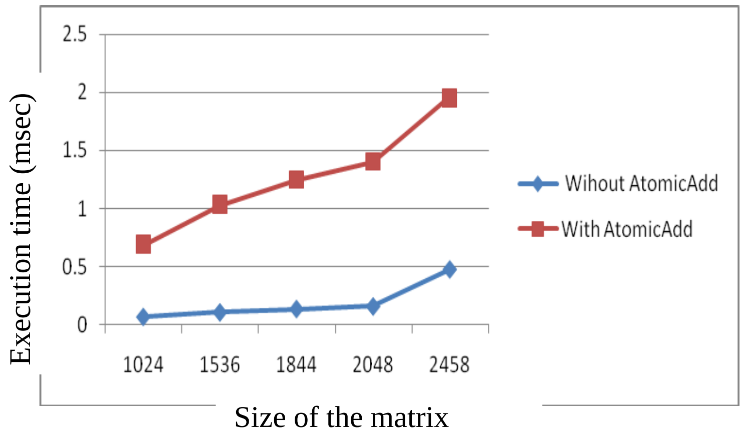
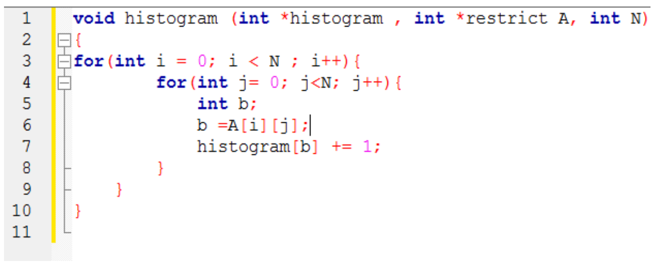


Figure (1) Execution time with consideration atomic addition and without

Also from the figure (1) it is clear that this kernel scale well.

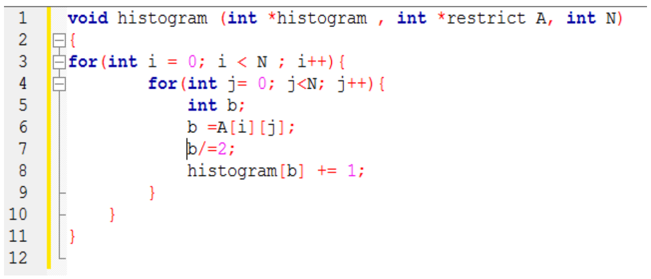
This is because atomicAdd will serialize threads that are accessing the same array index that depends on the data.

Now we will make some modification on source code and see the impact of the scalability when we compare these results with the original one. Figure (2.a) is the original source code and (2.b) is the modified source code. The modification here just is increase the number of operations. Here we divide the value b by 2. Definitely the result will be wrong but here we don't care about that. We need only know the impact of that operation on the scalability



```
void histogram (int *histogram , int *restrict A, int N)
{
for(int i = 0; i < N ; i++){
        for(int j= 0; j<N; j++){
            int b;
            b =A[i][j];
            histogram[b] += 1;
        }
    }
}
```

a. Before modification



```
void histogram (int *histogram , int *restrict A, int N)
{
for(int i = 0; i < N ; i++){
        for(int j= 0; j<N; j++){
            int b;
            b =A[i][j];
            b/=2;
            histogram[b] += 1;
        }
    }
}
```

b. After modification

Figure (2): The histogram kernel before and after modification

The parameters which produced after modification compared before those modification as follow:

1. Before

```
#define BLOCKSIZE 32

#define MERGE_LEVEL 1

#define SKEW_LEVEL 1
```

2. After

```
#define BLOCKSIZE 32

#define MERGE_LEVEL 1

#define SKEW_LEVEL 2
```

This change is not due to the modification. It is due to the different state of the underlying machine. In order to get stable results, the user can run the target function in main.c for multiple times using loop which will then converted as is by RT-CUDA.

Figure (3) shows the execution time versus the size of matrix respect to two cases (before and after modification)[Note: The logarithmic function applied on the values after modification because they are large and that make the values before modification as they were zero in the plotting]. It is clear from the figure that the simple modification consumes a lot of time to be executed because the value of b was not being stored in the share to enable us use it again but it is stored in the memory and then we need to access to it from there and this take time. This large increasing in the execution time makes the kernel not scalable.

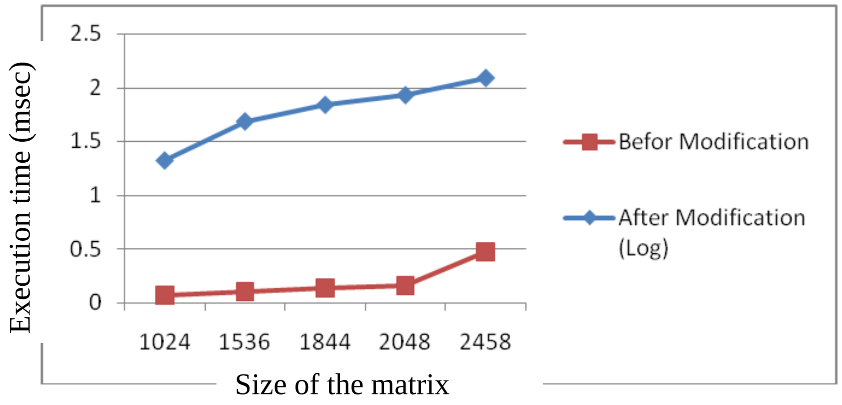


Figure (3): The Execution time of histogram kernel before and after modification

This change is not due to the modification. It is due to the incorrect optimal parameters due to different state of the underlying machine as stated in the above reponse. In order to get stable results, the user can run the target function in main.c for multiple times using loop which will then converted as is by RT-CUDA.

2. Matrix-Matrix Multiplication (MM) outputs

- Some comments on the output kernel

A. Each thread work on block of the result matrix and the size of that block depends on MERGE_LEVEL and BLOCK_SIZE variables, where MERGE_LEVEL determines how many columns in that block but the number of the rows in that block is determined by BLOCK_SIZE.

Correct.

B. The matrix A is divided to tiles and the shared memory is used to store those tiles of matrix A.

Correct.

C. matrix B is accessed from the GM and it wasn't being tiled. To optimize this output, the compiler needs to be modified also to tile the matrix be to decrease the overhead.

Currently RT-CUDA supports tiling on any one matrix. It is need to be done in future releases of RT-CUDA.

Performance Evaluation:

In this experiment I want to test the scalability of MM kernel using different sizes of the used matrices.

Figure (4,5) show the execution time for MM multiplication when different sizes are used. I used 1024, 1536, 1844, 2048, and 2458 this is as if it N, 1.4N, 1.8N, 2N, and 2.4N.

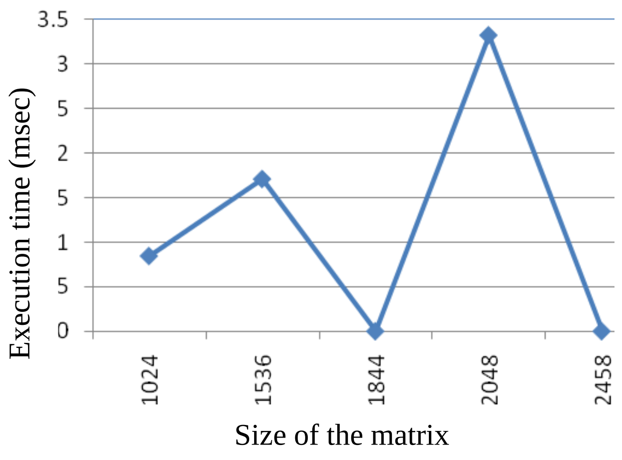


Figure (4): The Execution time of MM kernel for different matrix size.

In figure (4) it is clear that the execution time is zero when some sizes are used but if we check these size values we will fine that these values are not multiple of 32 (Warp size). That means there is a limit in the software and we can only work with N which is multiple of 32 (Size of warp). To proof this, in figure (5) I did another experiment since all the values are multiple of 32.

This is correct. In order to use matrices which are not multiple of 32, we need to use padding rows/columns in the original matrix with zeros. Alternatively, user the use RT-CUDA API functions for such matrices.

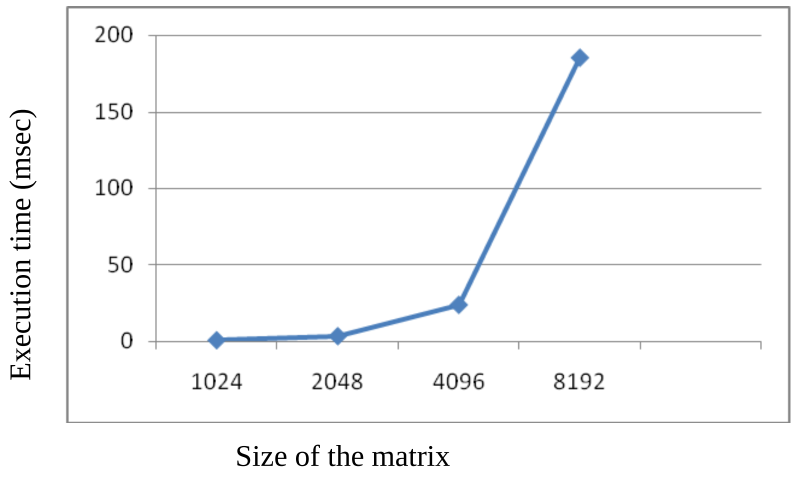


Figure (5): The Execution time of MM kernel for different matrix size .

Table (1) shows the exact values of execution time of MM multiplication kernel, since the figure (5) depends on these values. Here we show this table to verify if the execution time of size 2N is larger than eight times of execution time of N (i.e. is ET(2N) >= 8*ET(N)?).

From the table, it is clear that ET(2N) >= 8*ET(N)). Also it is clear that when N goes largely up then the relation will be ET(2N) >> = 4*ET(N))

N	ET
1024	0.847904
2048	3.532864
4096	23.973152
8192	185.398819

Table (1): Execution time (ET) against size of matrix for MM Mult. kernel

Now we will check if the produced parameters given in params.h file is optimal value or no.

To verify that, we will change the BLOCK_SIZE to 32, 128, 256 and 1024. Then at each case we use different size for N as 1024, 2048, 4096, and 8192. Table (2) shows the execution time against different size of in when the BLOCK-SIZE varies. It is clear from the table that no big difference when 128, 256, 512, or 1024 threads/block/ (Excluding 32 threads/block which give highest execution time) are used but the best one is when 1024 thread/block is used. It is worthy to mention that the value of BLOCK-SIZE in the params.h produced after compiling was 512. That mean this value is not the optimal value.

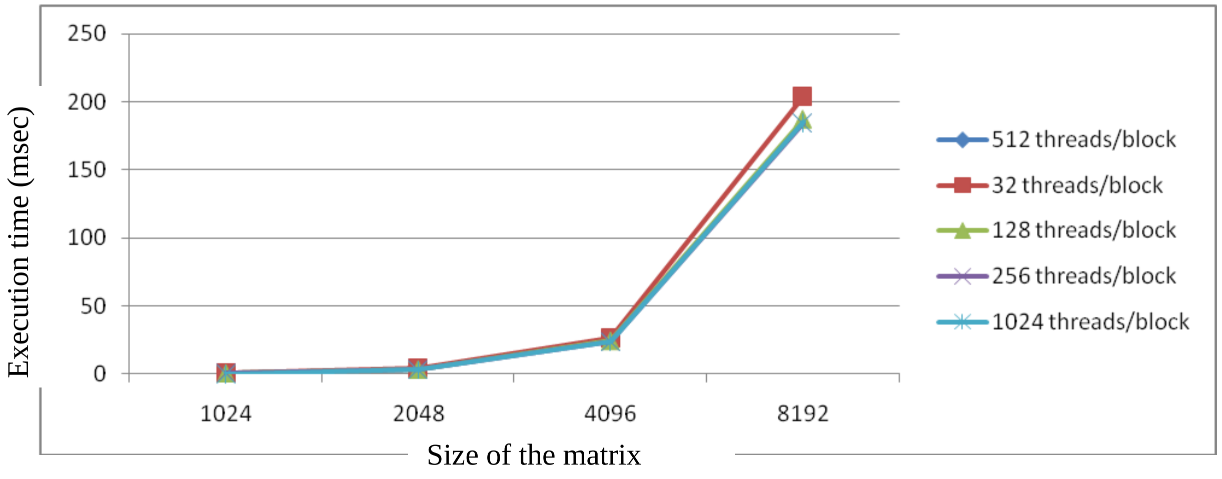
Tabel (2)

N	32 threads/block	128 threads/block	256 threads/block	512 threads/block	1024 threads/block
1024	0.960896	0.78368	0.850368	0.847904	0.034
2048	4.811584	3.6368	3.445408	3.532864	3.316
4096	26.511072	24.681408	24.009312	23.973152	23.585

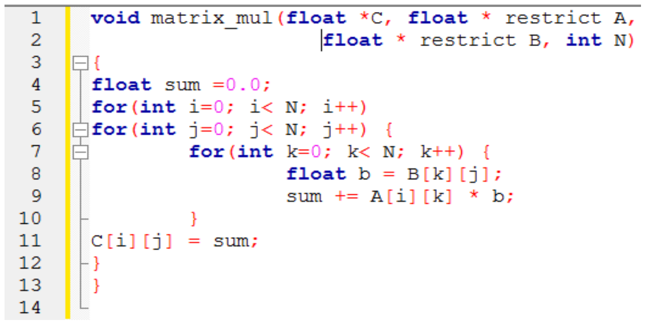
8192	203.913025	187.436386	184.410782	185.398819	184.997

It is due to the incorrect optimal parameters due to different state of the underlying machine as stated in the above response. In order to get stable results, the user can run the target function in main.c for multiple times using loop which will then converted as is by RT-CUDA.

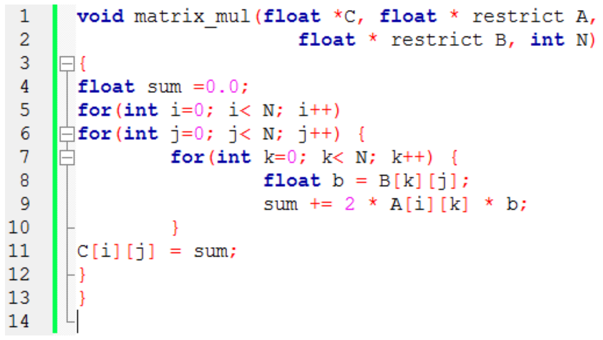
Figure (6) shows the plotting of the values in table (2) and it is clear that no big difference when 128, 256, 512, or 1024 threads per block with respect to the execution time but this time is large when 32 thread/block is used.



Now we will make some modification of the source code (C code) of the MM multiplication to see the how the output will be modified and then to test the scalability of the kernel by comparing the execution time before and after modification. Figures 7.a &7.b show the source c code before and after modification respectively.



```
void matrix_mul(float *C, float * restrict A,
                     float * restrict B, int N)
{
float sum =0.0;
for(int i=0; i< N; i++)
for(int j=0; j< N; j++) {
          for(int k=0; k< N; k++) {
                    float b = B[k][j];
                    sum += A[i][k] * b;
          }
C[i][j] = sum;
}
}
```



```
void matrix_mul(float *C, float * restrict A,
                   float * restrict B, int N)
{
float sum =0.0;
for(int i=0; i< N; i++)
for(int j=0; j< N; j++) {
        for(int k=0; k< N; k++) {
                float b = B[k][j];
                sum += 2 * A[i][k] * b;
        }
C[i][j] = sum;
}
}
```

Figure (7.a) : MM multi. before modifi. Figure (7.a) : MM multi. after modifi.

The parameters which produced after modification (params.h) compared before those modification as follow:

1. Before

```
#define BLOCKSIZE 512
#define MERGE_LEVEL 4
#define SKEW_LEVEL 215
```

2. After

```
#define BLOCKSIZE 32
#define MERGE_LEVEL 1
#define SKEW_LEVEL 32
```

This seems to be incorrect argument as SKEW_LEVEL cannot be equal to 215 because RT-CUDA evaluate SKEW_LEVEL as multiple of 2.

