Available online at www.sciencedirect.com
SciVerse ScienceDirect
Procedia Computer Science 9 (2012) 1910 – 1919
Procedia Computer Science

International Conference on Computational Science, ICCS 2012

CUDA: Compiling and optimizing for a GPU platform

Gautam Chakrabarti[1], Vinod Grover, Bastiaan Aarts, Xiangyun Kong, Manjunath Kudlur, Yuan Lin, Jaydeep Marathe, Mike Murphy, Jian-Zhong Wang

NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, USA

Abstract

Graphics processor units (GPUs) have evolved to handle throughput oriented workloads where a large number of parallel threads must make progress. Such threads are organized around shared memory making it possible to synchronize and cooperate on shared data. Current GPUs can run tens of thousands of hardware threads and have been optimized for graphics workloads. Several high level languages have been developed to easily program the GPUs for general purpose computing problems. The use of high-level languages introduces the need for highly optimizing compilers that target the parallel GPU device.

In this paper, we present our experiences in developing compilation techniques for a high level language called CUDA C. We explain the CUDA architecture and programming model and provide insights into why certain optimizations are important for achieving high performance on a GPU. In addition to classical optimizations, we present optimizations developed specifically for the CUDA architecture. We evaluate these techniques, and present performance results that show significant improvements on hundreds of kernels as well as applications.

Keywords: CUDA, GPGPU, compiler optimizations

1. Introduction

For the last decade, graphics processors (GPUs) have been evolving rapidly in several dimensions. First, the densities observed in GPUs are outpacing those in commodity CPUs [1]. The latest GPU from NVIDIA, called Fermi, has roughly 3 billion transistors. Secondly these GPUs have evolved to provide high throughputs where every pixel needs to be computed and painted in a fixed frame time. The GPU can create, run and retire a large number of threads very rapidly. The GPU uses multithreading to hide latency – when a thread stalls it is beneficial to have several threads that are ready to run. The register state of the threads is replicated and that makes it very cheap to switch to a waiting thread.

The aggregate compute power of such devices is tremendous and many have tried to use GPUs for general purpose scientific computations (GPGPUs) [2][3]. Early attempts tried to use existing graphics languages and APIs [4] for numeric and scientific computations. General experience from such attempts was that

[1]Corresponding author
Email address: gchakrabarti@nvidia.com (Gautam Chakrabarti)

doi:10.1016/j.procs.2012.04.209

it required a high programming and learning effort to repurpose software APIs and languages meant for graphics applications. A programmer, in order to leverage existing software, had to understand graphics hardware and abstractions for programming such machines.

Later attempts tried to move away from graphics abstractions by designing and implementing stream oriented languages designed to exploit the parallelism inherent in GPUs. Brook [5] was one such language based on C that added notions of streams and kernels. This was a significant improvement over the previous approaches but a new language requires a significant learning effort.

The biggest challenge is how to design software for GPUs that is easy to write and yet can help achieve high performance at a relatively modest development cost.

In this paper we describe a system called CUDA C that makes significant progress towards addressing this challenge. CUDA C is a heterogeneous programming environment, with minimal extensions to C/C++, to help programmers write applications for GPUs. CUDA C has been successfully learnt easily and used by hundreds of thousands of C/C++ programmers. This early success of CUDA C inspired similar technologies such as DirectCompute [6] and OpenCL [7].

This paper makes the following contributions:

- We present a study of the CUDA architecture and programming model, and some high-level optimizations that a compiler should have to achieve high performance in CUDA kernels.
- We provide insights into why these optimizations are important.
- We give a detailed description of a production quality CUDA compiler and its implementation.
- We provide a detailed study of performance of 521 kernels and how compiler optimizations affect their performance. We also present performance results from an application and a benchmark suite. Our results show performance improvements up to 32X the baseline performance.

The rest of the paper is organized as follows. In section 2 we give an overview of the CUDA architecture and the CUDA compiler. In section 3 we give a detailed description of existing and new optimizations that are part of the compiler. In section 4 we evaluate the effectiveness of these optimizations on 521 CUDA kernels from benchmarks and applications, as well as on the runtime performance of other applications. In section 5 we describe related work. Finally in section 6 we conclude with some directions for future work.

2. CUDA architecture and compiler

In this section, we present some characteristics of the CUDA architecture, and a brief overview of the CUDA compiler.

