January 2016

Programming models, compilers, and runtime systems for accelerator computing

Amit Sabne
Purdue University

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Amit Sabne

Entitled Programming Models, Compilers, and Runtime Systems for Accelerator Computing

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

RUDY EIGENMANN	
MILIND KULKARNI	
ANAND RAGHUNATHAN	
SAMUEL P. MIDKIFF	

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): RUDY EIGENMANN

Approved by: V. Balakrishnan 07/21/2016

Head of the Department Graduate Program Date

PROGRAMMING MODELS, COMPILERS, AND RUNTIME SYSTEMS FOR ACCELERATOR COMPUTING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Amit J. Sabne

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2016

Purdue University

West Lafayette, Indiana

Dedicated to Ramprasad Joshi who cultivated my inquisitiveness

ACKNOWLEDGMENTS

First and foremost, I thank my advisor, Prof. Rudolf Eigenmann, whose unconditional support made this dissertation possible. His remarkable patience and flexibility helped me navigate through the process so much so I barely realized that he was physically away for half the duration. I admire his willingness to provide me complete independence in my research and his wisdom that got me out of dead ends from time to time.

Many people kept me motivated through the PhD process. I thank my committee members Prof. Samuel Midkiff, Prof. Anand Raghunathan, and Prof. Milind Kulkarni for the inspiring interactions. I thank Prof. Charles Bouman, Xiao Wang, and Jordan Kisner for their help through the MBIR project. I am grateful to Prof. T. N. Vijaykumar for his insightful comments in the HeteroDoop project. My present and former labmates have been a useful bunch to brainstorm with. I thank them all: Putt Sakdhnagool, Okwan Kwon, Fahed Jubair, Hansang Bae, Aurangzeb, and Tsung-Tai Yeh. I am grateful to my collaborator-turned-mentors, Prof. Jeffrey Vetter and Seyong Lee from the Oak Ridge National Laboratory for their valuable guidance.

The PhD process required a lot of personal support too. I thank my parents, my sister, and Kanak, my girlfriend, to have shown unwavering faith in me and to have sacrificed a lot over the years. I was fortunate to have been roommates with my old friends from the undergraduate years: Rohan Gandhi, Neeraj Gadgil, and Saikumar Reddy. I thank all my friends – the list is too long to be mentioned – for their immense support.

I am grateful to the J. N. Tata Endowment for awarding me the scholarship for graduate studies.

Finally, I must thank the faculty and staff of the ECE department, as well as Purdue University, to have offered me the PhD opportunity and to have made my stay here a joyful ride.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Sabne, Amit J. PhD, Purdue University, August 2016. Programming Models, Compilers, and Runtime Systems for Accelerator Computing. Major Professor: Rudolf Eigenmann.

The last decade has seen a growing trend in the use of accelerators for high-performance computing. Accelerators are computationally powerful, massively parallel co-processors. Examples of accelerators include NVIDIA CUDA GPUs [1], AMD GCN GPUs [2], Intel Xeon Phi [3] (Many Core Processors), and Cell Broadband Engine [4]. Accelerators gain most of their performance through massive parallelism. A key challenge hindering the widespread use of accelerators is their programmability. The architectural complexity of these accelerators necessitates explicitly parallel programming models and creates a multitude of tuning avenues that must be explored to obtain high performance. This thesis proposal offers improved programming models, compilation techniques, and runtime systems to make accelerator programming easy, while bettering/retaining the performance.

In this abstract, we first briefly describe some of these accelerator architectures. Due to the massive parallelism, accelerators typically rely on explicitly parallel programming models, which we describe next. Lastly, the abstract provides an overview of open issues in accelerator programming, and the contributions made by this thesis to address them.

Accelerator Architectures

Graphical Processing Units (GPUs) were the first accelerator architectures to have received widespread attention. The advances in GPU technologies led to their instruction set to resemble the ones of CPUs, creating an opportunity for general-purpose computations on GPUs [5]. A primary motivation for using GPUs for general-

purpose computation was the massive parallelism imbibed in them. GPUs work in an SIMD fashion, where thousands of threads can co-exist on the chip. GPU architectures comprise many light-weight cores, which are computationally much less powerful than the CPU cores. However, due to a large number of such cores, the achievable throughput was much higher than the CPU.