Table (3) show the execution time of kernel before and after modification. It is clear that a simple modification on source kernel give us big difference respect to time execution of the output kernel

Table (3)

N	Before modification	After modification
1024	0.847904	23.275
2048	3.532864	182.065

4096	23.973152	1431.377
8192	185.398819	11302.358

The reason behind this sudden increasing is the values of the parameters produced in params.h

Now manually we will change those parameters to see the effete on the execution time.

The parameters will change to be as follow:

```
#define BLOCKSIZE 512

#define MERGE_LEVEL 4

#define SKEW_LEVEL 215
```

Actually these parameters are the same those produced before modification.

As it shown in the table (4), fourth column (Named "After modification (with different parameters)") shows the execution time of the kernel after modification but here also after parameters are changed manually. It is clear that there is big difference comparing with the execution time shown in third column (when the original parameters produced by the compiler are used) and this means that the compiler did not give us the optimal values for the those parameters.

N	Before modification	After modification	After modification (with different parameters)
1024	0.847904	23.275	1.056096
2048	3.532864	182.065	6.749216
4096	23.973152	1431.377	53.990143
8192	185.398819	11302.358	424.355377

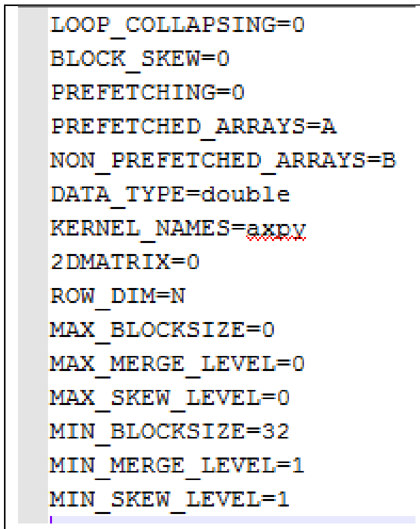
Again, it is due to the incorrect optimal parameters due to different state of the underlying machine as stated in the above response. In order to get stable results, the user can run the target function in main.c for multiple times using loop which will then converted as is by RT-CUDA.

Student 3: AXPY and Jacobi iterative solver

<u>Analysis and Inspection of the generated CUDA code</u>

<u>AXPY:</u>

a. Inspect the configuration file and try linking its information with the nature of the application as a user hint to guide the compiler.

 ```LOOP_COLLAPSING=0 BLOCK_SKEW=0 PREFETCHING=0 PREFETCHED_ARRAYS=A NON_PREFETCHED_ARRAYS=B DATA_TYPE=double KERNEL_NAMES=axpy 2DMATRIX=0 ROW_DIM=N MAX_BLOCKSIZE=0 MAX_MERGE_LEVEL=0 MAX_SKEW_LEVEL=0 MIN_BLOCKSIZE=32 MIN_MERGE_LEVEL=1 MIN_SKEW_LEVEL=1```	1) **LOOP_COLLAPSING** and **BLOCK_SKEW** and **2DMATRIX** all are equal zero (0) because kernel has 1D resultant. Furthermore the value of **PREFETCHING** = 0 because the kernel has **NO** 2D matrix in the computation. 2) Only the A matrix need to be pre fetched. 3) **MAX_BLOCKSIZE**, and **MAX_MERGE_LEVEL** are equal to zero, this will allow the compiler to check all possibilities and select the best values for the BLOCK_SIZE and MERGE_LEVEL parameters
AXPY config file	

b. Inspect the CUDA generated code:

(1) Identify the mapping of the thread to the result (Th → result),

Since the parameter MERGE_LEVEL has the value 1, each thread th computes one element in the final result, i.e., thread Th(B_{id} , T_{id}) computes the value at B_{id}*BLOCKSIZE+T_{id} in the final result vector C.

(2) Find whether tilting is used or not and how it is deciding on the tile size.

Yes, the two vectors A and C is tiled into tiles of size BLOCKSIZE. Threads within a block are computing partial result of one tile.

Wrong. This is not tiled as this problem is 1D and tiling only works with 2D using prefetching enabled.

(3) find if coalesced access is granted,

Since consecutive threads within a block are accessing consecutive elements in the global memory, coalesced access is not granted.

Incorrect Answer. If consecutive threads within a block are accessing consecutive elements then the coalesced access is guaranteed.

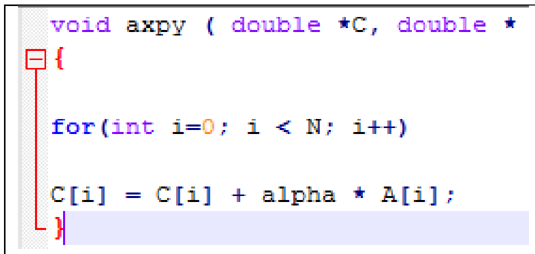
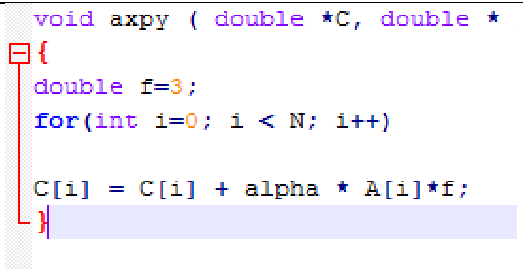
(4) Find whether data is copied from GM into ShM prior to computation.

No, data are copied from GM into ShM. All computations are performed in the global memory

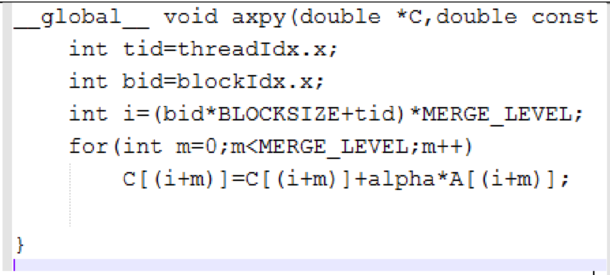
Correct. There is no sharing of data so no shared memory usage required. Although, it will use data cache.

c. Carry out some slight modification of the source code and study its effect on the generated CUDA code:

<u>Modification on the C code:</u>

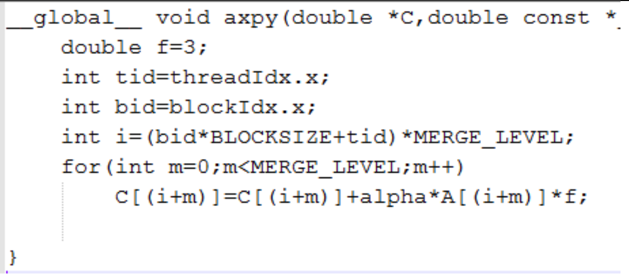
 `void axpy ( double *C, double *` `{` `for(int i=0; i < N; i++)` `C[i] = C[i] + alpha * A[i];` `}`	 `void axpy ( double *C, double *` `{` `double f=3;` `for(int i=0; i < N; i++)` `C[i] = C[i] + alpha * A[i]*f;` `}`
AXPY	AXPY after modification

<u>Results of this modification on the generated CUDA code</u>



```
__global__ void axpy(double *C,double const
    int tid=threadIdx.x;
    int bid=blockIdx.x;
    int i=(bid*BLOCKSIZE+tid)*MERGE_LEVEL;
    for(int m=0;m<MERGE_LEVEL;m++)
        C[(i+m)]=C[(i+m)]+alpha*A[(i+m)];

}
```

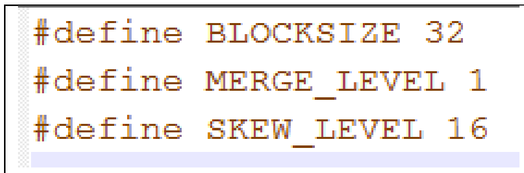
Generated AXPY original CUDA kernel



```
__global__ void axpy(double *C,double const *
    double f=3;
    int tid=threadIdx.x;
    int bid=blockIdx.x;
    int i=(bid*BLOCKSIZE+tid)*MERGE_LEVEL;
    for(int m=0;m<MERGE_LEVEL;m++)
        C[(i+m)]=C[(i+m)]+alpha*A[(i+m)]*f;

}
```

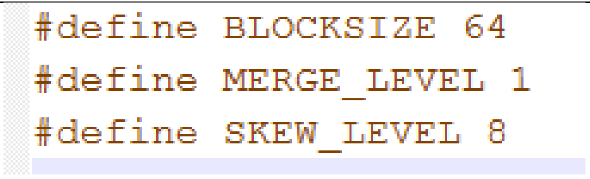
Generated AXPY CUDA kernel after modification

<u>This modification affects the parameters as the following:</u>



```
#define BLOCKSIZE 32
#define MERGE_LEVEL 1
#define SKEW_LEVEL 16
```

Generated AXPY params file



```
#define BLOCKSIZE 64
#define MERGE_LEVEL 1
#define SKEW_LEVEL 8
```

Generated AXPY params file after modification

After modified the c code, the resultant CUDA code is modified consequentially, the compiler selects different optimal values for both BLOKSIZE and SKEW_LEVEL

The change in parameters here may be due to the increased number of floating point operations or different state of underlying machine. In order to get stable results, the user can run the target function in main.c for multiple times using loop which will then converted as is by RT-CUDA.

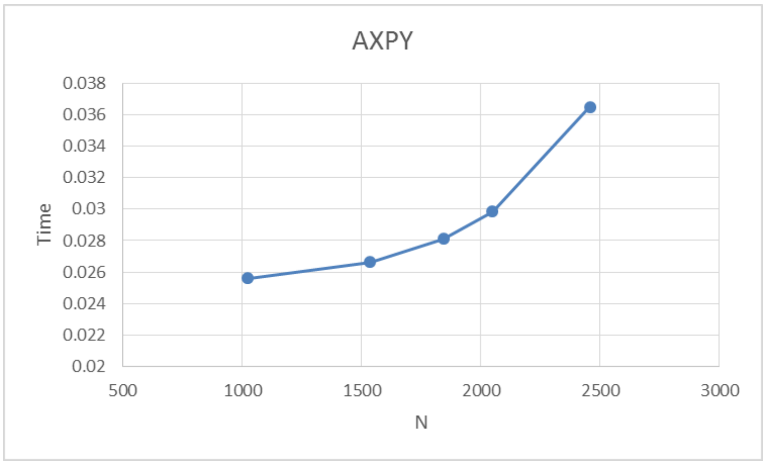
Run-time Analysis of Scalability. Analysis of problem scalability based on run time data:

a. For each application, compile the C source code after inspection of the configuration file, and obtains the corresponding RT-CUDA code. Run the code on GPU and collect the execution time (ET(N)). Vary the problem size as N, 1.4N, 1.8N, 2N , and 2.4N for some reasonable value of N while avoiding small values and too large values. Plot the execution time versus the problem size.

The student is expected to show the ET plots versus problem size and provide careful comments on the scalability of ET(N). Is the plotted data scales in way according to the arithmetic complexity of the problem! For example, if the arithmetic complexity of the problem is O(N^2), then ET(2N) must be >= 4*ET(N).

AXPY Results

AXPY	
N	Time
1024	0.0256
1536	0.026624
1844	0.028096
2048	0.029856
2458	0.03648

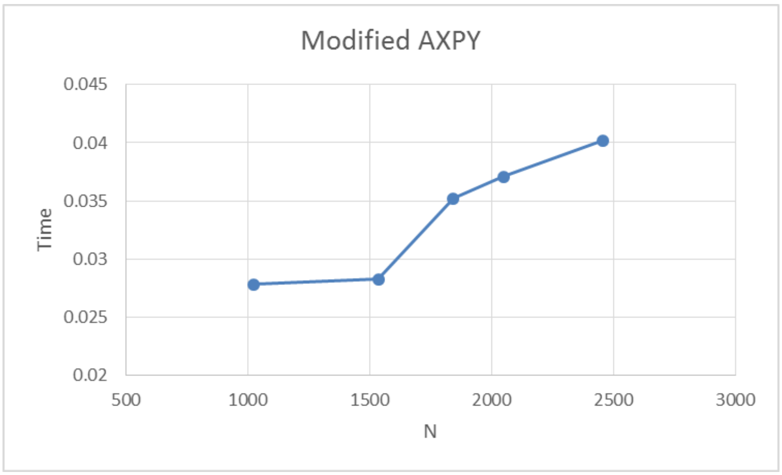


The complexity of AXPY is O (N). However, when the size N is duplicated (from 1024 to 2048) the execution time increased by around 16%. This indicates that the parameters is well tuned therefore, incrementing the size has little effect on the execution time.

This little increase is just due to the overhead of thread blocks (TB) scheduling not actuallu the increase in N. As per the optimal parameters, for N=1024 the TB=2 while for N=2048 the TB=4 which still not fully utilize the GPU resources.

b. Carry out some slight modification of the source code and study its effect (execution time scalability) on the run time performance of the generated code, i.e. another test is to modify the C coding at the computed equation. For example, instead of using a vector V we may use 2V and see the impact on performance. The student is expected to show the ET plots versus problem size and link this to the ET of the original problem (before the change) and try see the impact (scalability) of the change by observing the plots.

Modified AXPY Results

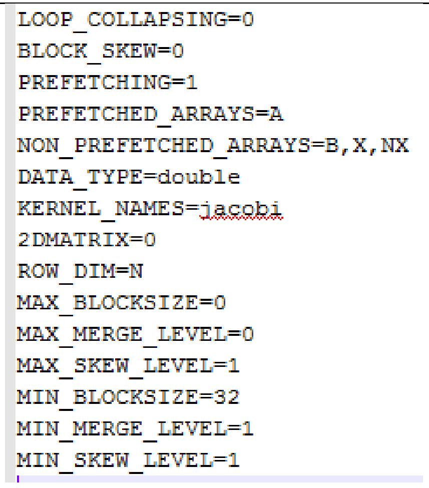


Modified AXPY	
N	Time
1024	0.027776
1536	0.028256
1844	0.0352
2048	0.037088
2458	0.040192

The slightly modification on the original code has insignificant change in the execution time comparing with the original code, this indicates that the compiler selects suitable parameters to handle the modification on the code

JACOBI:

a. Inspect the configuration file and try linking its information with the nature of the application as a user hint to guide the compiler.


```
LOOP_COLLAPSING=0
BLOCK_SKEW=0
PREFETCHING=1
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B,X,NX
DATA_TYPE=double
KERNEL_NAMES=jacobi
2DMATRIX=0
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=1
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```
	1) **LOOP_COLLAPSING** and **BLOCK_SKEW** and **2DMATRIX** all are equal zero (0) because kernel has 1D resultant.
	2) The **PREFETCHING** = 1 because the kernel has 2D matrix in the computation.
	3) Only the A matrix need to be pre fetched.
	4) **MAX_BLOCKSIZE**, and **MAX_MERGE_LEVEL**= 0 this will allow the compiler to check all possibilities and select the best values for the BLOCK_SIZE and MERGE_LEVEL parameters

JACOBI config file	

b. Inspect the CUDA generated code:

(1) Identify the mapping of the thread to the result (Th → result),

Since the parameter MERGE_LEVE has the value 1, each thread th computes one element in the final result, i.e., thread Th(B_{id} , T_{id}) computes the value at B_{id}*BLOCKSIZE+T_{id} in the final result vector NX.

(2) Find whether tilting is used or not and how it is deciding on the tile size.

Yes, the matrix A is tiled into tiles of size BLOCKSIZE * BLOCKSIZE. Threads within a block are cooperating reading the elements from the tile in global memory to shared memory (**each thread reads one element within the tile**) and computing the partial sum in the shared memory.