2.1. CUDA Architecture

The CUDA [8] architecture is built around an array of multithreaded streaming multiprocessors (SMs) in an NVIDIA GPU. The data-parallel compute kernels in an application are off-loaded for concurrent execution on the GPU *device*, while the remainder of the application is executed on the CPU *host*. A CUDA SM has a Single Instruction Multiple Thread (SIMT) architecture. A thread executing a kernel is part of a *cooperative thread array* (CTA). Threads in a CTA are scheduled in groups of parallel threads called *warps*. The register set in the SM is partitioned among the warps running on the SM. As a result, the number of threads that can simultaneously be scheduled on an SM is dependent on the number of registers used by the kernel and the number of registers available on the SM. We define *occupancy* to be the ratio of the number of resident warps to the maximum number of resident warps on an SM [9]. It is important for a CUDA C compiler to reduce register usage to improve occupancy, without sacrificing code quality of the kernel.

All threads in a warp execute the same instruction. If threads in a warp take different paths in a conditional branch, then the conditional is said to be *thread-variant*, and the threads are said to *diverge* at the branch. In such a scenario, the taken branches are executed serially. It is best for performance if no branch divergence occurs.

```
{ // D, S: array of int
  int *p = D;
  if (m)
    p = S;
  else if (n)
    foo(&p);
  // use (*p)
}
```

```
{ // X, W: array of int
  q = X;
  for (int i = 0; i < n; ++i) {
    /* omitted */ = q[i];
    q = &W[1024-i];
  }
}
```

```
__device__ int *G1;
__shared__ int G2;

__device__ void f(void)
{ G1 = &G2; }

__device__ void g(void)
{ /* use (*G1) */ }
```

```
__device__ void f(int *a, int *b)
{ /* use (*a), (*b) */ }

__device__ int D;
__shared__ int S;

__device__ void g(void)
{ f(&D, &S); }
```

Figure 1: Memory space analysis motivating examples: (a) Conditional statements (b) Loop statement (c) Escaped address (d) Memory space across calls

2.2. Compiler Overview

The compiler, called nvcc, is part of NVIDIA's production CUDA toolchain. The heterogeneous CUDA program containing host and device code is input to a CUDA C language front end (CUDAFE). The front end partitions the program into a host part and a device part. The host part is compiled as C++ code by the host compiler and the device part is fed to a high-level backend based on Open64 [10] and targets the PTX instruction set. The PTX code is compiled by a device specific optimizing code generator called PTXAS. The compiled host code is combined with the device code to create an executable application. The work presented in this paper is based on the Open64 framework and covers the high level and the low level optimizations in Section 3.

2.2.1. Open64 Background

Open64 is an open-source production quality compiler infrastructure based on the SGI Pro64 compiler. It translates the input code to an IR called WHIRL. Open64's high level optimization passes, PreOptimizer (PreOpt) and WHIRL Optimizer (WOPT), are invoked on this representation. Finally code generation (CG) phase translates the optimized WHIRL to its internal IR. It performs low level optimizations, and emits PTX output.

2.2.2. PTX Overview

The output language of Open64, PTX [11], is an abstraction of the underlying hardware. It is a machine independent ISA that is compiled to generate machine code. PTX has an unlimited number of registers. As a result, the Open64 phase does not perform register allocation. The allocation of device registers is done in PTXAS phase.

3. Optimization

We added several optimizations in Open64's WOPT phase, and several PTX level transformations in CG. In this section, we present *memory space analysis*, *variance analysis*, and *memory access vectorization*, which we implemented in the Open64 compiler. As addressed in Section 5, many of these are based on prior research contributions. However to the best of our knowledge, our work is the first effort to apply these for compilations targeting a GPU device. We also present some classical optimizations. Finally, we explain how some of these phases may help optimize an example code segment.

3.1. Memory space analysis

The GPU has a hierarchy of address spaces. The compiler generates *specific* memory access instructions for *local*, *shared*, or *global* address spaces. On the Fermi architecture, if the compiler is unable to determine the address space for a memory access, then it generates a *generic* access.

3.1.1. Motivation

We want to resolve address spaces at compile-time because specific memory accesses are faster than the generic versions. In addition, if the compiler is unable to resolve the address space of a memory reference, then it may need to insert a *convert* operation from a specific to a generic address, which incurs overhead. Determining the address space of a memory reference also helps alias analysis and memory disambiguation.

The idea is that two pointers pointing to two different address spaces do not alias. As a result, generating specific memory access instructions is crucial for achieving good performance in CUDA applications.

Without memory space analysis, it may not be possible to determine which memory space a pointer points to. In Figure 1a, if the data objects `D` and `S` reside in the same memory space *M*, and if after inlining function `foo` pointer `p` is also determined to point to an object in memory space *M*, then this analysis may be able to determine that pointer `p` points to memory space *M*, and hence, be able to use specific memory accesses. If `D` and `S` reside in different address spaces, or if the pointer `p` cannot be resolved after the function call, then the memory accesses through `p` will be generic. In the example in Figure 1b, if data objects `X` and `W` are in the same address space, then the accesses to pointer `q` can be specific memory load operations.