This throughput-oriented paradigm also gave birth to other accelerator architectures, such as Cell Broadband Engine and Intel Xeon Phi processors. These architectures were immediately adopted by the high-performance computing community and many supercomputers started to gain a high percentage of their computational power through the accelerators.

Typical high-performing accelerators are connected as coprocessors to the host CPU in the system, through a PCIe bus. The main memory of the accelerator is separate from the CPU. Further, the accelerator architectures possess specialized memory spaces that can be used to boost the program performance.

Programming Accelerator Architectures

The architectural distinctions of accelerators necessitate specialized programming models. First, to utilize the massive parallelism, explicitly parallel programming models, such as CUDA [1] and OpenCL [6] are necessary. In order to obtain acceleration, the programmer must identify the compute-intensive, parallel code regions and offload them on to the accelerator as *kernels*. Secondly, since accelerators have discrete memories, input data must be copied from the CPU memory into the accelerator, and vice-versa for the output data. These data transfers need to be handled by the programmer. Both CUDA and OpenCL offer necessary APIs for data transfers. Recently, there have been attempts to unify the CPU-accelerator memory spaces. Although such mechanisms improve accelerator programmability, they suffer on the performance, owing to page-level data handling. For HPC uses, programmers still manually perform the data transfers. Lastly, these architectures have unique memory hierarchies. CUDA, which is specific to NVIDIA GPUs, provides explicit functionalities to utilize on-chip memories, such as shared, texture and constant memories. It also offers architecture- specific functions for using vector units, performing warp-operations etc.

Despite the availability of CUDA and OpenCL, programming accelerator architectures remains difficult and error prone, owing to the complexity of these models. Programmers need to rewrite their applications to make use of the accelerators. To obtain better programmability, researchers created more user-friendly, high-level programming models, such as OpenMPC [7] and hiCUDA [8]. The former offers an OpenMP to CUDA translation system with additional, CUDA-specific directives added to OpenMP. The latter presents directives to be placed on a serial program, and a compiler that translates the annotated program into a CUDA program. Such high-level programming models have now been standardized. OpenACC [9] is the first such OpenMP-like programming standard for accelerators. Commercial compilers [10–12] are presently available for OpenACC. Recently published OpenMP 4.0 [13] extends the OpenMP standard to accelerators.

Thesis Contributions

While the high-level programming models offer better programmability, programmers still face several issues. We highlight such open issues below, and briefly summarize our contributions to address them.

- **Program tuning** : Several factors affect the program performance in accelerator programs. These factors include compiler optimizations (e.g. parallel loop interchange, data transfer reduction) and user specifications (e.g. number of threads to use, CPU-accelerator synchronization) [14–16]. Due to the presence of a large number of these factors, achieving the best performance requires tuning, which may be performed by an automatic tuning system, or manually by the programmer. In Chapter 1, we evaluate the effects of compiler optimizations on OpenMP to CUDA translation. This work is built on top of the OpenMPC [7] system, which comprises an auto-translator from OpenMP to CUDA. OpenMPC offers many compiler optimizations and an auto-tuning system for the same. This auto-tuner performs a pruned-exhaustive search among the possible program configurations. Chapter 1 will present a more advanced tuning approach that reduces the search time-complexity to a polynomial time, and finds better program configurations than the pruned-exhaustive search approach. With this new tuning method, we analyze the performance effects of each compiler optimization.

- **Limited accelerator memory sizes** : Accelerators typically use non-virtualized memories. Therefore, the programmer must ensure that the data size used by the problem fits inside the main memory. Otherwise, the programmer must carry out manual computation blocking so that the data fits within the accelerator memory. The available automatic solutions, such as zero-copy memory, or unified virtual memory [1], require excessive CPU-GPU communication while the kernel is in flight, lowering the performance. Chapter 2 addresses this issue of limited memory spaces on accelerators. It presents a novel compiler technique, called *Computation Splitting* or *COSP* [17] that automatically splits the computation so as to make the problem fit in the accelerator memory, without lowering the performance.