Wrong. A is tiled by MERGE_LEVEL * BLOCK_SIZE

(3) Find if coalesced access is granted,

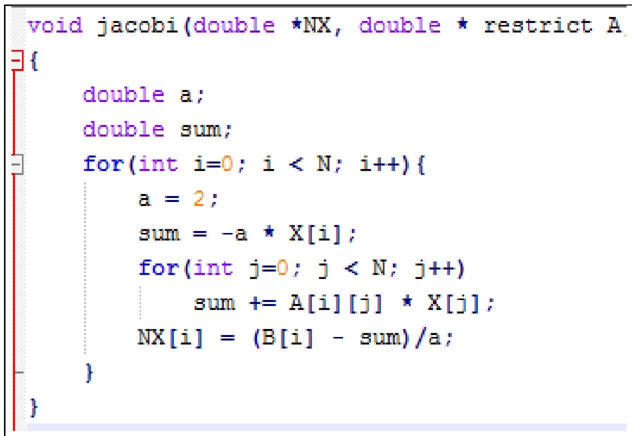
Coalesced access is not granted, because consecutive threads within a block are accessing non-consecutive elements in the global memory.

Incorrect Answer. Here, consecutive threads within a block are accessing consecutive elements in GM while loading the tile from GM to ShM.

(4) Find whether data is copied from GM into ShM prior to computation.

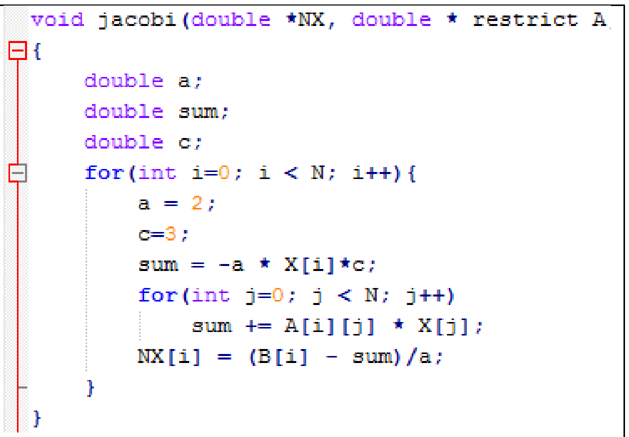
Yes, data are copied from GM into ShM prior to computation. The kernel defines a shared matrix named As and copies data from the tile into it.

a. Carry out some slight modification of the source code and study its effect on the generated CUDA code:



```
void jacobi(double *NX, double * restrict A
{
    double a;
    double sum;
    for(int i=0; i < N; i++){
        a = 2;
        sum = -a * X[i];
        for(int j=0; j < N; j++)
            sum += A[i][j] * X[j];
        NX[i] = (B[i] - sum)/a;
    }
}
```

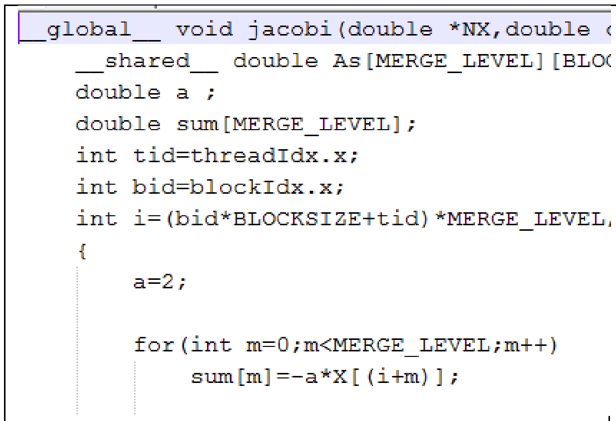
Jacobi



```
void jacobi(double *NX, double * restrict A
{
    double a;
    double sum;
    double c;
    for(int i=0; i < N; i++){
        a = 2;
        c=3;
        sum = -a * X[i]*c;
        for(int j=0; j < N; j++)
            sum += A[i][j] * X[j];
        NX[i] = (B[i] - sum)/a;
    }
}
```

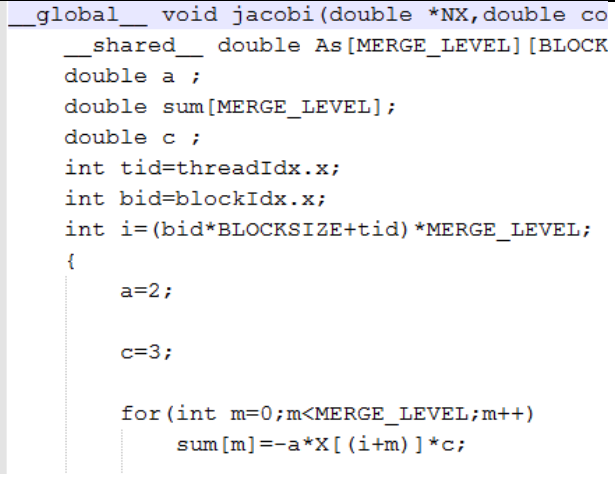
Jacobi After modification

Results of this modification on the generated CUDA code



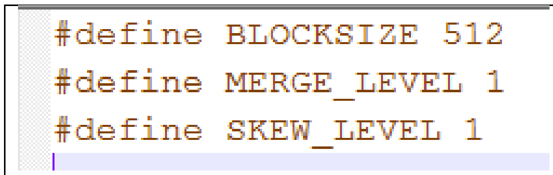
```
__global__  void jacobi(double *NX,double
    __shared__  double As[MERGE_LEVEL][BLO
    double a ;
    double sum[MERGE_LEVEL];
    int tid=threadIdx.x;
    int bid=blockIdx.x;
    int i=(bid*BLOCKSIZE+tid)*MERGE_LEVEL
    {
        a=2;

        for(int m=0;m<MERGE_LEVEL;m++)
            sum[m]=-a*X[(i+m)];
```

Generated Jacobi original CUDA kernel



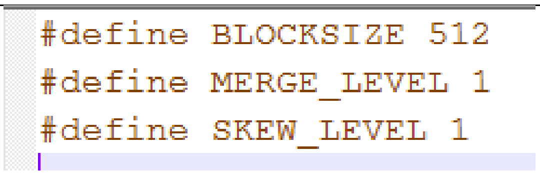
```
__global__  void jacobi(double *NX,double co
    __shared__  double As[MERGE_LEVEL][BLOCK
    double a ;
    double sum[MERGE_LEVEL];
    double c ;
    int tid=threadIdx.x;
    int bid=blockIdx.x;
    int i=(bid*BLOCKSIZE+tid)*MERGE_LEVEL;
    {
        a=2;

        c=3;

        for(int m=0;m<MERGE_LEVEL;m++)
            sum[m]=-a*X[(i+m)]*c;
```

Generated Jacobi CUDA kernel after modification



```
#define BLOCKSIZE 512
#define MERGE_LEVEL 1
#define SKEW_LEVEL 1
```

Generated Jacobi params file



```
#define BLOCKSIZE 512
#define MERGE_LEVEL 1
#define SKEW_LEVEL 1
```

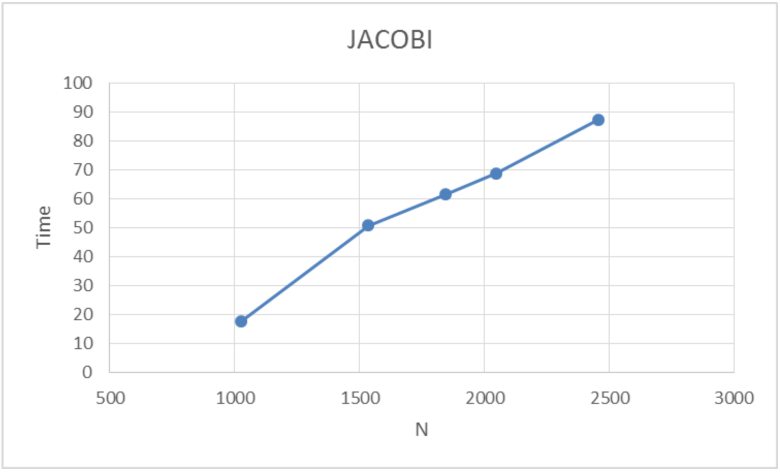
Generated Jacobi params file after modification

The modification on the code has no effect on the parameters for Jacobi solver

Run-time Analysis of Scalability. Analysis of problem scalability based on run time data:

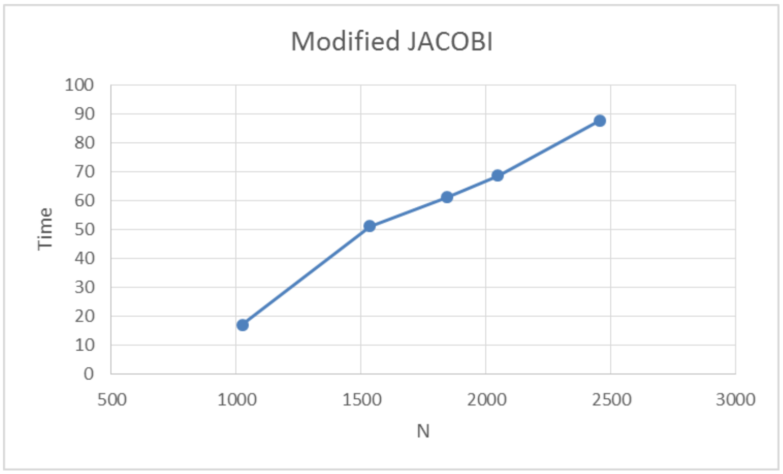
Jacobi Results:

Jacobi	
N	time
1024	17.71606
1536	50.73251
1844	61.60954
2048	68.81725
2458	87.46851



For Jacobi solver, when the size N is duplicated (from 1024 to 2048) the execution ET(2N) becomes around 4*ET(N). This indicates that the generated code is scaled well with the problem size.

<u>Modified Jacobi Results:</u>



Modified Jacobi	
N	Time
1024	17.14266
1536	51.25802
1844	61.09395
2048	68.65597
2458	87.74966

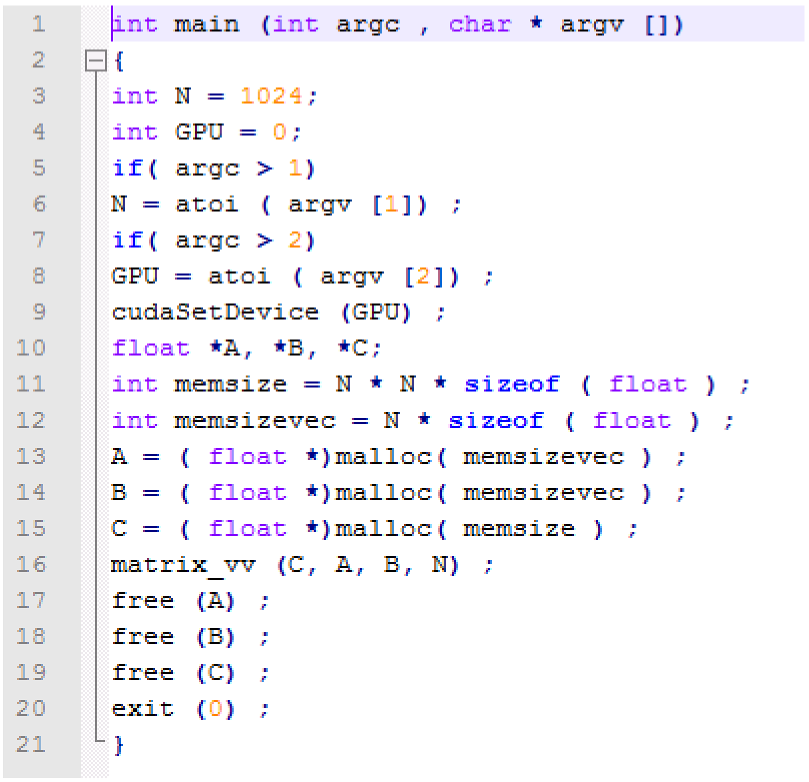
The slightly modification on the original code for Jacobi solver has insignificant change in the execution time comparing with the original code, this indicates that the parameters are chosen well by the compiler to handle the modification on the code.

Student 4: Vector inner product (VV) and Conjugate Gradient (CG)

Q1

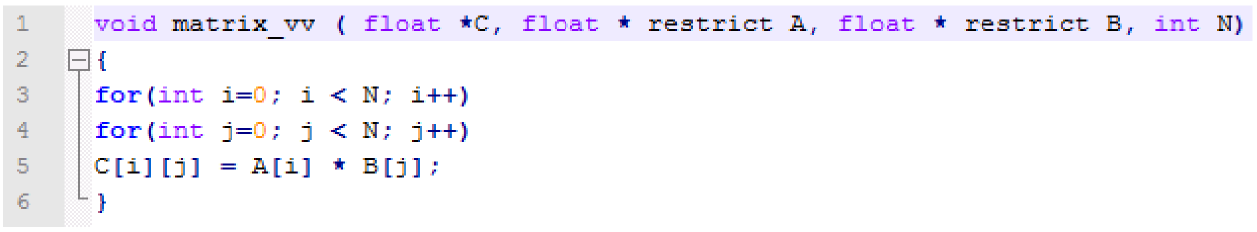
1. This part is related to Vector-vector multiplication inner product (VV)

   a. Here is the main.c code as input



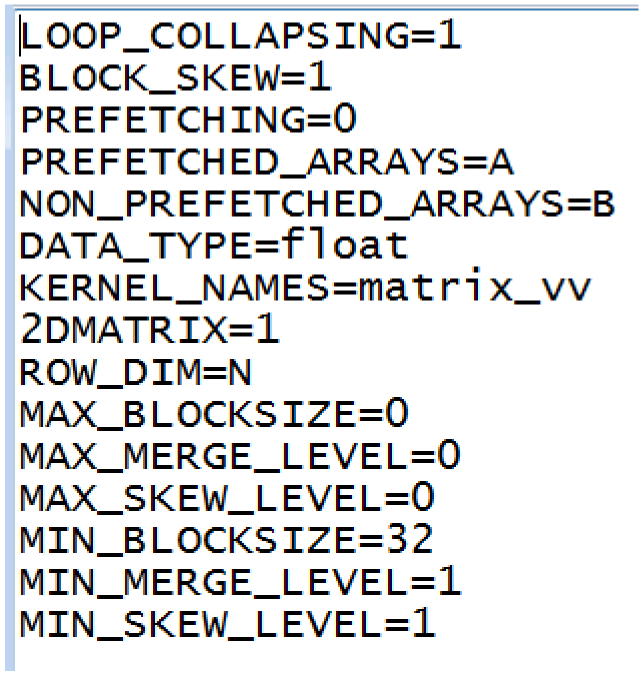
```
int main (int argc , char * argv [])
{
int N = 1024;
int GPU = 0;
if( argc > 1)
N = atoi ( argv [1]) ;
if( argc > 2)
GPU = atoi ( argv [2]) ;
cudaSetDevice (GPU) ;
float *A, *B, *C;
int memsize = N * N * sizeof ( float ) ;
int memsizevec = N * sizeof ( float ) ;
A = ( float *)malloc( memsizevec ) ;
B = ( float *)malloc( memsizevec ) ;
C = ( float *)malloc( memsize ) ;
matrix_vv (C, A, B, N) ;
free (A) ;
free (B) ;
free (C) ;
exit (0) ;
}
```