CUDA C does not have a way to state what memory space a pointer points to. In Figure 1c, the declaration of `G1` implies that the pointer itself resides in global memory. It does not state what memory space it points to. Similarly, in Figure 1d, there is no way to indicate what memory space the arguments to function `f` point to. As a result, a pointer can escape in situations such as when an address in a specific memory space is assigned to a globally accessible pointer (assigning to `G1` in Figure 1c), or when a pointer is passed as argument to a function (Figure 1d). When a pointer escapes, the compiler may use specific memory accesses *only* if it is able to resolve a particular instance of the escaped pointer access.

3.1.2. Implementation

The analysis pass is a forward data-flow analysis on elements of a lattice. The transfer function is monotone - it moves the state of an expression down a lattice path from $\top$ to a specific memory space, and then to $\perp$. The analysis propagates the memory space of address expressions from a point in the program where the target memory space is known. Typically, the forward flow begins from a point where the address is taken of an object residing in a certain address space. All address expressions start with uninitialized memory space ($\top$). The transfer function is applied to move the element to a specific address space. A *meet* operation between two specific address spaces pushes the expression down to unknown memory space ($\perp$). The analysis completes once it reaches a fixed point. Addresses marked $\perp$ are accessed generically.

3.2. Variance analysis

As we explained in Section 2.1, threads executing a kernel may evaluate an instruction differently if the instruction depends on thread-variant data like thread id. Computations that depend on thread id will evaluate differently potentially generating more thread-variant data. If a branch condition is potentially thread-variant, then threads might diverge at the branch.

3.2.1. Motivation

Variance analysis [12] is used to identify thread-variant and thread-invariant instructions. The goal of variance analysis is to determine the thread-variance state of all expressions. The analysis results of this pass can be utilized by other optimization passes. For example, to minimize the serialized execution of divergent branches, it is important that divergent branch statements are as short as possible. Optimizations such as partial-redundancy elimination may attempt to move computations from straight-line sequence into branches. Such optimizations can utilize variance analysis results to prevent moving code into a branch if it is a divergent branch.

Similarly, jump threading transformation may clone a statement and move it from outside a branch to both branches of a conditional statement. CUDA C provides a textually aligned barrier (*__syncthreads*). Hence, all or none of the threads in a block must execute the same textual barrier instruction. This implies that a barrier instruction must not be cloned and inserted in divergent code sequences to preserve correctness. Variance analysis can be utilized to prevent such transformations.

3.2.2. Implementation

The key insight and property of the CUDA programming model that makes variance analysis possible is that every thread of a kernel reads the same parameters and thread-variant instructions and accesses are easily identified. Examples of thread-variant accesses include reads from thread id and results of atomic instructions.

```
Build forward data flow
Create work-list with initial set of variant values
while work-list is not empty
  /* Traverse data flow to propagate variance state forward */
  do
    Pop element from work-list
    Propagate variance state from def to use
    if variance state of use changed
      push LHS of use to work-list
    end if
  while work-list is not empty

  /* Traverse control dependence graph */
  Propagate variance state from branch condition to \
  expressions in branch
  if variance state of expression changed
    push LHS of expression to work-list
  end if
end while
```

```
for each basic block (BB)
  for each instruction in the BB
    if instruction is a vectorization candidate
      start vector, add access to it
      for each remaining instruction in the BB
        if instruction inhibits vectorization
          stop vector formation
          emit any legal vector already formed
          break
        end if /* vectorization inhibited */
        if can add to vector
          add access to vector
        end if /* access added to vector */
      end for  /* each remaining instruction in BB */
      emit vector if legal
    end if     /* vector candidate? */
  end for      /* each instruction in BB */
end for        /* each BB */
```

Figure 2: (a) Variance analysis algorithm (b) Memory access vectorization algorithm

We compute thread variance by optimistically assuming that every expression and statement is thread-invariant except for the set that is initially required to be thread dependent. We propagate the variance from this initial set to the data and control dependence successors, i.e. the program dependence graph. This is effectively computing the *forward program slice* ([13] [14] [15]) of the initially assumed set of thread-variant instructions. In short, every statement and instruction in the forward slice of thread-variant instructions must be assumed thread-variant and the rest can be assumed thread-invariant. We perform this analysis on SSA-based IR [16] in the WOPT phase (Figure 2a).

3.3. Memory access vectorization

The GPU can support coalescing of per-thread memory accesses into short vectors of two or four elements. For example, instead of a thread executing two 32-bit loads (*ld*) from adjacent addresses, it can execute a single vector load (*ld.v2*) to load 64 bits of data at once. By vectorizing we reduce the amount of memory access latency from multiple separate accesses.