- **Multi-GPU scaling** : Many supercomputers comprise of nodes containing multiple GPUs. The previous solutions to exploit all GPUs of a node required manual computation partitioning [18, 19]. There existed one automatic approach that provides a single device image for all GPUs [20]. However, the computation size in such an approach is restricted by the combined memory of all the GPUs. The COSP technique in Chapter 2 generates data-independent subproblems. The chapter describes how these subproblems can be mapped to multiple GPUs attached to a node, overcoming the memory size barrier.

- **Computation-communication overlap** : As the input data must be copied in the accelerator memory before the computation, and the output data must be copied out from the accelerator memory to the CPU memory, an opportunity arises to overlap this communication with the kernel computations. Programmers must manually obtain this overlapping behaviour. Chapter 2 will present an automatic mechanism to perform such pipelining and a runtime tuning mechanism to obtain a high performing pipeline stage size. This is in contrast to previous approaches [21–24] where pipelining was exploited on the GPU to hide device memory latency by copying data into the on-chip programmer-managed cache first.

- **Accelerator clusters** : MapReduce is a popular programming framework for distributed computing. However, when it comes to accelerator clusters, available systems either use CPUs [25–28], or GPUs [29–31]. Since CPUs perform better on IO-intensive applications, while GPUs perform better on the compute-intensive ones, a need of a framework that uses both CPUs and GPUs arises to cater to all possible workloads. Chapter 3 will present HeteroDoop, a MapReduce programming system that exploits both CPUs and GPUs in a cluster. HeteroDoop [32] programs require simpler modifications to already available CPU-only MapReduce programs, unlike other GPU-MapReduce frameworks that require explicit parallelization or custom APIs [30, 33, 34].

- **Control flow divergence** : The SIMD execution model employed by accelerators mandates the set of threads/lanes in the SIMD to execute the same instruction. If the threads encounter a branch wherein some threads take the **`if`** while others take the **`else`** path, then the execution of these two divergent outcomes gets serialized. The lockstep execution can only resume once the immediate post-dominator (IPDOM) of the divergent branch is reached. Such divergence can worsen the program performance under the presence of "unstructured" control flow [35]. Unfortunately, there is little formalization available on what unstructured control flow means. Chapter 4 will formalize the notion of structured control flow graphs. The chapter will also present an unstructured-to-structured conversion algorithm which eliminates the exponential code expansion possible in previous approaches [36–40]. Our control flow structuring approach improves divergent code execution on SIMD units, to which no software solution was previously available [35].

1. EFFECTS OF COMPILER OPTIMIZATIONS IN OPENMP TO CUDA TRANSLATION

1.1 Introduction

OpenMP is an established standard in parallel programming and is of particular interest for today's and future multicores. There is a large and growing code base, the standard is widely understood and is well documented, and there exists a multitude of compilers and supporting tools. These features are of paramount importance to the programmer. They help reduce the difficulty and the cost of developing parallel software significantly.

The number of new parallel languages that have been proposed in even just the past two decades is massive. The question of cost versus benefit arises with every such proposal. Unfortunately, few quantitative analyses are available that would allow one to find out if the same objective could have been achieved with an existing language standard and what are costs and benefits of new versus old, in terms of performance and productivity. Obviously, any new language will start from zero in building a code base, compilers, tools, and programming experience.

A new language development has emerged in the context of new graphics processing units, or accelerators. These devices offer promising avenues towards low-energy, highly parallel computation for a class of parallel applications. Among the proposed programming models are CUDA and OpenCL, both of which allow the programmer to access architecture-specific features. These architecture-specific interfaces, however, significantly depart from the parallel programming semantics offered by standards, such as OpenMP. The cost/benefit question arises anew.

Previous work has addressed this cost/benefit question [41]. This work provided quantitative comparisons of hand-written CUDA programs versus equivalent programs written in OpenMP and translated to CUDA. Using an automatic translator and tuning system, called OpenMPC, we were able to achieve performance results that came close to hand-coded CUDA on a large set of benchmarks. The contribution of the present chapter is to address three open issues of that work.