Here is the kernel as input



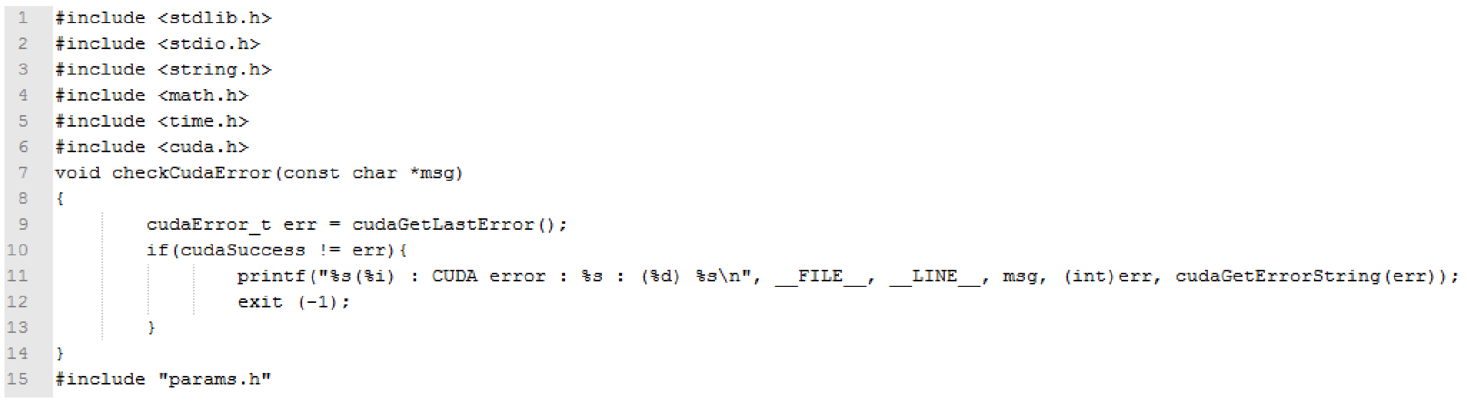
```
void matrix_vv ( float *C, float * restrict A, float * restrict B, int N)
{
for(int i=0; i < N; i++)
for(int j=0; j < N; j++)
C[i][j] = A[i] * B[j];
}
```

And here is the config file

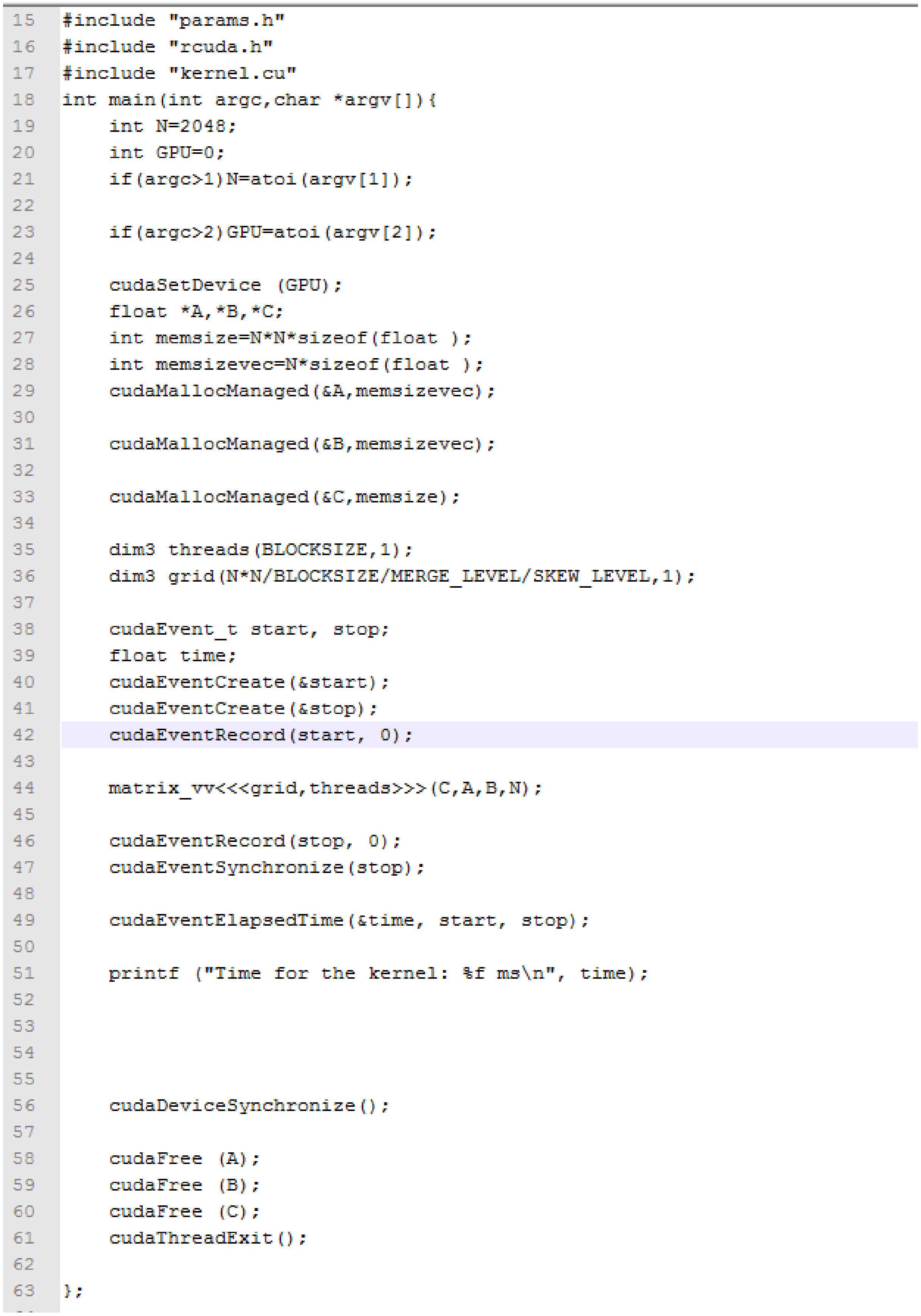


```
LOOP_COLLAPSING=1
BLOCK_SKEW=1
PREFETCHING=0
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B
DATA_TYPE=float
KERNEL_NAMES=matrix_vv
2DMATRIX=1
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=0
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Here is the output files

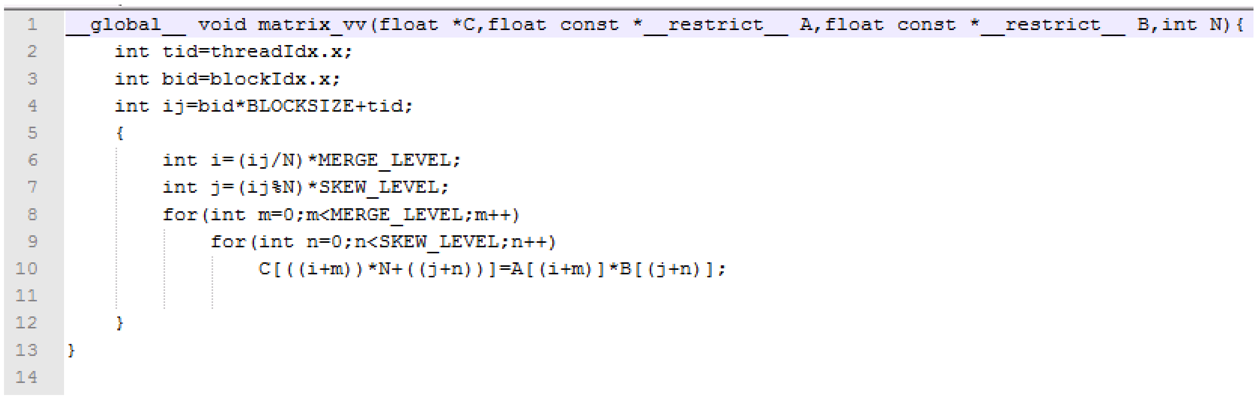


```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %s\n", __FILE__, __LINE__, msg, (int)err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
```



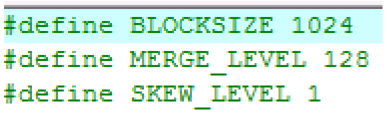
```
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc,char *argv[]){
    int N=2048;
    int GPU=0;
    if(argc>1)N=atoi(argv[1]);

    if(argc>2)GPU=atoi(argv[2]);

    cudaSetDevice (GPU);
    float *A,*B,*C;
    int memsize=N*N*sizeof(float );
    int memsizevec=N*sizeof(float );
    cudaMallocManaged(&A,memsizevec);

    cudaMallocManaged(&B,memsizevec);

    cudaMallocManaged(&C,memsize);

    dim3 threads(BLOCKSIZE,1);
    dim3 grid(N*N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1);

    cudaEvent_t start, stop;
    float time;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    cudaEventRecord(start, 0);

    matrix_vv<<<grid,threads>>>(C,A,B,N);

    cudaEventRecord(stop, 0);
    cudaEventSynchronize(stop);

    cudaEventElapsedTime(&time, start, stop);

    printf ("Time for the kernel: %f ms\n", time);




    cudaDeviceSynchronize();

    cudaFree (A);
    cudaFree (B);
    cudaFree (C);
    cudaThreadExit();

};
```

Here is the kernel file



```
__global__ void matrix_vv(float *C,float const *__restrict__ A,float const *__restrict__ B,int N){
    int tid=threadIdx.x;
    int bid=blockIdx.x;
    int ij=bid*BLOCKSIZE+tid;
    {
        int i=(ij/N)*MERGE_LEVEL;
        int j=(ij%N)*SKEW_LEVEL;
        for(int m=0;m<MERGE_LEVEL;m++)
            for(int n=0;n<SKEW_LEVEL;n++)
                C[((i+m))*N+((j+n))]=A[(i+m)]*B[(j+n)];

    }
}

```

And here is the param.h



```
#define BLOCKSIZE 1024
#define MERGE_LEVEL 128
#define SKEW_LEVEL 1
```

And here some snapshot after running for different values of N. (I tired N = 1023 but it didn't work)

This is correct. In order to use matrices which are not multiple of 32, we need to use padding rows/columns in the original matrix with zeros. Alternatively, user the use RT-CUDA API functions for such matrices.

b. At VV there is no need to prefetching as the operation will done once then the result will store back, so prefetching is set to zero. 2DMatrix is enabled to 1 because one of vectors must be transposed.

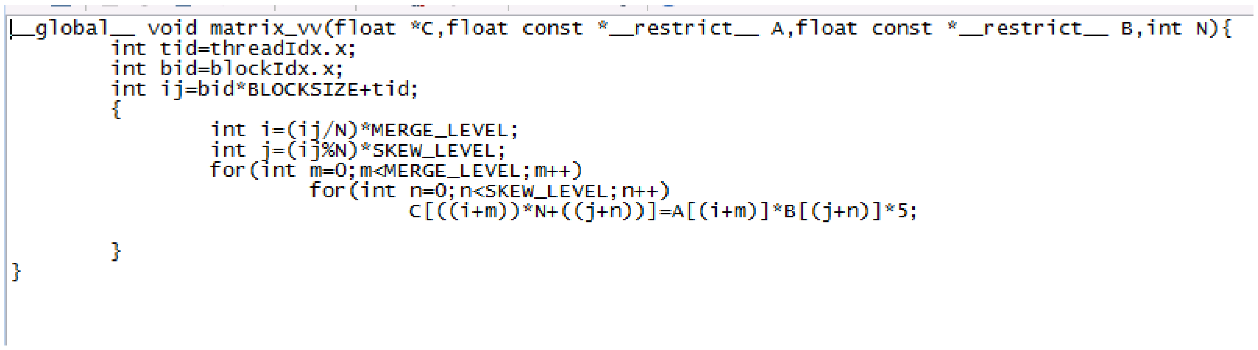
```
int tid=threadIdx.x;
        int bid=blockIdx.x;
        int ij=bid*BLOCKSIZE+tid;
        {
                int i=(ij/N)*MERGE_LEVEL;
                int j=(ij%N)*SKEW_LEVEL;
                for(int m=0;m<MERGE_LEVEL;m++)
```

for(int n=0;n<SKEW_LEVEL;n++)

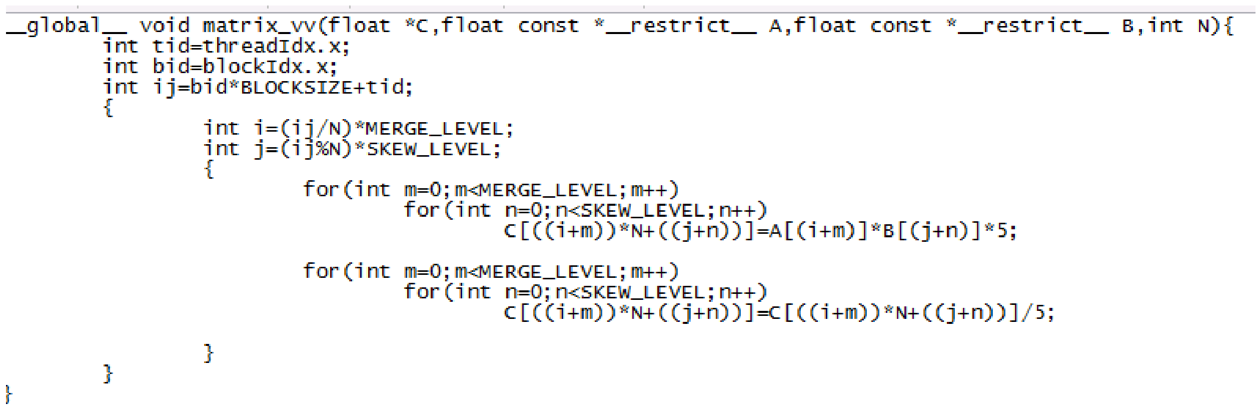
C[((i+m))*N+((j+n))]=A[(i+m)]*B[(j+n)];

Here the code of the kernel so it is taking the merge level as 128 while the skew is one and the block size is 1024, sure the code here get use of the coalesced.

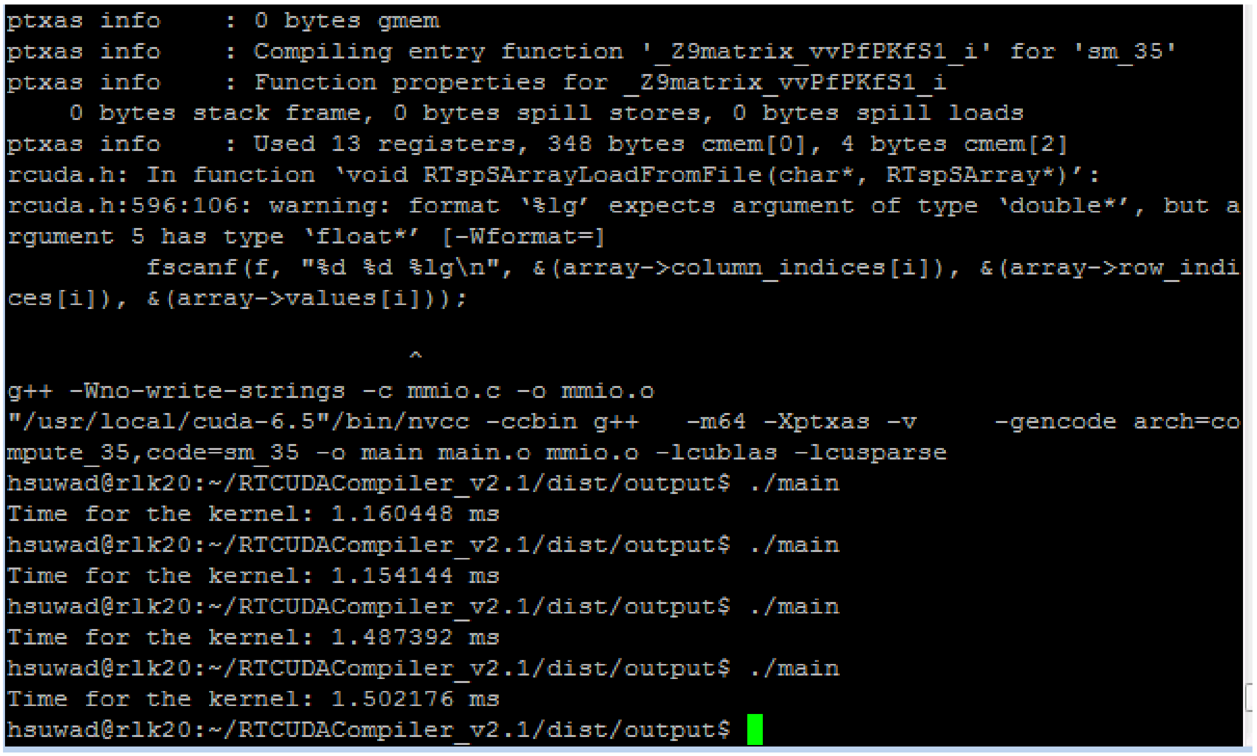
Here the grid is (blocksize,threadID), I relative and affected by merge size and j is relative to skew which is one here the tiling used as block in one dimension 128 * 1 . Yes there is copying of data and there is a synchronization point