This optimization is performed in the CG phase at the basic block level, operating on PTX-like CG IR (Figure 2b). The ability to coalesce two loads or two stores depends on whether the memory accesses are contiguous. If possible, the object alignment may be increased to enable a vector access. We also have to check whether any intervening instructions clobber or depend on the registers and memory that the potential vector uses. For v vectorizable memory accesses in a basic block with n instructions, the complexity of the algorithm is $O(n \times v)$.

3.4. Other optimizations

Loop unrolling implemented on WHIRL in the WOPT phase proved to be very useful, because it can enable other optimizations. It helps fold computations of loop induction variables. It enables scalar replacement of array and struct accesses. This often makes a considerable difference if it can optimize expensive local memory accesses. Full unrolling also enables vectorization.

Partial redundancy elimination of loads and expressions benefits CUDA performance. PRE of loads (LPRE) [17] can optimize away expensive memory accesses. PRE of expressions (EPRE) reduces redundancies in general, and enables other downstream optimizations.

PRE also often moves computations present in sequential code into branch statements. However, optimizations that in general tend to increase the length of code sequences controlled by a divergent branch should be avoided. Hoisting extra computations out of branches is beneficial in such cases.

3.5. An example

Let us look at a simple example (Figure 3a) of a loop accessing an array residing in __device__ memory, and analyze how some of these optimizations can be applied. Figure 3b shows the unoptimized PTX code for the kernel in pseudo code. Note how the array base-address computations are redundantly performed in each iteration of the loop.

```
// some constant "N"
__device__ int D[N];
__device__ int E[N];

__global__ void kernel(int n,
                       int m)
{
  int *p = n > m ? D : E;
  // some code
  // "k" is thread-variant
  for (int i=0; i<k; ++i)
    sum += p[i] + p[n];
  // code that uses "sum"
}
```

```
convert "D" to generic addr
convert "E" to generic addr
set "p" selecting from "D","E"
// "p" has generic address
Loop::
 load "n" from param space
 compute &"p[n]"
 load generic from &"p[n]"
 compute &"p[i]"
 load generic from &"p[i]"
 perform addition
 increment index
 compare
 branch conditional to Loop
```

```
convert "D" to generic addr
convert "E" to generic addr
set "p" selecting from "D","E"
// "p" has generic address
load "n" from param space
compute &"p[n]"
load generic from &"p[n]"
Loop::
 compute &"p[i]"
 load generic from &"p[i]"
 perform addition
 increment index
 compare
 branch conditional to Loop
```

```
set "p" selecting from "D","E"
// "p" has specific address
load "n" from param space
compute &"p[n]"
load global from &"p[n]"
Loop::
 compute &"p[i]"
 load global from &"p[i]"
 perform addition
 increment index
 compare
 branch conditional to Loop
```

Figure 3: (a) CUDA code (b) Pseudo code for unoptimized kernel (c) After EPRE, LPRE (d) After memory space analysis

Table 1: Summary of performance results

Optimization name	Number of kernels	Number(%) of kernels with improvement	Number(%) of kernels with no change	Number(%) of kernels with slowdown	Minimum improvement(%)	Maximum improvement(%)
Unrolling	521	424 (81.38)	90 (17.27)	7 (1.34)	-6.10	3109.84
EPRE	521	417 (80.04)	100 (19.19)	4 (0.77)	-14.15	1315.80
Memory space analysis	521	213 (40.88)	278 (53.36)	30 (5.76)	-7.68	107.59
LPRE	521	47 (9.02)	430 (82.53)	44 (8.45)	-25.46	29.37
Memory access vectorization	521	21 (4.03)	497 (95.39)	3 (0.58)	-7.72	75.41
Hoisting	521	10 (1.92)	504 (96.74)	7 (1.34)	-6.50	15.83

Performing EPRE optimization on the code segment enables hoisting the computation of the address of `p[n]` out of the loop. Subsequently, running the LPRE optimization hoists the generic load of `p[n]` out of the loop (Figure 3c). Memory space analysis then proves that pointer `p` points to global memory space. As a result, all the load operations from generic address space can be converted to loads from specific (in this case, global) memory space, making the conversion to generic address space at the beginning of the code segment also redundant (Figure 3d). In addition, loop unrolling may enable vectorization of accesses to `p`.

4. Performance Evaluation

In this section, we present an evaluation of classical as well as newly implemented optimizations in the NVIDIA CUDA 4.0 compiler. The performance results demonstrate the importance of these optimizations in any compiler targeting CUDA. This makes all these optimizations as relevant even in the LLVM-based NVIDIA CUDA 4.1 compiler.