- The previous work provided overall performance numbers. The breakdown into individual techniques was not yet available. In this chapter, we quantify the contributions of each individual technique. Of particular interest in this analysis is also the importance of CUDA-specific OpenMP extensions, which are generated automatically in the OpenMPC system.

- A key component of the OpenMPC is its tuning system, which empirically searches through a large space of optimization variants and tries to find the best. The initial OpenMPC system used an inefficient exhaustive search mechanism. In this work, we use an improved *Iterative Elimination (IE)* [42] based tuning system that significantly reduces the tuning time.

- A problem faced by all empirical tuning systems is the variability of execution times, even for the same program executed repeatedly on the same platform in single-user setup. This effect makes it difficult to correctly measure the impact of an optimization technique. A common approach is to average over multiple runs, increasing tuning time. We have developed a new method that identifies optimizations that are vulnerable to runtime variations and uses increased measuring time only for those.

The remainder of the chapter is organized as follows. Section 1.2 describes OpenMPC and its available optimization options. It also identifies opportunities for improvement in the present OpenMPC tuning system. Section 1.3 explains our tuning mechanism for finding the best tuning options. Individual performance analysis is shown in Section 1.4. Section 4.9 presents the takeaways from this chapter.

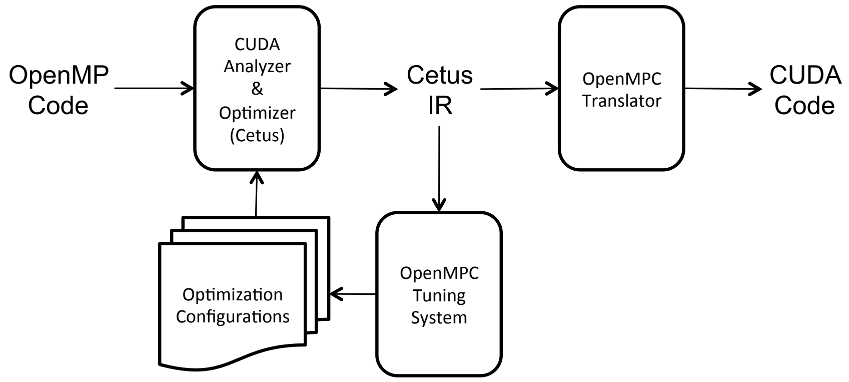


Figure 1.1.: OpenMPC workflow

1.2 Overview of OpenMPC System

OpenMPC [41] is a programming framework that generates CUDA programs from OpenMP programs. The framework includes an extended OpenMP programming interface, a source-to-source translator, and an automatic tuning system. The programming interface extends OpenMP with a new set of directives and environment variables (henceforth referred to as CUDA extensions[1]) for controlling CUDA-related parameters and optimizations. OpenMP translates standard OpenMP programs by applying a set of program transformations and by inserting CUDA extensions. OpenMPC includes an empirical tuning system that automatically generates, prunes, and searches the optimization space and determines the best combination of optimizations. Fig. 1.1 shows the workflow of the OpenMPC translator. Fig. 1.2 displays a small example of the OpenMPC translated CUDA code for Jacobi benchmark.

1.2.1 Optimization options

There are 18 optimization options available in OpenMPC, grouped into 4 categories: (1) Program environment configuration, (2) Data caching strategy, (3) Data offloading optimizations, and (4) Code transformation. Table 1.1 shows all optimiza-

[1]Our CUDA extensions are not meant to be a proposal for extending the OpenMP standard. They represent a research framework for exploring questions such as those addressed in this chapter.