```
__global__ void matrix_vv(float *C,float const *__restrict__ A,float const *__restrict__ B,int N){
        int tid=threadIdx.x;
        int bid=blockIdx.x;
        int ij=bid*BLOCKSIZE+tid;
        {
                int i=(ij/N)*MERGE_LEVEL;
                int j=(ij%N)*SKEW_LEVEL;
                for(int m=0;m<MERGE_LEVEL;m++)
                        for(int n=0;n<SKEW_LEVEL;n++)
                                C[((i+m))*N+((j+n))]=A[(i+m)]*B[(j+n)]*5;

        }
}
```

I multiply the element by 5 here .. it is shown in kernel the effect of this is very slightly,,, and the generated code is perfect here and I also try to divided the result by 5 as shown ,, and the result looks like perfect.

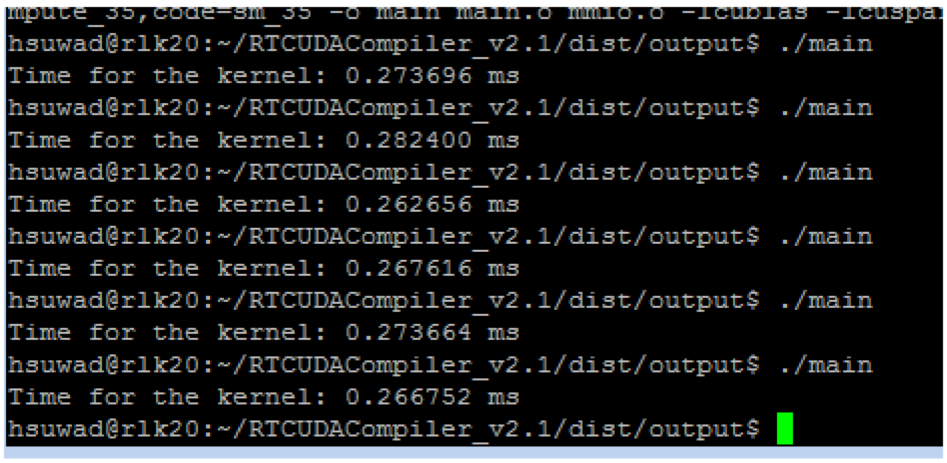

```
__global__ void matrix_vv(float *C,float const *__restrict__ A,float const *__restrict__ B,int N){
        int tid=threadIdx.x;
        int bid=blockIdx.x;
        int ij=bid*BLOCKSIZE+tid;
        {
                int i=(ij/N)*MERGE_LEVEL;
                int j=(ij%N)*SKEW_LEVEL;
                {
                        for(int m=0;m<MERGE_LEVEL;m++)
                                for(int n=0;n<SKEW_LEVEL;n++)
                                        C[((i+m))*N+((j+n))]=A[(i+m)]*B[(j+n)]*5;

                        for(int m=0;m<MERGE_LEVEL;m++)
                                for(int n=0;n<SKEW_LEVEL;n++)
                                        C[((i+m))*N+((j+n))]=C[((i+m))*N+((j+n))]/5;

                }
        }
}
```

Here it is considered as two loops and shows a good result get benefits from data locality (alittle affect from less than 1 to above 1 a little),, but every one of the equation considered to be in one loop even it could be written in one loop.



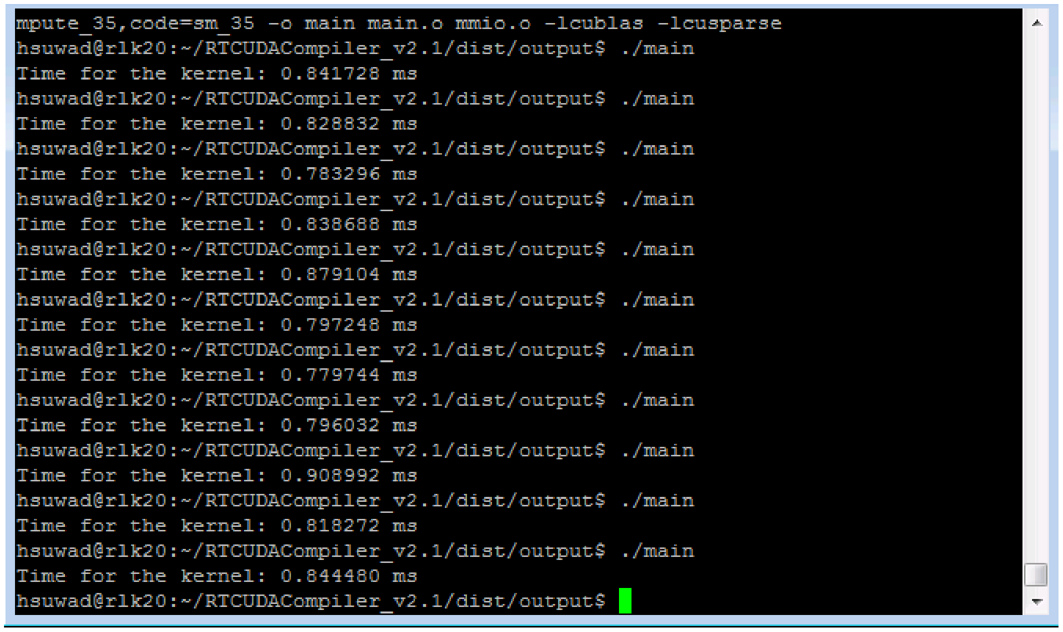
```
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z9matrix_vvPfPKfS1_i' for 'sm_35'
ptxas info    : Function properties for _Z9matrix_vvPfPKfS1_i
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 13 registers, 348 bytes cmem[0], 4 bytes cmem[2]
rcuda.h: In function 'void RTspSArrayLoadFromFile(char*, RTspSArray*)':
rcuda.h:596:106: warning: format '%lg' expects argument of type 'double*', but a
rgument 5 has type 'float*' [-Wformat=]
         fscanf(f, "%d %d %lg\n", &(array->column_indices[i]), &(array->row_indi
ces[i]), &(array->values[i]));

                          ^
g++ -Wno-write-strings -c mmio.c -o mmio.o
"/usr/local/cuda-6.5"/bin/nvcc -ccbin g++   -m64 -Xptxas -v     -gencode arch=co
mpute_35,code=sm_35 -o main main.o mmio.o -lcublas -lcusparse
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 1.160448 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 1.154144 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 1.487392 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 1.502176 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$
```

This one when N = 512



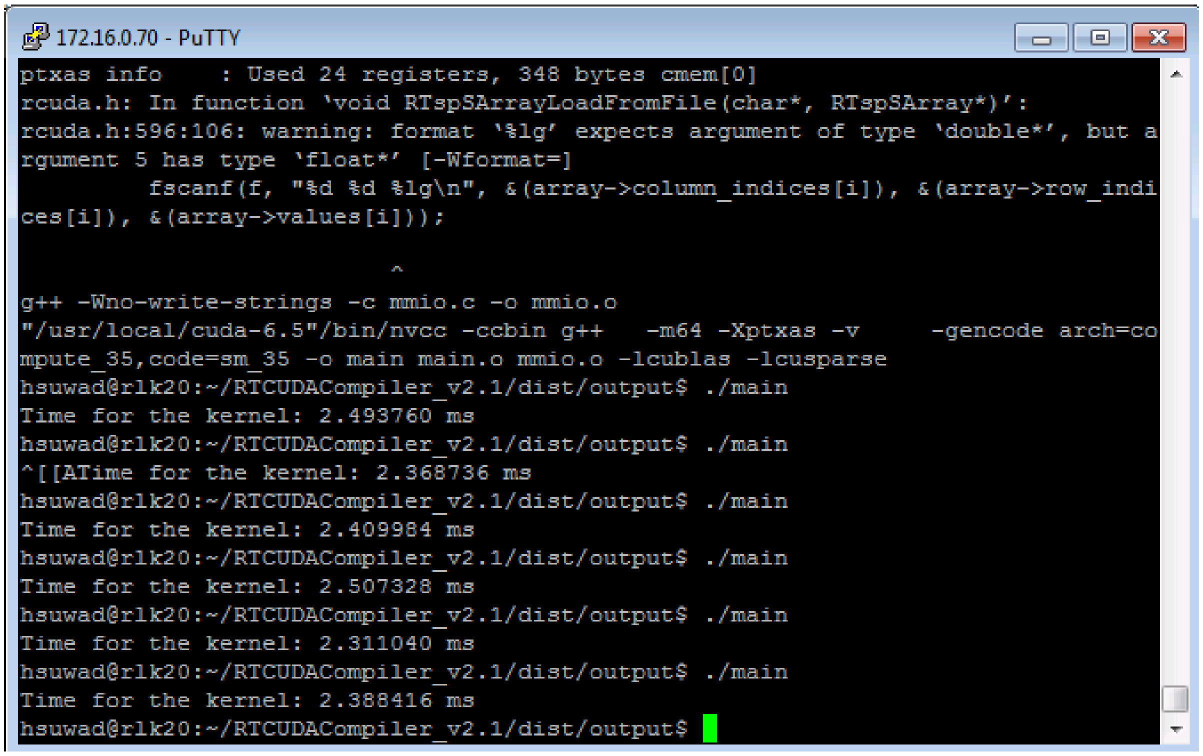
```
mpute_35,code=sm_35 -o main main.o mmio.o -lcublas -lcuspa
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.273696 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.282400 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.262656 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.267616 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.273664 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.266752 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$
```

When N = 1024



```
mpute_35,code=sm_35 -o main main.o mmio.o -lcublas -lcusparse
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.841728 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.828832 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.783296 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.838688 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.879104 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.797248 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.779744 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.796032 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.908992 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.818272 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 0.844480 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$
```

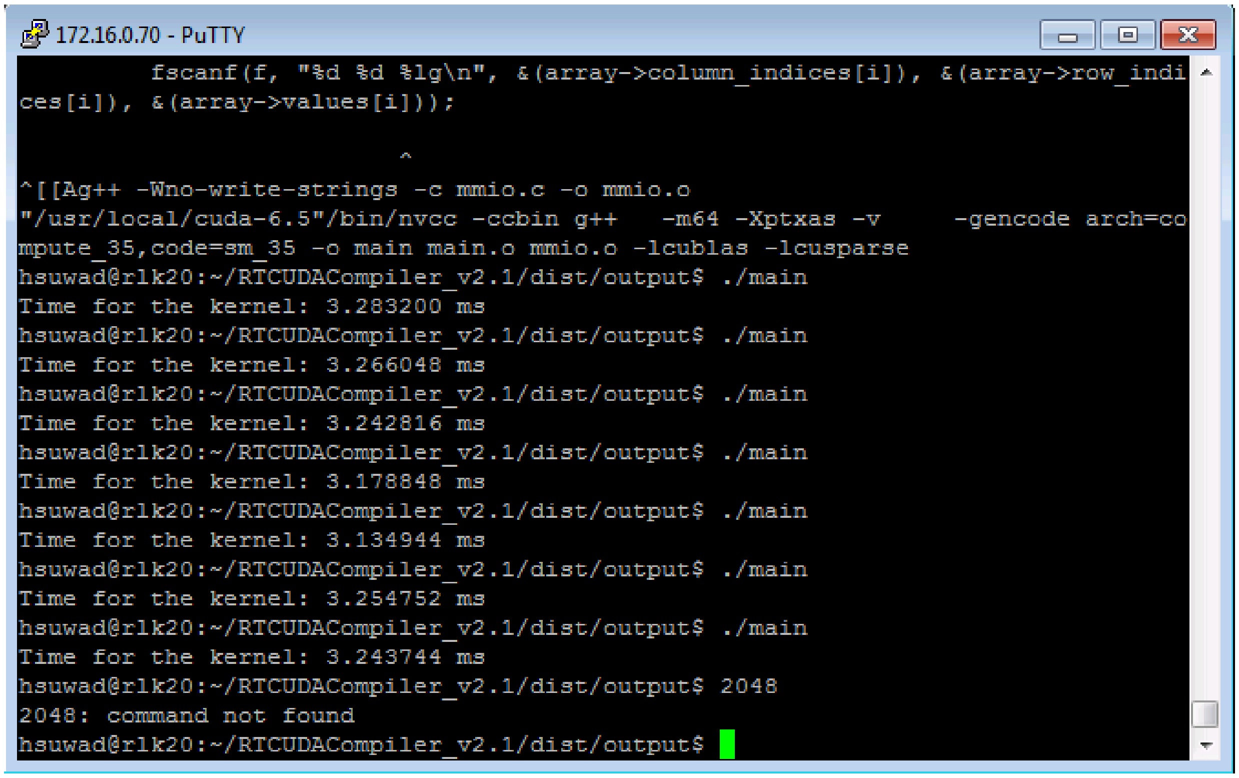
When N = 1792



172.16.0.70 - PuTTY

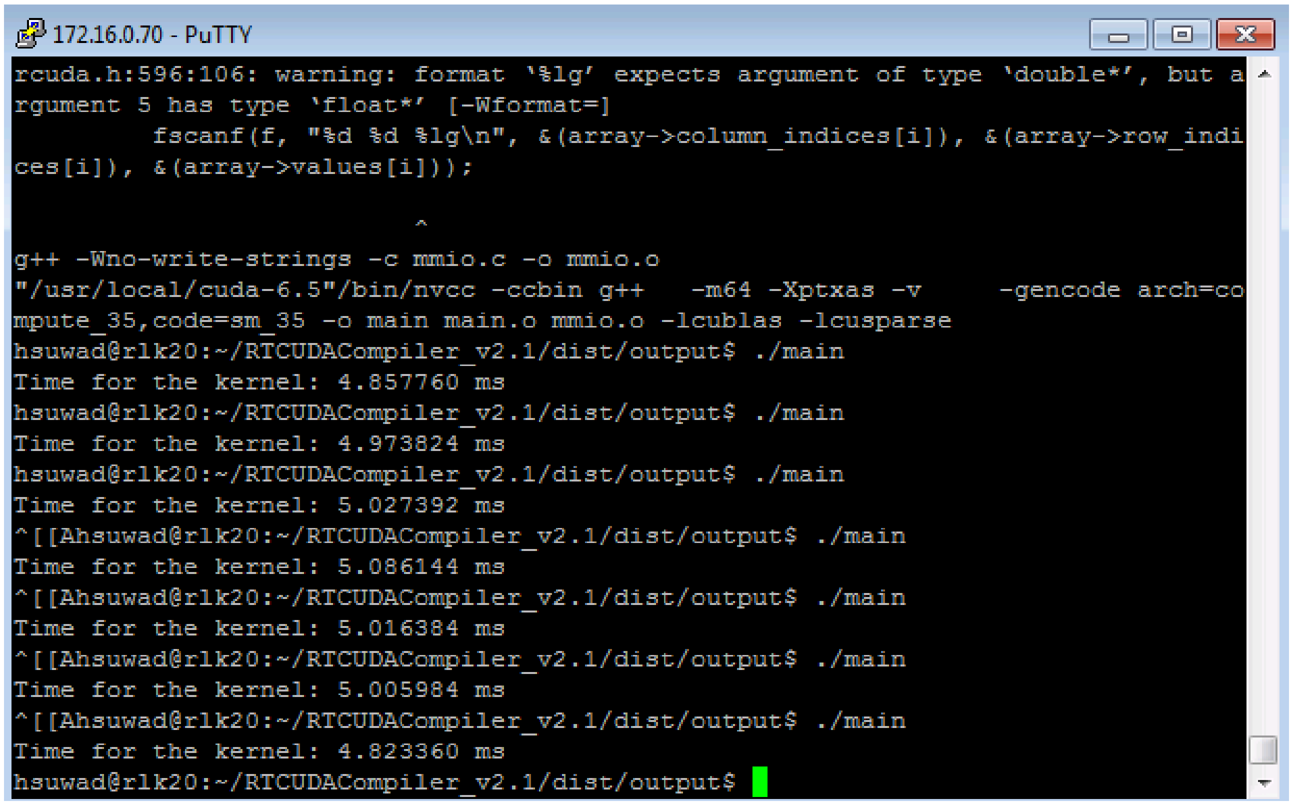
```
ptxas info    : Used 24 registers, 348 bytes cmem[0]
rcuda.h: In function 'void RTspSArrayLoadFromFile(char*, RTspSArray*)':
rcuda.h:596:106: warning: format '%lg' expects argument of type 'double*', but a
rgument 5 has type 'float*' [-Wformat=]
         fscanf(f, "%d %d %lg\n", &(array->column_indices[i]), &(array->row_indi
ces[i]), &(array->values[i]));