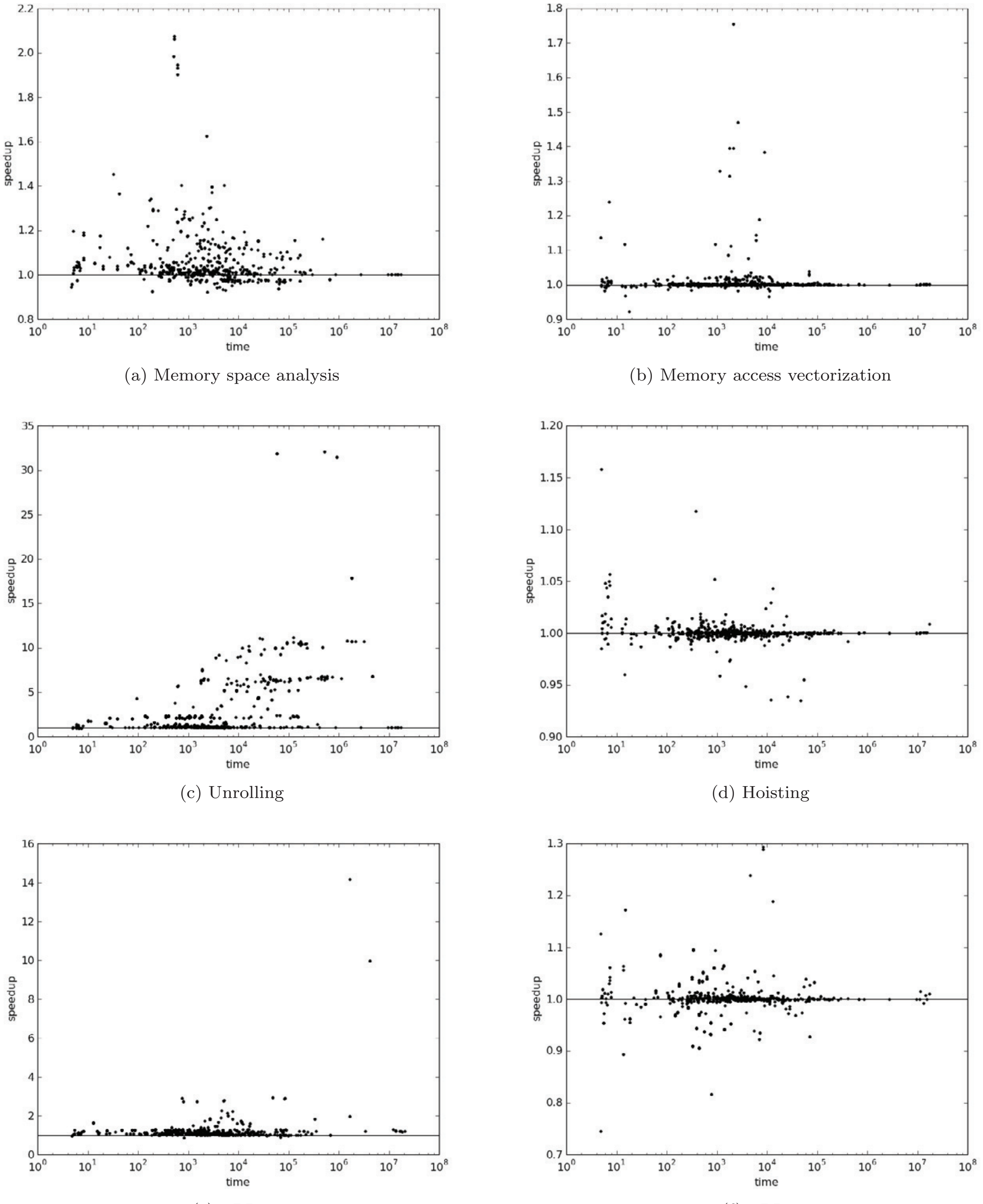
4.1. Experimental Methodology

We evaluated 521 kernels from NVIDIA CUBLAS [18] and CUFFT [19] libraries, and the CUDA C applications developed at NVIDIA. The CUBLAS and CUFFT libraries are CUDA implementations of the BLAS and FFT libraries respectively. We also present performance results from running the CUDA implementation of MD simulations in Amber 11 [20], and the Parboil Benchmark Suite [21]. Our measurements were on a 64-bit Linux (Fedora 10) host with Intel Core2 quad-core CPU and 4GB of memory. Some of the performance results using Amber 11 were taken using NVIDIA Fermi-based Tesla C2050, while all remaining results were taken on the Fermi-based GeForce GTX 560 Ti GPU.