Listing 1.1: Source code in OpenMP

```
#pragma omp parallel for private(i, j)
for (i = 1; i <= SIZE; i++){
  for (j = 1; j <= SIZE; j++){
    a[i][j]=(b[i-1][j]+b[i+1][j]+b[i][j-1]+b[i][j+1])/4.0f;
  }
}
```

Listing 1.2: Resulting CUDA kernel from OpenMPC translation

```
__global__ void kernel(...){

  int _bid = (blockIdx.x+(blockIdx.y*gridDim.x));
  int _gtid = (threadIdx.x+(_bid*blockDim.x));
  tid = (_gtid+1);

  if (tid<=SIZE){
    for (i=1; i<=SIZE; i ++ ){
      a[i][j] = ( b[i-1][tid]+ b[i+1][tid] +
        b[i][tid-1] + b[i][tid+1] )/4.0F;
    }
  }
}
```

Figure 1.2.: OpenMPC translation example

tion options in OpenMPC that are considered for individual optimization analysis. The first three groups are supported by our CUDA extensions. The fourth group is applied through source-to-source transformation in the OpenMPC compiler.

1.2.2 Improving the OpenMPC Tuning System

To analyze the effects of individual tuning options, we make use of the OpenMPC system, which allows us to implement the method in [43]: Using the highest-optimized program variant as a baseline, this method iteratively switches off one optimization at a time, to measure its effect in terms of the slowdown incurred. To this end, we have identified a number of open issues in OpenMPC, which we address in the present work.

Advanced Optimization Space Navigation:

The goal of an empirical tuning system is to generate a set of optimizations that yield best performance. In OpenMPC, 18 optimizations are available as compiler flags. Finding the best combination from these flags is non-trivial, because each optimization can improve or worsen the performance of a program, depending upon its characteristic and depending on other present optimizations.

The initial OpenMPC system uses simple *exhaustive search* to navigate the space of optimization variants. This space can be very large (for n on-off optimization options, the size is 2^n). OpenMPC reduces this space using aggressive tuning heuristics, which we refer to as *pruned exponential search (PE)*. PE does the program analysis to prune the tuning space by removing the inapplicable or non-beneficial tuning options for the particular program. It then runs exhaustive search over the remaining tuning options. However, two issues remain: The resulting search space can still be large (which was acceptable for obtaining the original research results [41], but can be too

Table 1.1.: Optimization options in OpenMPC

Program Environment Configuration

Compiler Flags	Description
cudaThreadBlockSize=N	Set the default CUDA thread block size
assumeNonZeroTripLoops	Assume that all loops have non-zero iterations

Data Caching Strategy

Compiler Flags	Description
shrdSclrCachingOnReg	Cache shared scalar variables onto GPU register
shrdArryElmtCachingOnReg	Cache shared array elements onto GPU register
shrdSclrCachingOnSM	Cache shared scalar variables onto GPU shared memory
prvtArryCachingOnSM	Cache private array variables onto GPU shared memory
shrdArryCachingOnTM	Cache 1-dimensional, R/O shared array variables onto GPU texture memory
shrdSclrCachingOnConst	Cache R/O shared scalar variables onto GPU constant memory
shrdArryCachingOnConst	Cache R/O shared array variables onto GPU constant memory

Data Offloading Optimization

Compiler Flags	Description
useMallocPitch	Use cudaMallocPitch() for 2-dimensional arrays
useGlobalGMalloc	Allocate GPU variables as global variables which provides more scope for reducing memory transfers
globalGMallocOpt	Apply CUDA malloc optimization for globally allocated GPU variables
cudaMallocOptLevel=N	Set CUDA malloc optimization level for locally allocated GPU variables
cudaMemTrOptLevel=N	Set CUDA CPU-GPU memory transfer optimization level

Code Transformation

Compiler Flags	Description
localRedVarConf=N	Configure how local reduction variables are generated for array-type variables
useMatrixTranspose	Apply Matrix Transpose optimization
useParallelLoopSwap	Apply Parallel Loop Swap optimization
useUnrollingOnReduction	Apply Loop Unrolling for in-block reduction

Table 1.2.: Variations on GPU Programs

Benchmark	Relative Standard Deviation for Memory Transfer Time (A)	Relative Standard Deviation for Computation Time (B)	Ratio (A/B)
NW (8192)	0.2395	0.0128	18.71
Jacobi (12288)	0.7394	0.0001	7394
CG (W)	0.2562	0.0706	3.63
FT (W)	0.1521	0.0112	13.58

long for end users). In addition, sometimes the aggressive pruning heuristics may eliminate the best optimization combination.