                                  ^
g++ -Wno-write-strings -c mmio.c -o mmio.o
"/usr/local/cuda-6.5"/bin/nvcc -ccbin g++   -m64 -Xptxas -v     -gencode arch=co
mpute_35,code=sm_35 -o main main.o mmio.o -lcublas -lcusparse
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 2.493760 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
^[[ATime for the kernel: 2.368736 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 2.409984 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 2.507328 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 2.311040 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 2.388416 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$
```

When N = 2048 which is 2N



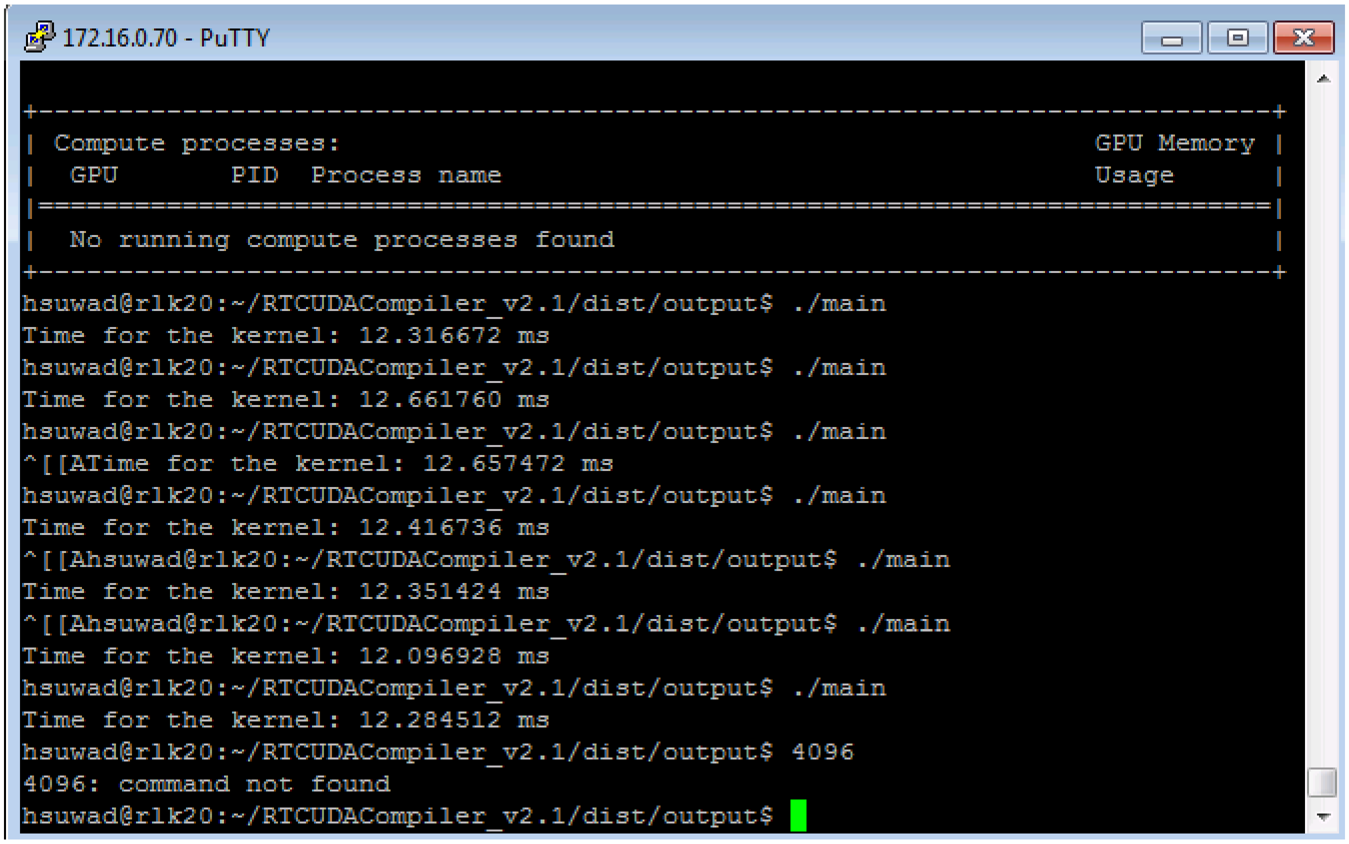
```
          fscanf(f, "%d %d %lg\n", &(array->column_indices[i]), &(array->row_indi
ces[i]), &(array->values[i]));

                             ^
^[[Ag++ -Wno-write-strings -c mmio.c -o mmio.o
"/usr/local/cuda-6.5"/bin/nvcc -ccbin g++   -m64 -Xptxas -v     -gencode arch=co
mpute_35,code=sm_35 -o main main.o mmio.o -lcublas -lcusparse
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 3.283200 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 3.266048 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 3.242816 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 3.178848 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 3.134944 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 3.254752 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 3.243744 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ 2048
2048: command not found
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$
```

Here N = 2560 as it is from original N 2.5 N



```
rcuda.h:596:106: warning: format '%lg' expects argument of type 'double*', but a
rgument 5 has type 'float*' [-Wformat=]
          fscanf(f, "%d %d %lg\n", &(array->column_indices[i]), &(array->row_indi
ces[i]), &(array->values[i]));

                             ^
g++ -Wno-write-strings -c mmio.c -o mmio.o
"/usr/local/cuda-6.5"/bin/nvcc -ccbin g++   -m64 -Xptxas -v     -gencode arch=co
mpute_35,code=sm_35 -o main main.o mmio.o -lcublas -lcusparse
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 4.857760 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 4.973824 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 5.027392 ms
^[[Ahsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 5.086144 ms
^[[Ahsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 5.016384 ms
^[[Ahsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 5.005984 ms
^[[Ahsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 4.823360 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$
```

Here N = 4096

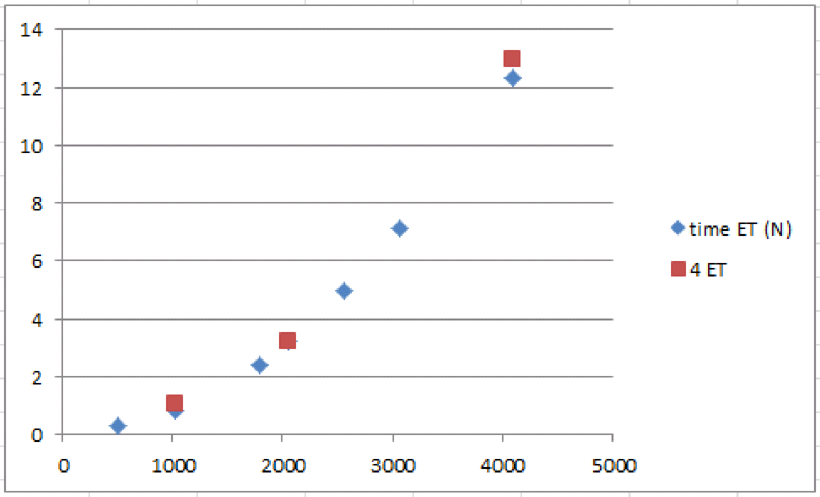


```
+------------------------------------------------------------------------------+
| Compute processes:                                               GPU Memory |
|  GPU       PID  Process name                                     Usage      |
|==============================================================================|
|  No running compute processes found                                          |
+------------------------------------------------------------------------------+
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 12.316672 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 12.661760 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
^[[ATime for the kernel: 12.657472 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 12.416736 ms
^[[Ahsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 12.351424 ms
^[[Ahsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 12.096928 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ ./main
Time for the kernel: 12.284512 ms
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ 4096
4096: command not found
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$
```

a. **Run-time Analysis of Scalability**. Analysis of problem scalability based on run time data:

a. F

N Size	Measured time	4* ET
512	0.27	
1024	0.81	1.08
1792	2.4	
2048	3.24	3.24
2560	4.97	
3072	7.13	
4096	12.35	12.96

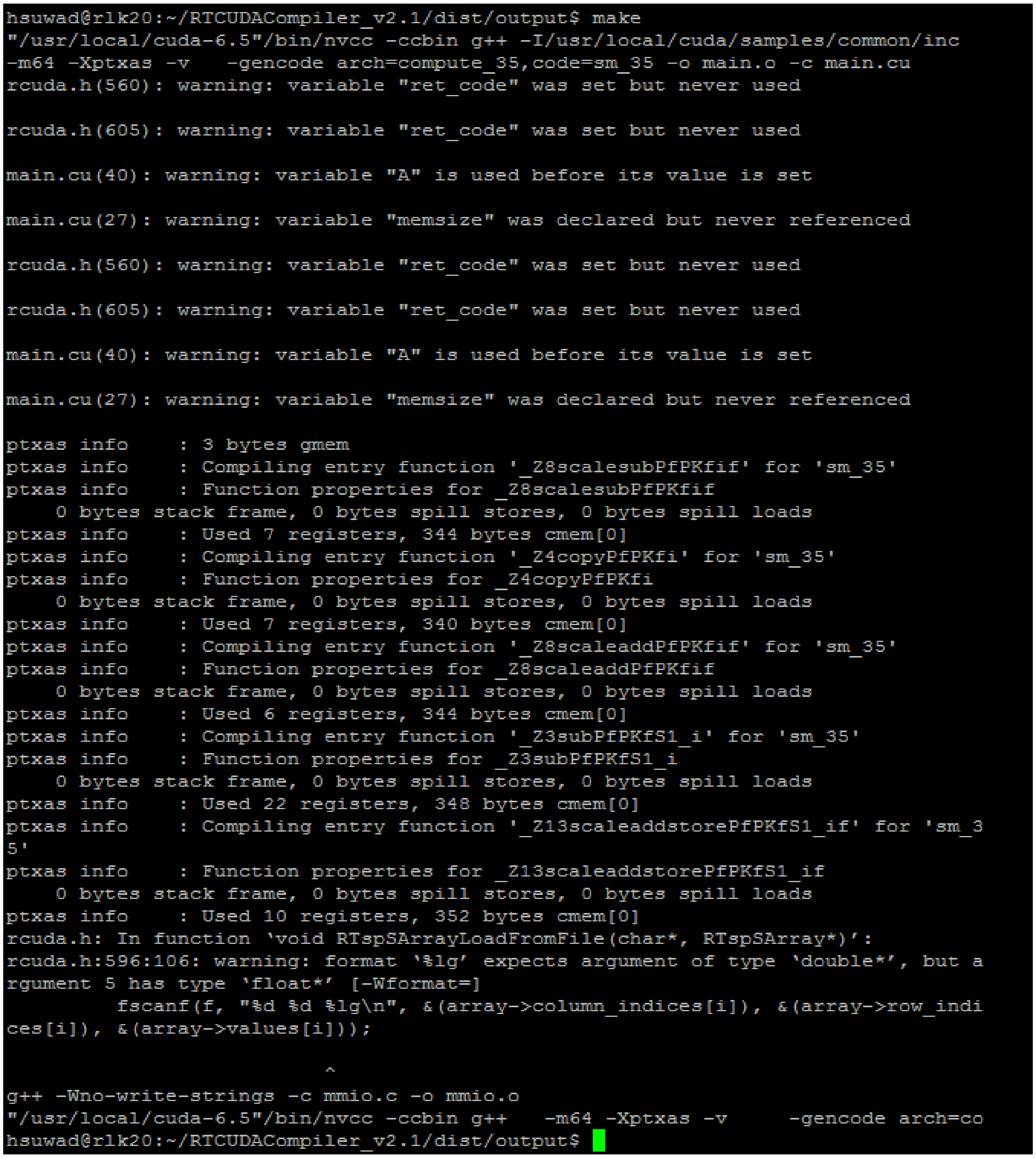


b. Here the graph shows. The effect of changing the size of N, also the approximation of the time for Later values of n depending on the value of N = 512, and 4 times it for the preceding ones when it is applicable.

Q2

Because the kernel here has 5 functions. We have to call every one alone, we get these numbers 256,1,1 64,1,1 256,1,1 32,1,1 and 64,1,1. But when calling the original file we must use one parameters of them, I tried to combine them as Ayaz told me but I still have error message even though I tried the entire possible situation.

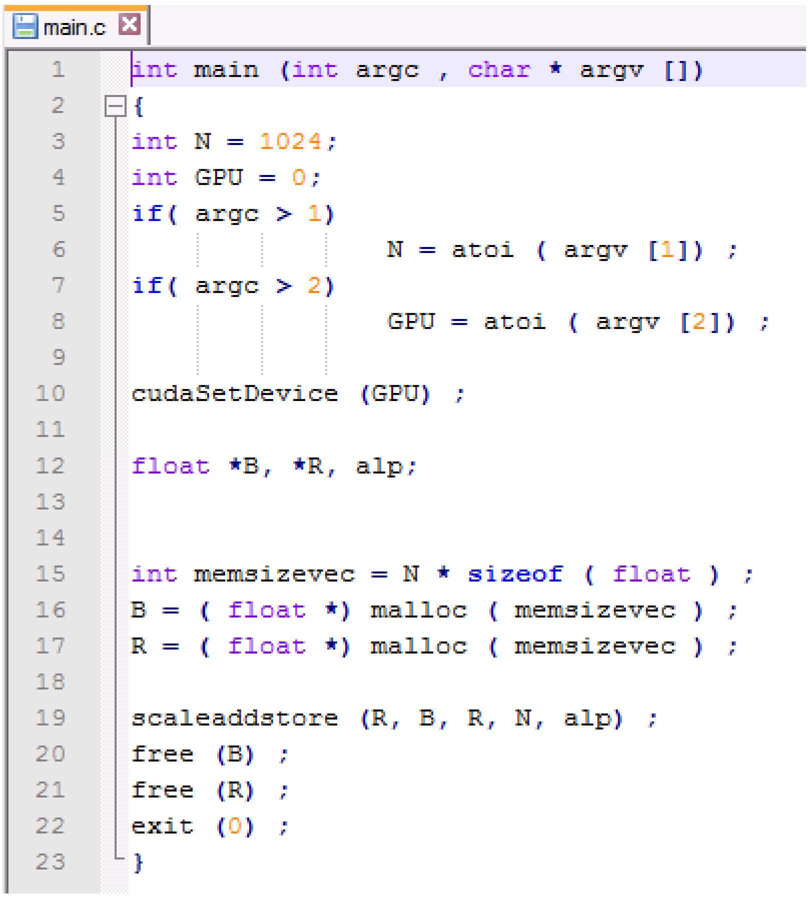
These are not the errors. These are just warning messages by the nvidia compilers which can be ignored.



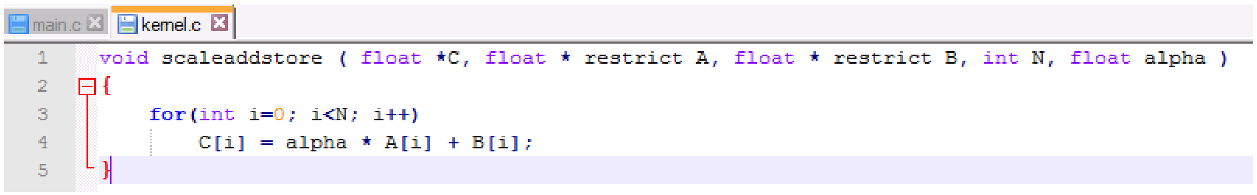
```
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ make
"/usr/local/cuda-6.5"/bin/nvcc -ccbin g++ -I/usr/local/cuda/samples/common/inc
-m64 -Xptxas -v   -gencode arch=compute_35,code=sm_35 -o main.o -c main.cu
rcuda.h(560): warning: variable "ret_code" was set but never used