All workloads were built in 64-bit mode. Performance measurements were collected by disabling one optimization at a time and compiling and running the tests to see the effects of that optimization compared to when it was enabled. For each workload we used arithmetic mean of 3 measurements.

4.2. Results on kernels

Table 1 shows a summary of the performance results with a row for each optimization that was tested. The fourth column shows the percentage of kernels on which the effect of the optimization is within a tolerance range of +/- 3%. The third and fifth columns respectively show the percentage of kernels that had an improvement and a slowdown, beyond the tolerance level. The last two columns provide the worst and best improvement in percentage respectively. Figure 4 provides a more detailed view of the performance results for the CUDA kernels. For each of the graphs, the x-axis plots the execution time of the kernels in



(a) Memory space analysis

(b) Memory access vectorization

(c) Unrolling

(d) Hoisting

(e) EPRE

(f) LPRE

Figure 4: Performance improvement: x-axis plots size of workload in number of cycles, y-axis plots speedup.

cycles, and the y-axis plots the speedup for each optimization. As a result, speedup plots above the `1.0` mark are improvements.

Memory space analysis caused significant performance gains (Figure 4a), with a few kernels at a workload size of close to 1000 cycles achieving around 2X improvement. Focusing on the kernel that had a 62% improvement, it also had benefits of 11% and 39% improvement due to this analysis, when invoked with different workloads. This kernel had many memory loads inside loops, which were deduced to be shared memory accesses. In addition, this optimization resulted in reduced register pressure when register allocation was performed in the subsequent PTXAS phase. This can happen due to the benefits of memory space analysis mentioned in Section 3.1.1. The register footprint for this kernel reduced from 39 to 32 registers. As explained in Section 2.1, this improvement in turn increased occupancy of the GPU.

Loop unrolling has a significant impact on performance even with large workloads (Figure 4c), with kernels gaining up to 32X speedup. We analyzed the reasons behind some of the large speedups of around 10X, 17X, and 32X. Many loops in these kernels were fully unrolled, which often improves performance [22]. This transformation resulted in an increase in register pressure, which was offset by the fact that unrolling enabled reduction of the stack frame required by the kernel from 240 bytes to 0 bytes. The kernel had loops that stored array elements, followed by loops that used these array elements. Without unrolling, the loops contained many expensive local memory accesses. With full unrolling of the loops, these accesses were entirely eliminated by using more registers. We timed one of the tests that had the kernel with a 32X improvement. Even though the complete test contained other bottlenecks, the speedup for this kernel improved total runtime by more than 10% from 40s to 36s.

Out of the slowdowns caused by this optimization, all except a 4% slowdown were on kernels with extremely short running times (< 6 cycles). For the 4% slowdown, unrolling caused increase in register usage from 30 to 36 registers, resulting in decreased parallelism.

Large workloads of close to 10000 cycles show around 30% improvement due to PRE of load expressions (LPRE) (Figure 4f). For the kernels that slowed down, LPRE increased the live ranges causing slight increase in register pressure. For one of the kernels that had a relatively large workload and faced 7% slowdown, LPRE increased register usage from 32 to 33, in turn reducing parallelism.

Kernels that had large improvements from LPRE had expensive type conversions on the results of load operations. PRE of these loads enabled commoning of these conversion operations. In addition, loop invariant load expressions in several hot loops were optimized by PRE. These combined optimization scenarios resulted in significant decrease in register pressure and a drastic reduction in register spilling. Savings in register spilling also eliminates costly local memory references.

4.3. Results on large applications

We also evaluated the performance effects of these optimizations on the Amber 11 application "NonSetup CPU time" on a Tesla C2050 based on the Fermi architecture. Out of the 6 optimizations evaluated above, EPRE and memory access vectorization improved the runtimes of this application, while the remaining optimizations did not have any effect. Among the different versions of the runs, EPRE resulted in improvements ranging from 16% to 21%. For the largest workload, EPRE improved the timing from 1179s to 984s. Memory access vectorization caused improvements in the 1-2% range.

We also evaluated performance results using the Parboil Benchmark Suite [21]. The time spent doing GPU computation as provided by the benchmark was used for comparison. 4 of the 6 optimizations resulted in significant performance improvement. Unrolling gave the maximum benefit of up to around 9X, while EPRE resulted in benefits ranging from 4% to 90%. Memory space analysis improved the "mm" benchmark by 19%, while vectorization caused improvements from 4% to 14%.