Runtime Variations – A Key Problem of Auto-Tuning Systems:

In computer systems, unpredictable system variations during program execution are usual. They arise due to OS overheads, other running processes, or underlying hardware operations. Although these variations do not affect the correctness of the program, they can impact its execution time. We define this type of variation as *runtime variation*. Although runtime variation does not disrupt program execution, in auto-tuning system, runtime variation can be problematic. Since the auto-tuning systems improves the program based on execution time, the variation can cause some beneficial optimizations to be removed from the tuning result and vice versa.

One of the significant observations made during our study was the fact that most of the variations on GPU programs are due to the variations in memory transfer times. Since GPU and CPU do not share a common address space, memory transfers form an essential part of GPU programs. GPUs are generally connected to the CPU using a PCIe bus, thereby leading to a variability in the memory transfer times. Table 1.2 compares the relative standard deviation in computation time and the memory transfer time. Relative standard deviation is a percentage of the ratio of standard deviation to the mean of the sample. It acts as an indicator as of how the variations relate to the average. From Table 1.2, we can see that the relative standard deviations in memory transfer can be as much as 7000 times the relative standard deviations in computation time.

To alleviate runtime variations, one can average execution times across multiple runs. However, multiple executions can increase the tuning time significantly. The PE algorithm does not take runtime variations into consideration, and therefore is more prone to erroneous final option combinations on GPU programs.

Objectives of this Work:

Our goal is to determine the impact of individual optimization techniques in the OpenMP to CUDA translator. To this end, we use the improved OpenMPC translation and tuning system, which can find the best combination of optimization techniques for each program. In doing so, it also reports the performance difference made by individual optimizations. We proceed as follows.

- We modify a previously described *Iterative Elimination (IE)* [42] tuning algorithm to make it applicable to GPU programs.
- We describe a generic tuning methodology to deal with memory transfer time based variations of GPU applications.
- With the best tuning option combination generated by the above tuning system, we analyze the impact of each tuning option or compiler flag.

The next section presents the new tuning algorithm. Section 1.4 presents results obtained using this methodology.

1.3 Modified IE (MIE) Algorithm for OpenMPC

To address the issues presented in Section 1.2.2, we propose a *Modified IE (MIE)* algorithm, which is a tuning algorithm based on *Iterative Elimination (IE)* [42]. In this section, we briefly describe IE and then present our MIE algorithm.

1.3.1 Iterative Elimination

The IE algorithm is shown in Algorithm 1. IE begins by switching on all optimization options, and then iteratively measures their effect by switching off one tuning option at a time. Next, it removes the one with the most negative effect. The process repeats until all remaining optimizations show non-negative effects. The complexity of IE is $O(n^2)$, compared to $O(2^n)$ of the *PE* algorithm.

Algorithm 1: Iterative Elimination Algorithm

```
   Input: $n$ = Number of Tuning Options ($F_1, F_2, \ldots F_n$)
   Output: $B = \{F_1 = 1, F_2 = 1, \ldots, F_n = 1\}$ B is a set of combination options
1  $i \leftarrow 1$; $NextB \leftarrow B$;
    // $NextB$ stores the fastest combination in every iteration
2  for ($i = 1 \rightarrow n$) do
3      for ($j = 1 \rightarrow n$) do
4          if ($F_j \neq 0$) then
               // Compares the runtimes
5              $NextB = \min(NextB,\ B \text{ with } F_j = 0)$;
       // Termination: No $F_i$ has changed from 1 to 0
6      if ($NextB = B$) then
           // None of the switched on options has a negative
           impact
7          break;
8      $B \leftarrow NextB$;
       // Start next iteration with a new baseline $NextB$
```

Another tuning method, *Combined Elimination (CE)* [42] performs the option removal in a more aggressive fashion, under the assumption that some interferences between options are negligible. The tuning time of CE is known to be shorter than IE. However, since the performance of IE is known to be the best amongst the available tuning algorithms [42], we chose IE as our base algorithm. Other algorithms could be adapted in place of IE in our system [44, 45]. Unlike the work in [46], which uses optimal ordering of compiler flags, IE tries to find the best tuning options set, irrespective of the order.