rcuda.h(605): warning: variable "ret_code" was set but never used

main.cu(40): warning: variable "A" is used before its value is set

main.cu(27): warning: variable "memsize" was declared but never referenced

rcuda.h(560): warning: variable "ret_code" was set but never used

rcuda.h(605): warning: variable "ret_code" was set but never used

main.cu(40): warning: variable "A" is used before its value is set

main.cu(27): warning: variable "memsize" was declared but never referenced

ptxas info    : 3 bytes gmem
ptxas info    : Compiling entry function '_Z8scalesubPfPKfif' for 'sm_35'
ptxas info    : Function properties for _Z8scalesubPfPKfif
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 7 registers, 344 bytes cmem[0]
ptxas info    : Compiling entry function '_Z4copyPfPKfi' for 'sm_35'
ptxas info    : Function properties for _Z4copyPfPKfi
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 7 registers, 340 bytes cmem[0]
ptxas info    : Compiling entry function '_Z8scaleaddPfPKfif' for 'sm_35'
ptxas info    : Function properties for _Z8scaleaddPfPKfif
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 6 registers, 344 bytes cmem[0]
ptxas info    : Compiling entry function '_Z3subPfPKfS1_i' for 'sm_35'
ptxas info    : Function properties for _Z3subPfPKfS1_i
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 22 registers, 348 bytes cmem[0]
ptxas info    : Compiling entry function '_Z13scaleaddstorePfPKfS1_if' for 'sm_3
5'
ptxas info    : Function properties for _Z13scaleaddstorePfPKfS1_if
    0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 10 registers, 352 bytes cmem[0]
rcuda.h: In function 'void RTspSArrayLoadFromFile(char*, RTspSArray*)':
rcuda.h:596:106: warning: format '%lg' expects argument of type 'double*', but a
rgument 5 has type 'float*' [-Wformat=]
         fscanf(f, "%d %d %lg\n", &(array->column_indices[i]), &(array->row_indi
ces[i]), &(array->values[i]));

                          ^
g++ -Wno-write-strings -c mmio.c -o mmio.o
"/usr/local/cuda-6.5"/bin/nvcc -ccbin g++   -m64 -Xptxas -v     -gencode arch=co
hsuwad@rlk20:~/RTCUDACompiler_v2.1/dist/output$ 
```

I will include a process of one kernel. And the combined file for all the kernels

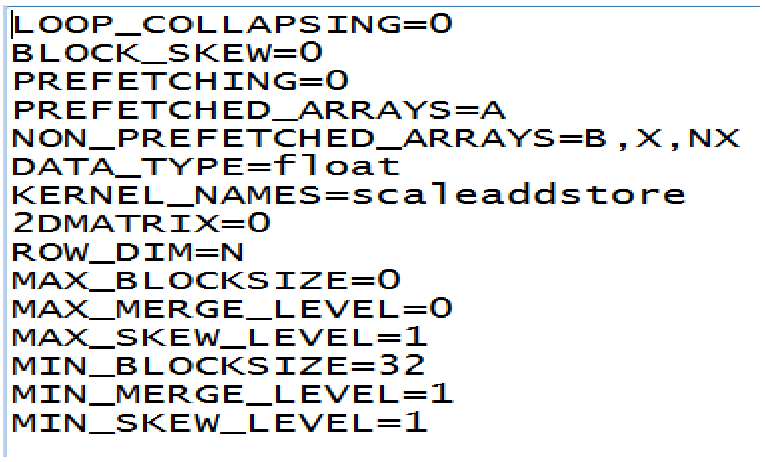
Main.c


main.c
```
int main (int argc , char * argv [])
{
int N = 1024;
int GPU = 0;
if( argc > 1)
                N = atoi ( argv [1]) ;
if( argc > 2)
                GPU = atoi ( argv [2]) ;

cudaSetDevice (GPU) ;

float *B, *R, alp;


int memsizevec = N * sizeof ( float ) ;
B = ( float *) malloc ( memsizevec ) ;
R = ( float *) malloc ( memsizevec ) ;

scaleaddstore (R, B, R, N, alp) ;
free (B) ;
free (R) ;
exit (0) ;
}
```

The kernel

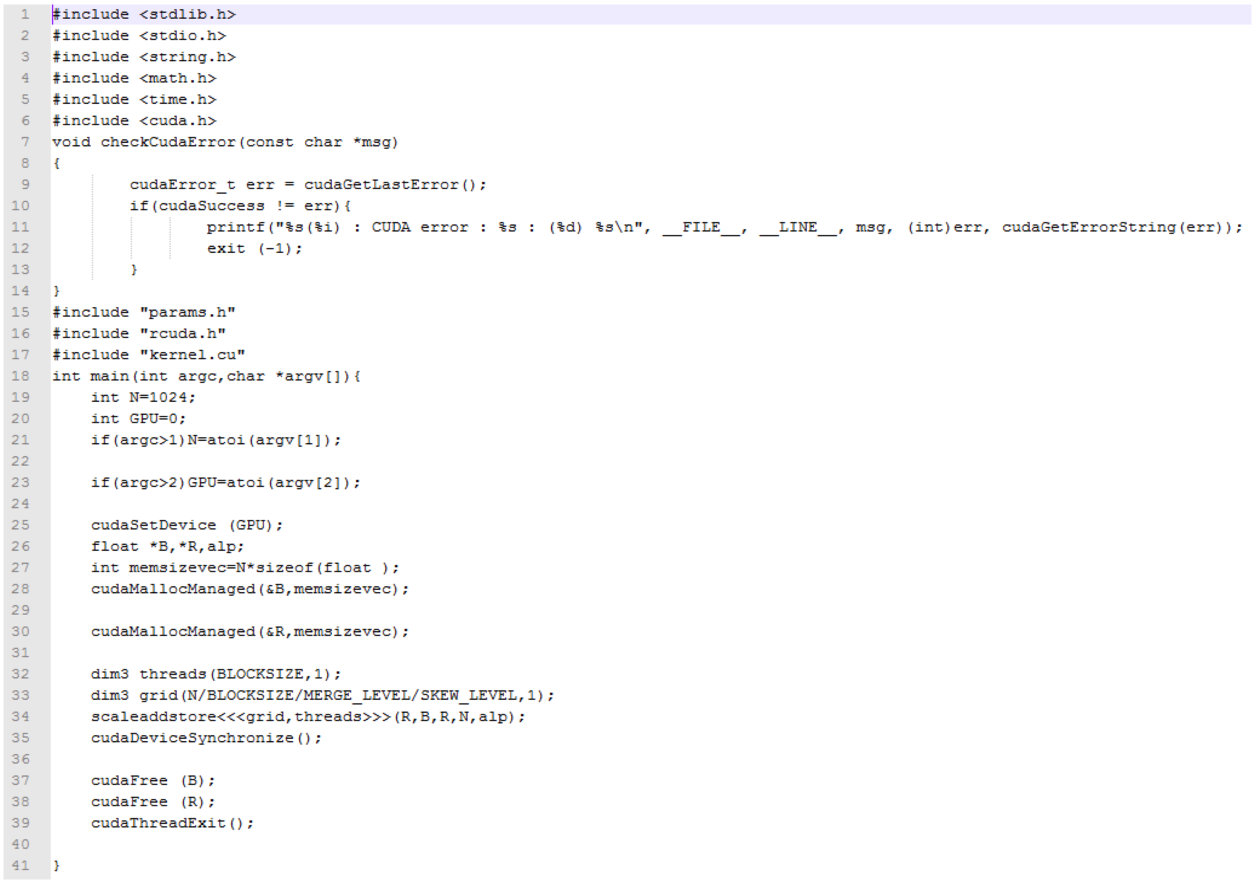

main.c kernel.c
```
void scaleaddstore ( float *C, float * restrict A, float * restrict B, int N, float alpha )
{
    for(int i=0; i<N; i++)
        C[i] = alpha * A[i] + B[i];
}
```

The config file



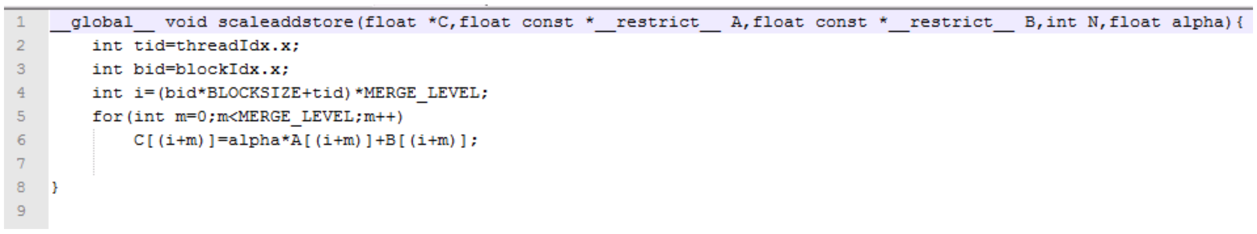
```
LOOP_COLLAPSING=0
BLOCK_SKEW=0
PREFETCHING=0
PREFETCHED_ARRAYS=A
NON_PREFETCHED_ARRAYS=B,X,NX
DATA_TYPE=float
KERNEL_NAMES=scaleaddstore
2DMATRIX=0
ROW_DIM=N
MAX_BLOCKSIZE=0
MAX_MERGE_LEVEL=0
MAX_SKEW_LEVEL=1
MIN_BLOCKSIZE=32
MIN_MERGE_LEVEL=1
MIN_SKEW_LEVEL=1
```

Here is the main . cu generated

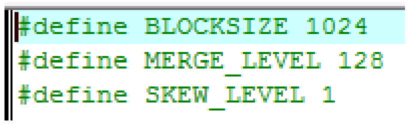


```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <cuda.h>
void checkCudaError(const char *msg)
{
        cudaError_t err = cudaGetLastError();
        if(cudaSuccess != err){
                printf("%s(%i) : CUDA error : %s : (%d) %s\n", __FILE__, __LINE__, msg, (int)err, cudaGetErrorString(err));
                exit (-1);
        }
}
#include "params.h"
#include "rcuda.h"
#include "kernel.cu"
int main(int argc,char *argv[]){
    int N=1024;
    int GPU=0;
    if(argc>1)N=atoi(argv[1]);

    if(argc>2)GPU=atoi(argv[2]);

    cudaSetDevice (GPU);
    float *B,*R,alp;
    int memsizevec=N*sizeof(float );
    cudaMallocManaged(&B,memsizevec);

    cudaMallocManaged(&R,memsizevec);

    dim3 threads(BLOCKSIZE,1);
    dim3 grid(N/BLOCKSIZE/MERGE_LEVEL/SKEW_LEVEL,1);
    scaleaddstore<<<grid,threads>>>(R,B,R,N,alp);
    cudaDeviceSynchronize();

    cudaFree (B);
    cudaFree (R);
    cudaThreadExit();

}
```

Here is the kernel.cu and then param.h



```
__global__ void scaleaddstore(float *C,float const *__restrict__ A,float const *__restrict__ B,int N,float alpha){
    int tid=threadIdx.x;
    int bid=blockIdx.x;
    int i=(bid*BLOCKSIZE+tid)*MERGE_LEVEL;
    for(int m=0;m<MERGE_LEVEL;m++)
        C[(i+m)]=alpha*A[(i+m)]+B[(i+m)];