5. Related Work

CUDA C resembles the SPMD model exemplified in Titanium [23]. Both have a textually aligned barrier that can be used for synchronizing parallel activities and for ordering reads and writes of shared memory. CUDA is based on C/C++ which is inherently unsafe whereas Titanium is based on Java and provides type safety and memory safety. This was a conscious design decision made to allow high performance and interoperability with a rich ecosystem of existing software. On balance this has borne out well, though lack of type and memory safety makes debugging of new parallel software hard. OpenCL [7] has similar goals as CUDA. It is very similar to CUDA, with two key differences. First, CUDA is an integrated heterogeneous language where the host and the device parts form a single program - this makes programming easier. In OpenCL, there is a host program and a separate device program. Second, in OpenCL the memory spaces of data are explicitly part of the type system and it is not possible to mix data of different spaces. This makes design of reusable software hard. In CUDA memory spaces are not part of the type system but act as storage qualifiers.

In this paper, we present a set of optimizations implemented in a CUDA compiler for improving performance of computations that run on the GPU. Other approaches to improving performance of GPU applications include transforming the data-layout to reduce bank conflicts ([24] [25]) and optimizing CPU-GPU communication [26]. These prior research efforts used source-to-source compilation, that applied the transformations and re-generated CUDA C program. On the other hand, our CUDA compiler applies a different set of optimizations on the general purpose computations to be executed on the GPU, and generates PTX code, that can be offline-compiled or JIT-compiled for running on the GPU. As a result, the compiler transformations presented in this paper are complementary to the above mentioned prior work. So our optimizations can be applied on top of the optimizations arising out of other research efforts in the field.

Ryoo et al. present a set of optimization principles that help application performance on the CUDA architecture [22]. There are metrics to prune the optimization space benefiting GPU applications [27]. While the focus of our work is efficient execution of programs on the GPU, there is also a lot of work aimed at transforming a CUDA program for execution on multi-core CPUs ([28] [12]).

The memory space analysis implemented in our compiler can be thought of as a combined analysis [29] which tracks memory spaces of pointers optimistically. So in a sense this is similar to the SSA based optimistic constant propagation [30] where our lattice is the memory spaces treated as constants instead of standard arithmetic constants.

Larsen et al. present a coalescing technique [31] that also works on basic blocks. Their technique differs from us in how they form the vectors, as well as in the usage of dependence analysis to determine coalescing candidates.

The variance analysis described in this paper is based on the algorithm described in [12], where Stratton et al. use the analysis for targeting a CUDA program to multi-core CPUs. Our application of thread variance does not need to track the exact thread dimensions dependence for a variant expression, so in that sense it is a simpler application of their general method.

The rest of the optimizations that are implemented in Open64 are already covered extensively in the existing literature and we mention a few of the important ones here. The PRE algorithm implemented in Open64 is described in detail in [32] and [33]. This algorithm is a SSA based reformulation of the classical lazy code motion [34] method.

6. Conclusions and Future Work

We have presented an overview of the CUDA architecture and programming model that provides an insight into compiler optimizations that are essential for a GPU architecture. We have given a detailed description of our implementation of the high-level optimizer in the widely used CUDA C compiler, including some new optimizations that we have developed. We have presented analyses to explain why these new and other existing optimizations are beneficial for GPUs.

To the best of our knowledge, our work is the first effort to apply these optimizations to compilations of general-purpose computations targeting a GPU device. Moreover, our compilation techniques can be applied on top of existing methodologies for application performance on GPUs. We have presented detailed performance evaluation results that demonstrate the importance of our optimizations in any compiler targeting the CUDA architecture. We observe that our compiler architecture enables general purpose computing to utilize the massive parallelism in today's GPUs.

In the future we would like to investigate how thread divergence should be minimized. We would like various optimization passes to use variance analysis to help keep divergent code sequences to a minimum.

Acknowledgements

We thank the anonymous reviewers for their invaluable feedback.