1.3.2 Grouping of different Optimization Options

To deal with the problem of runtime variations, a direct implementation of IE would require multiple runs and averaging before eliminating an optimization option. This would lead to high tuning times, because the runtime variations of GPU programs can be large.

Comparing only the computation runtime instead of the total execution time can eliminate the effect of memory transfer variations on tuning. To achieve that, the behavior of memory transfers must be the same between two comparable candidate combinations of IE. If this invariant is maintained, the memory transfer time can be subtracted from total execution time (e.g., by obtaining these times from available hardware profilers) an optimization technique is evaluated by IE.

An intuitive strategy would be to apply techniques that affect memory transfers (i.e. data offloading optimizations shown in Table 1.1) in a first tuning phase, averaging the results over multiple runs. In a second phase, the remaining optimization options are tuned, whereby transfer times are removed from execution times. In this way, most of the runtime variations in the GPU program can be filtered out; a single run suffices.

The split into the two phases is beneficial only when the data offloading optimizations do not interfere with other. That is not always the case. For example,

Table 1.3.: Grouping of OpenMPC Options for Tuning (MemTR = Memory Transfer Optimization, Comp = Computation Optimization). Options in paranthesis imply multi-values options

Phase	Type	Tuning Options
1	MemTR	useGlobalGMalloc, globalGMallocOpt, cudaMallocOptLevel=1, cudaMemTrOptLevel=2
2	Comp	useUnrollingOnReduction, useLoopCollapse, useMatrixTranspose, useParallelLoopSwap, prvtArryCachingOnSM, localRedVarConf=0, assumeNonZeroTripLoops
3	Dependent	useMallocPitch
4	Independent	ArrayCache = {shrdArryElmtCachingOnReg, shrdArryCachingOnTM, shrdArryCachingOnConst} ScalarCache = {shrdSclrCachingOnReg, shrdSclrCachingOnSM, shrdSclrCachingOnConst}

useMallocPitch, which manages 2D array allocation and transfer, may or may not be beneficial depending on the stride of 2D array accesses. Since *useParallelLoopSwap* transforms the array accesses in the code, *useMallocPitch* may improve performance if *useParallelLoopSwap* is applied.

To address this problem, *MIE* uses a third phase, in which memory transfer optimizations that are affected by computation optimization options are placed. This phase also averages runtimes over multiple runs. In a fourth phase, MIE tunes separately those optimizations that do not interact with others. It uses a simple, fast tuning algorithm for this phase.

Phase 1 contains all memory transfer-based (data offloading) optimizations, except useMallocPitch. *Phase 2* contains program environment configuration and code transformation options that impact the computation. *Phase 3* contains dependent optimizations. With the currently available tuning options in OpenMPC, *Phase 3* contains only *useMallocPitch*. This technique impacts the data offloading (memory transfers), but is dependent upon computation technique *useParallelLoopSwap*.

Phase 4 contains data caching optimizations. They are independent of the techniques in the other groups.

1.3.3 MIE Running Strategy

With the above groups of optimizations in place, we now describe the MIE run strategy.

1. **Data Offload Optimizations**: First the algorithm runs IE with the *Phase 1* optimizations as the input set. Since these options all impact memory transfers, they are vulnerable to high runtime variations. The MIE algorithm runs each IE stage multiple times and considers the average execution times for making elimination decisions.

2. **Computation Optimizations**: The configuration formed in *Phase 1* is the baseline configuration. MIE now appends *Phase 2* options to this configuration and runs IE over all new options. While making comparisons between two combinations, the memory transfer time is removed from the comparison, effectively considering only the computation time. This helps reduce the effect of variations to a large extent. This stage requires calculation of the time spent in copying the data between CPU and GPU memories. This is accomplished by using the CUDA profiler. Using this method, MIE avoids averaging over multiple runs, substantially reducing the time required.