}

```



```
#define BLOCKSIZE 1024
#define MERGE_LEVEL 128
#define SKEW_LEVEL 1
```

As final result multiple kernels has a lot of difficulty and manual work to do.

This is correct that converting programs containing multiple functions may have some difficulties for the user to merge. This functionality will be added in future release of RT-CUDA.

REFERENCES

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *J. Parallel Distrib. Comput.*, vol. 68, no. 10, pp. 1370–1380, Oct. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2008.05.014

[2] R. G. Belleman, J. Bdorf, and S. F. P. Zwart, "High performance direct gravitational n-body simulations on graphics processing units ii: An implementation in cuda," *New Astronomy*, vol. 13, no. 2, pp. 103 – 112, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1384107607000760

[3] B. Dumitrescu, J.-L. Roch, and D. Trystram, "Fast matrix multiplication algorithms on mimd architectures," *Parallel Algorithms Appl.*, vol. 4, no. 1-2, pp. 53–70, 1994.

[4] V. Strassen, "Gaussian elimination is not optimal." *Numerische Mathematik*, vol. 14, no. 3, pp. 354–356, 1969.

[5] S. Winograd, *Some Remarks on FAST Multiplication of Polynomials*, ser. Research reports // IBM, 1973. [Online]. Available: http://books.google.com.sa/books?id=T87etgAACAAJ

[6] D. H. Bailey, "Extra high speed matrix multiplication on the cray-2," *SIAM J. Sci. Stat. Comput.*, vol. 9, no. 3, pp. 603–607, May 1988. [Online]. Available: http://dx.doi.org/10.1137/0909040

[7] J. Li, S. Ranka, and S. Sahni, "Strassen's matrix multiplication on gpus," in *Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems*, ser. ICPADS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 157–164. [Online]. Available: http://dx.doi.org/10.1109/ICPADS.2011.130

[8] A. Buluç and J. R. Gilbert, "Highly parallel sparse matrix-matrix multiplication," *CoRR*, vol. abs/1006.2183, 2010.

[9] W. Cao, L. Yao, Z. Li, Y. Wang, and Z. Wang, "Implementing sparse matrix-vector multiplication using cuda based on a hybrid sparse matrix format," in *Computer Application and System Modeling (ICCASM), 2010 International Conference on*, vol. 11. IEEE, 2010, p. V11161.

[10] A. Monakov and A. Avetisyan, "Implementing blocked sparse matrix-vector multiplication on nvidia gpus," in *Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, ser. SAMOS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp.

289–297. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03138-0_32

[11] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu, "Languages and compilers for parallel computing," J. N. Amaral, Ed. Berlin, Heidelberg: Springer-Verlag, 2008, ch. CUDA-Lite: Reducing GPU Programming Complexity, pp. 1–15. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89740-8_1

[12] S. Lee, S.-J. Min, and R. Eigenmann, "Openmp to gpgpu: a compiler framework for automatic translation and optimization," *SIGPLAN Not.*, vol. 44, no. 4, pp. 101–110, Feb. 2009. [Online]. Available: http://doi.acm.org/10.1145/1594835.1504194

[13] T. D. Han and T. S. Abdelrahman, "hicuda: High-level gpgpu programming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 78–90, 2011.

[14] G. Rudy, *CUDA-CHiLL: A Programming Language Interface for GPGPU Optimizations and Code Generation.* School of Computing, University of Utah, 2010. [Online]. Available: http://books.google.com.sa/books?id=vg66ZwEACAAJ

[15] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan, "A framework for dynamically instrumenting gpu compute applications within gpu ocelot," in *Proceedings of the Fourth Workshop on General*

Purpose Processing on Graphics Processing Units, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 9:1–9:9. [Online]. Available: http://doi.acm.org/10.1145/1964179.1964192

[16] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A gpgpu compiler for memory optimization and parallelism management," *SIGPLAN Not.*, vol. 45, no. 6, pp. 86–97, Jun. 2010. [Online]. Available: http://doi.acm.org/10.1145/1809028.1806606

[17] Y. Yan, M. Grossman, and V. Sarkar, "Jcuda: A programmer-friendly interface for accelerating java programs with cuda," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 887–899. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03869-3_82

[18] L. Chen, *Exploring Novel Many-Core Architectures for Scientific Computing*. BiblioBazaar, 2012. [Online]. Available: http://books.google.com.sa/books?id=RklfLwEACAAJ

[19] V. Volkov and J. W. Demmel, "Benchmarking gpus to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 31:1–31:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413402

[20] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with gpu accelerators," in *IPDPS Workshops*. IEEE, 2010, pp.

1–8.

[21] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov, “Qr factorization on a multicore node enhanced with multiple gpu accelerators,” in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 932–943. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2011.90

[22] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach (Applications of GPU Computing Series)*, 1st ed. Morgan Kaufmann, Feb. 2010. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0123814723

[23] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, “Program optimization space pruning for a multithreaded gpu,” in *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO ’08. New York, NY, USA: ACM, 2008, pp. 195–204. [Online]. Available: http://doi.acm.org/10.1145/1356058.1356084

[24] R. Nath, S. Tomov, and J. Dongarra, “An improved magma gemm for fermi graphics processing units,” *Int. J. High Perform. Comput.*

Appl., vol. 24, no. 4, pp. 511–515, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1177/1094342010385729

[25] R. Nath, S. Tomov, T. T. Dong, and J. Dongarra, "Optimizing symmetric dense matrix-vector multiplication on gpus," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. New York, NY, USA: ACM, 2011, pp. 6:1–6:10. [Online]. Available: http://doi.acm.org/10.1145/2063384.2063392

[26] A. H. El Zein and A. P. Rendell, "From sparse matrix to optimal gpu cuda sparse matrix vector product implementation," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, ser. CCGRID '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 808–813. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2010.81

[27] Z. Wang, X. Xu, W. Zhao, Y. Zhang, and S. He, "Optimizing sparse matrix-vector multiplication on CUDA," in *International Conference on Education Technology and Computer*, 2010.

[28] S. Xu, H. X. Lin, and W. Xue, "Sparse matrix-vector multiplication optimizations based on matrix bandwidth reduction using nvidia cuda," in *Proceedings of the 2010 Ninth International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, ser.

DCABES '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 609–614. [Online]. Available: http://dx.doi.org/10.1109/DCABES.2010.162

[29] M. Yang, C. Sun, Z. Li, and D. Cao, "An Improved Sparse Matrix -Vector Multiplication Kernel for Solving Modified Equation in Large Scale Power Flow Calculation on CUDA," in *IEEE 7th International Power Electronics and Motion Control Conference - ECCE Asia*, 2012.

[30] D. Yan, H. Cao, X. Dong, B. Zhang, and X. Zhang, "Optimizing algorithm of sparse linear systems on gpu," in *Proceedings of the 2011 Sixth Annual ChinaGrid Conference*, ser. CHINAGRID '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 174–179. [Online]. Available: http://dx.doi.org/10.1109/ChinaGrid.2011.45

[31] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 152–163, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1555815.1555775

[32] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for gpu architectures," *SIGPLAN Not.*, vol. 45, no. 5, pp. 105–114, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837853.1693470

[33] P. Guo and L. Wang, "Accurate cuda performance modeling for sparse matrix-vector multiplication," 2012.

[34] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley, and K. Yelick, "Self-adapting linear algebra algorithms and software," *Proc. IEEE*, vol. 93, no. 2, pp. 293–312, February 2005.

[35] X. Cui, Y. Chen, C. Zhang, and H. Mei, "Auto-tuning dense matrix multiplication for gpgpu with cache," in *Proceedings of the 2010 IEEE 16th International Conference on Parallel and Distributed Systems*, ser. ICPADS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 237–242. [Online]. Available: http://dx.doi.org/10.1109/ICPADS.2010.64

[36] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *In International Parallel & Distributed Processing Symposium (IPDPS*, 2010.

[37] Y. Li, J. Dongarra, and S. Tomov, "A note on auto-tuning gemm for gpus," in *Proceedings of the 9th International Conference on Computational Science: Part I*, ser. ICCS '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 884–892. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01970-8_89

[38] J. Kurzak, S. Tomov, and J. Dongarra, "Autotuning gemms for fermi." LAPACK Working Note, Tech. Rep. 245, Apr. 2011. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn245.pdf

[39] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *IS-PASS*. IEEE Computer Society, 2010, pp. 235–246.

[40] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Gu 2007. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_

[41] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Eurographics 2005, State of the Art Reports*, pp. 21–51, August 2005. [Online]. Available: http://graphics.idav.ucdavis.edu/publications/print_pub?pub_id=844

[42] K. Mueller, F. Xu, and N. Neophytou, "Why do commodity graphics hardware boards (gpus) work so well for acceleration of computed tomography?" pp. 64 980N–64 980N–12, 2007. [Online]. Available: + http://dx.doi.org/10.1117/12.716797

[43] Nvidia, *Fermi Compute Architecture Whitepaper*, 2009.

[44] ——, *Kepler GK110 Compute Architecture Whitepaper*, 2012.

[45] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the gpu using cuda," in *Proceedings of the 14th International Conference on High Performance Computing*, ser. HiPC'07. Springer-Verlag, 2007, pp. 197–208. [Online]. Available: http://dl.acm.org/citation.cfm?id=1782174.1782200

[46] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the*

13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345220

[47] G. M. Striemer and A. Akoglu, "Sequence alignment with gpu: Performance and design challenges," in *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy*, 2009, pp. 1–10.

[48] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete fourier transforms on graphics processors," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 2:1–2:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413373

[49] M. Hussein, A. Varshney, and L. Davis, "On implementing graph cuts on cuda," in *First Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, October 2007.

[50] S. Xiao and W. chun Feng, "Inter-block gpu communication via fast barrier synchronization," in *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010*, 2010, pp. 1–12.

[51] M. Harris, "Optimizing Parallel Reduction in CUDA," http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf, NVIDIA, Tech. Rep., 2008.

[52] NVIDIA, *NVIDIA CUDA Programming Guide 2.0*, 2008.

[53] D. Rivera-Polanco, "Collective communication and barrier synchronization on nvidia cuda gpu," MS Thesis, University of Kentucky, 2009.

[54] NVIDIA, "Allinea ddt — nvidia developer zone," https://developer.nvidia.com/allinea-ddt, accessed: 30 December 2013.

[55] *CUDA C Best Practices Guide*, 4th ed., NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara 95050, USA, may 2011.

[56] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, June 2011.

[57] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Addison-Wesley, 2013.

[58] P. H. Ha, P. Tsigas, and O. J. Anshus, "The synchronization power of coalesced memory accesses." in *DISC*, ser. Lecture Notes in Computer Science, G. Taubenfeld, Ed., vol. 5218. Springer, 2008, pp. 320–334.

[59] NVIDIA, "Nvidias next generation cuda computer architecture kepler gk110," NVIDIA Corporation, Whitepaper, 2013.

[60] A. Jackson and O. Agathokleous, "Dynamic loop parallelisation," *CoRR*, vol. abs/1205.2367, 2012.

[61] Y. Yang and H. Zhou, "The implementation of a high performance gpgpu compiler," *International Journal of Parallel Programming*, vol. 41, no. 6, pp. 768–781, 2013.

[62] G. van den Braak, B. Mesman, and H. Corporaal, “Compile-time gpu memory access optimizations,” in *Embedded Computer Systems (SAMOS), 2010 International Conference on*, July 2010, pp. 200–207.

[63] G. Wang, “Coordinate strip-mining and kernel fusion to lower power consumption on gpu,” in *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011, pp. 1–4.

[64] A. Wakatani, “Effectiveness of a strip-mining approach for vq image coding using gpgpu implementation,” in *Image and Vision Computing New Zealand, 2009. IVCNZ ’09. 24th International Conference*, Nov 2009, pp. 35–38.

[65] J. Hennessy, D. Patterson, and K. Asanović, *Computer Architecture: A Quantitative Approach*, ser. Computer Architecture: A Quantitative Approach. Morgan Kaufmann/Elsevier, 2012.

[66] R. Farivar and R. Campbell, “Plasma: Shared memory dynamic allocation and bank-conflict-free access in gpus,” in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, Sept 2012, pp. 612–613.

[67] “Tuning cuda applications for kepler,” http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html, accessed: 10-06-2013.

[68] A. P. Ershov, “On programming of arithmetic operations,” *Commun. ACM*, vol. 1, no. 8, pp. 3–6, Aug. 1958.

[69] D. Ye, A. Titov, V. Kindratenko, I. Ufimtsev, and T. Martinez, “Porting optimized gpu kernels to a multi-core cpu: Computational quantum chem-

istry application example," in *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, July 2011, pp. 72–75.

[70] T. Ikeda, F. Ino, and K. Hagihara, "A code motion technique for accelerating generalpurpose computation on the gpu," in *In Proceedings of the International Parallel and Distributed Processing Symposium*, 2006, pp. 1–10.

[71] G. Murthy, M. Ravishankar, M. Baskaran, and P. Sadayappan, "Optimal loop unrolling for gpgpu programs," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–11.

[72] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, Aug. 2004. [Online]. Available: http://doi.acm.org/10.1145/1015706.1015800

[73] S.-W. Liao, Z. Du, G. Wu, and G.-Y. Lueh, "Data and computation transformations for brook streaming applications on multiprocessors," in *CGO*. IEEE Computer Society, 2006, pp. 196–207.

[74] M. Peercy, M. Segal, and D. Gerstmann, "A performance-oriented data parallel virtual machine for GPUs," in *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*. New York, NY, USA: ACM, 2006, p. 184.

[75] OpenMP, "The openmp api specification for parallel programming," http://openmp.org/wp, accessed: 18 March 2013.

[76] S. Lee and R. Eigenmann, "Openmpc: Extended openmp programming and tuning for gpus," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.36

[77] P. G. Inc., "Portlan group," http://www.pgroup.com/resources/accel.htm, accessed: 18 March 2013.

[78] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU '10. New York, NY, USA: ACM, 2010, pp. 51–61. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735698

[79] J. C. Beyer, E. J. Stotzer, A. Hart, and B. R. de Supinski, "Openmp for accelerators." in *IWOMP*, ser. Lecture Notes in Computer Science, B. M. Chapman, W. D. Gropp, K. Kumaran, and M. S. Mller, Eds., vol. 6665. Springer, 2011, pp. 108–121. [Online]. Available: http://dblp.uni-trier.de/db/conf/iwomp/iwomp2011.html#BeyerSHS11

[80] S. Lee and J. S. Vetter, "Early evaluation of directive-based gpu programming models for productive exascale computing," in *Proceedings of the International Conference on High Performance Computing, Networking,*

Storage and Analysis, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 23:1–23:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389028

[81] F. Bodin and S. Bihan, "Heterogeneous multicore parallel programming for graphics processing units," *Sci. Program.*, vol. 17, no. 4, pp. 325–336, Dec. 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1662626.1662632

[82] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," *SIGPLAN Not.*, vol. 43, no. 6, pp. 101–113, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1379022.1375595

[83] C. Chen, J. Chame, and M. Hall, "Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy," *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, vol. 0, pp. 111–122, 2005.

[84] R. Reyes, "Yacf: Yet another compiler framework," Universidad de La Laguna, Report, February 2013.

[85] T. Parr, *The Definitive ANTLR 4 Reference*, ser. Oreilly and Associate Series. Pragmatic Bookshelf, 2013.

[86] N. I. of Standards and Technology, "Text file formats," http://math.nist.gov/MatrixMarket/formats.html, accessed: 19 November 2014.

[87] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

Vitae

- Name: Ayaz ul Hassan Khan
- Nationality: Pakistan
- Date of Birth: 30th December, 1979
- Email: *ayazhk@gmail.com*
- Permenant Address: A-314 Block 2 Gulistan-e-Jauhar Karachi, Pakistan.

Education

- PhD Computer Science, King Fahd University of Petroleum and Minerals, Dhahran - Saudi Arabia
- MS Computer Science, Lahore University of Management Sciences, Lahore - Pakistan
- Bachelor of Computer Science and IT, NED University of Engineering and Technology, Karachi - Pakistan

Additional Qualifications/Short Courses

- Oracle Certified Professional (OCP) DBA Track

- Microsoft Certified Technology Specialist
- SAP: Introduction to ABAP Programming Course
- SAP Basics Course

Work Experience

- Assistant Professor, National University of Computer and Emerging Sciences, Aug 2009 - Jul 2010
- Software Engineer Database Architecture/Application Development, TRG (The Resource Group), Jul 2007 - Jul 2009
- Assistant Professor, Mohammad Ali Jinnah University, Aug 2006 - Jun 2007
- Software Engineer Application Development, Wavetec Private Limited, Jan 2005 - Jul 2006

Publications

- Ayaz ul Hassan Khan, Mayez Al-Mouhamed, Allam Fatayer, Anas Almousa, Abdulrahman Baqais, and Mohammad Assayony, "Padding Free Bank Conict Resolution for CUDA-Based Matrix Transpose Algorithm", Accepted in International Journal of Networked and Distributed Computing (IJNDC), Vol. 2, No. 3, 2014.

- Mayez Al-Mouhamed, and Ayaz ul Hassan Khan, “Exploration of Automatic Optimisation for CUDA Programming”, International Journal of Parallel, Emergent, and Distributed Systems (IJPEDS), 2014. DOI: 10.1080/17445760.2014.953158

- Basem Al-Madani, Ayaz ul Hassan Khan, and Zubair A. Baig, “A Novel Mobility-Aware Data Transfer Service (MADTS) Based on DDS Standards”, The Arabian Journal of Science and Engineering, 2014. DOI: 10.1007/s13369-014-0944-7

- A. H. Khan, M. A. Al-Mouhamed, A. Almousa, A. Fatayar, A. R. Ibrahim, and A. J. Siddiqui, “AES-128 ECB Encryption on GPUs and Effects of Input Plaintext Patterns on Performance”, 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2014), Las Vegas, USA.

- A. H. Khan, M. A. Al-Mouhamed, A. Almousa, A. Fatayar, A. Baqais, and M. Assayony, “Padding Free Bank Conict Resolution for CUDA-Based Matrix Transpose Algorithm”, 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2014), Las Vegas, USA.

- Mayez Al-Mouhamed and Ayaz ul Hassan Khan, “Exploration of Automatic Optimization for CUDA Programming”, 2nd IEEE International Conference on Parallel, Distributed and Grid Computing, India, PDGC2012, 2012.

- Ayaz ul Hassan Khan and Muhammad Shoieb Arshad, "Performance Improvement of Overlay Networks for P2P Systems", 3rd International Conference on Intelligent Systems, Modelling and Simulation in Malaysia, ISMS2012, 2012.

- Shafiqur Rehman, Aftab Ahmad, Huseyin Saricimen, Luai M. Al-Hadhrami, Shamsuddin Khan, and Ayaz ul Hassan Khan, "Development of a Web-Based Corrosion Cost Inventory System for Saudi Arabia", NACE International Corrosion 2012 Conference and Expo.

- Muhammad Yousuf Bawany, Dr. Asim Karim, Ayaz-ul-Hassan Khan, Muhammad Sibghatullah Siddiqui, "Outlier Detection in Evolving Data Streams: An Evaluation of the LOADED Algorithm", 3rd ACM International Conference on Intelligent Computing and Information Systems - ICICIS 2007, Egypt, March 15-18, 2007

- Ayaz ul Hassan Khan, "Investigate and Parallel Processing using E1350 IBM eServer Cluster", LAMBERT Academic Publishing, 2012.

Professional Skills

- Research and Development

- Project Management

- System Support

- Time Management
- Team Management
- People Coordination
- Leadership
- Flexibility

Achievements and Awards

- Awarded Travel Grant for Research by Higher Education Commission (HEC)
- Gold Medalist for getting 100% marks in SQL-PL/SQL paper of OCP.
- Academic scholarships from NED University in each year to remain within top 5 positions