References

[1] P. Hanrahan, Why is graphics hardware so fast?, in: PPoPP, ACM, New York, NY, USA, 2005, pp. 1–1.
[2] M. Harris, Mapping computational concepts to GPUs, in: SIGGRAPH '05: ACM SIGGRAPH 2005 Courses, ACM, New York, NY, USA, 2005, p. 50.
[3] D. Luebke, M. Harris, N. Govindaraju, A. Lefohn, M. Houston, J. Owens, M. Segal, M. Papakipos, I. Buck, GPGPU: general-purpose computation on graphics hardware, in: SC '06, ACM, New York, NY, USA, 2006, p. 208.
[4] W. R. Mark, R. S. Glanville, K. Akeley, M. J. Kilgard, Cg: a system for programming graphics hardware in a C-like language, ACM Trans. Graph. 22 (3) (2003) 896–907.
[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, P. Hanrahan, Brook for GPUs: stream computing on graphics hardware, in: SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, ACM, New York, NY, USA, 2004, pp. 777–786.
[6] C. Boyd, DirectX 11 DirectCompute: A teraflop for everyone (2010).
[7] Khronos OpenCL Working Group, The OpenCL Specification (May 2009).
[8] J. Nickolls, I. Buck, NVIDIA CUDA software and GPU parallel computing architecture, Microprocessor Forum (May 2007).
[9] NVIDIA, NVIDIA CUDA Compute Unified Device Architecture Programming Guide: Version 3.2, NVIDIA Corporation (November 2010).
[10] G. Gao, J. Amaral, J. Dehnert, R. Towle, The SGI Pro64 compiler infrastructure. Tutorial.
[11] NVIDIA, PTX: Parallel Thread Execution ISA Version 2.2, NVIDIA Corporation (October 2010).
[12] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, W. mei W. Hwu, Efficient compilation of fine-grained spmd-threaded programs for multicore cpus, in: CGO, 2010, pp. 111–119.
[13] M. Weiser, Program slicing, IEEE Trans. Software Eng. 10 (4) (1984) 352–357.
[14] F. Tip, A survey of program slicing techniques, J. Prog. Lang. 3 (3).
[15] J.-F. Bergeretti, B. Carré, Information-flow and data-flow analysis of while-programs, ACM Trans. Program. Lang. Syst. 7 (1) (1985) 37–61.
[16] F. C. Chow, S. Chan, S.-M. Liu, R. Lo, M. Streich, Effective representation of aliases and indirect memory operations in SSA form, in: Proceedings of the 6th International Conference on Compiler Construction, Springer-Verlag, London, UK, 1996, pp. 253–267.
[17] F. C. Chow, R. Kennedy, S.-M. Liu, R. Lo, P. Tu, Register promotion by partial redundancy elimination of loads and stores, in: PLDI, 1998, pp. 26–37.
[18] NVIDIA, CUBLAS Library User Guide, NVIDIA Corporation (August 2010).
[19] NVIDIA, CUFFT Library User Guide, NVIDIA Corporation (August 2010).
[20] D. Case, T. C. III, T. Darden, H. Gohlke, R. Luo, K. M. Jr., A. Onufriev, C. Simmerling, B. Wang, R. Woods, The Amber biomolecular simulation programs, J. Computat. Chem. 26 (2005) 1668–1688.
[21] IMPACT Research Group, UIUC, Parboil benchmark suite version 2.0.
URL `http://impact.crhc.illinois.edu/parboil.php`
[22] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. Kirk, W. W. Hwu, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, in: PPoPP, 2008.
[23] A. Krishnamurthy, A. Aiken, P. Colella, D. Gay, S. L. Graham, P. N. Hilfinger, B. Liblit, C. Miyamoto, G. Pike, L. Semenzato, K. A. Yelick, Titanium: A high performance java dialect, in: PPSC, 1999.
[24] I.-J. Sung, J. Stratton, W. mei Hwu, Data layout transformation exploiting memory-level parallelism in structured grid many-core applications, in: PACT, 2010.
[25] Y. Yang, P. Xiang, J. Kong, H. Zhou, A gpgpu compiler for memory optimization and parallelism management, in: PLDI, 2010, pp. 86–97.
[26] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, D. I. August, Automatic cpu-gpu communication management and optimization, in: PLDI, 2011, pp. 142–151.
[27] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, W. mei W. Hwu, Program optimization space pruning for a multithreaded GPU, in: CGO, 2008.
[28] J. A. Stratton, S. S. Stone, W. mei W. Hwu, MCUDA: An effective implementation of CUDA kernels for multi-core CPUs, in: Proceedings of the 21st International Workshop on Languages and Compilers for Parallel Computing, 2008, pp. 16–30.
[29] C. Click, K. D. Cooper, Combining analyses, combining optimizations, ACM Trans. Program. Lang. Syst. 17 (2) (1995) 181–196.
[30] M. N. Wegman, F. K. Zadeck, Constant propagation with conditional branches, ACM Trans. Program. Lang. Syst. 13 (2) (1991) 181–210.
[31] S. Larsen, S. Amarasinghe, Exploiting superword level parallelism with multimedia instruction sets, in: PLDI, 2000, pp. 145–156.
[32] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, F. Chow, Partial redundancy elimination in SSA form, ACM Trans. Program. Lang. Syst. 21 (3) (1999) 627–676.
[33] F. Chow, S. Chan, R. Kennedy, S. ming Liu, R. Lo, P. Tu, A new algorithm for partial redundancy elimination based on SSA form, in: PLDI, 1997, pp. 273–286.
[34] J. Knoop, O. Rüthing, B. Steffen, Lazy code motion, SIGPLAN Not. 27 (7) (1992) 224–234.