3. **Dependent Optimizations**: In the combination formed after *Phase 2*, MIE includes *Phase 3* option i.e. *useMallocPitch* and averages the runtimes over multiple executions to see if this option is beneficial and should be included. (Should there be more tuning options added in *Phase 3*, MIE would run IE on this group, with averaging runtimes over multiple executions.)

4. **Independent Optimizations**: Since this group does not depend upon other options, MIE iteratively runs each *Phase 4* option on top of the configuration formed in *Phase 3*, and adds the best value of each multi-valued option to the final optimization configuration.

1.4 Performance Analysis

1.4.1 Setup

We ran both the PE and the MIE algorithm on NVIDIA Quadro FX 5600 GPU device, which has 16 multiprocessors (SMs) clocked at 1.35GHz and 1.5 GB of memory. Each SM consists of 8 SIMD processing units (SPs) and has 16 KB of shared memory. The host CPU is a 3-GHz AMD dual-core processor with 12 GB memory. The OpenMPC generated CUDA programs were compiled using the NVIDIA CUDA Compiler (NVCC) with option -O3.

We demonstrate the effectiveness of our tuning system on NAS OpenMP Parallel benchmarks, Rodinia OpenMP benchmarks and some scientific computation applications. As described in 1.3.3, we run *Phase 1* and *Phase 3* options 5 times each and use the average runtimes for IE. For other groups, we compare only the computation times for IE runs.

1.4.2 Performance Comparison Between Pruned Exhaustive and Modified IE Algorithms

To evaluate the performance of the MIE algorithm, we show in Figure 1.3 the speedup of benchmarks achieved with MIE, normalized with respect to the PE algorithm. MIE performs better than the PE algorithm in most of the cases, averaging to a 11% performance improvement over PE. In fact, MIE outperforms Pruned Ex-

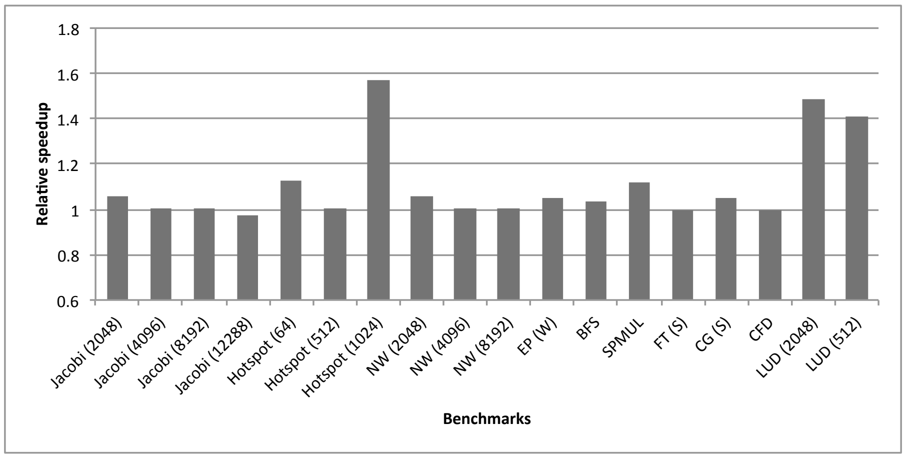


Figure 1.3.: Program Speedups of Modified IE relative to Pruned Exhaustive Algorithm

haustive method substantially for the Hotspot and LUD benchmarks. This effect is due to the over-pruning occurring in the PE method, thereby missing out on the best option combination.

Another important observation is the fact that MIE performs marginally better (2 to 5 %) compared to Pruned Exhaustive method on most other programs where over-pruning does not happen. This is counter-intuitive since PE is expected to search through all possible choices. It is explained due to the excessive memory transfer based variations, wherein the best option combination produced by the Pruned Exhaustive method may not be the optimal, rather it is the one that suffered the least.

Table 1.4 compares the tuning time required by the Pruned Exhaustive algorithm against the tuning time required by the MIE algorithm. The advantage of IE in terms of tuning time is evident from this table.

1.4.3 Impact of Individual Optimization Options

As stated earlier, to analyze the effect of individual tuning options in OpenMPC, we follow the method from [43], wherein we turn off one optimization at a time

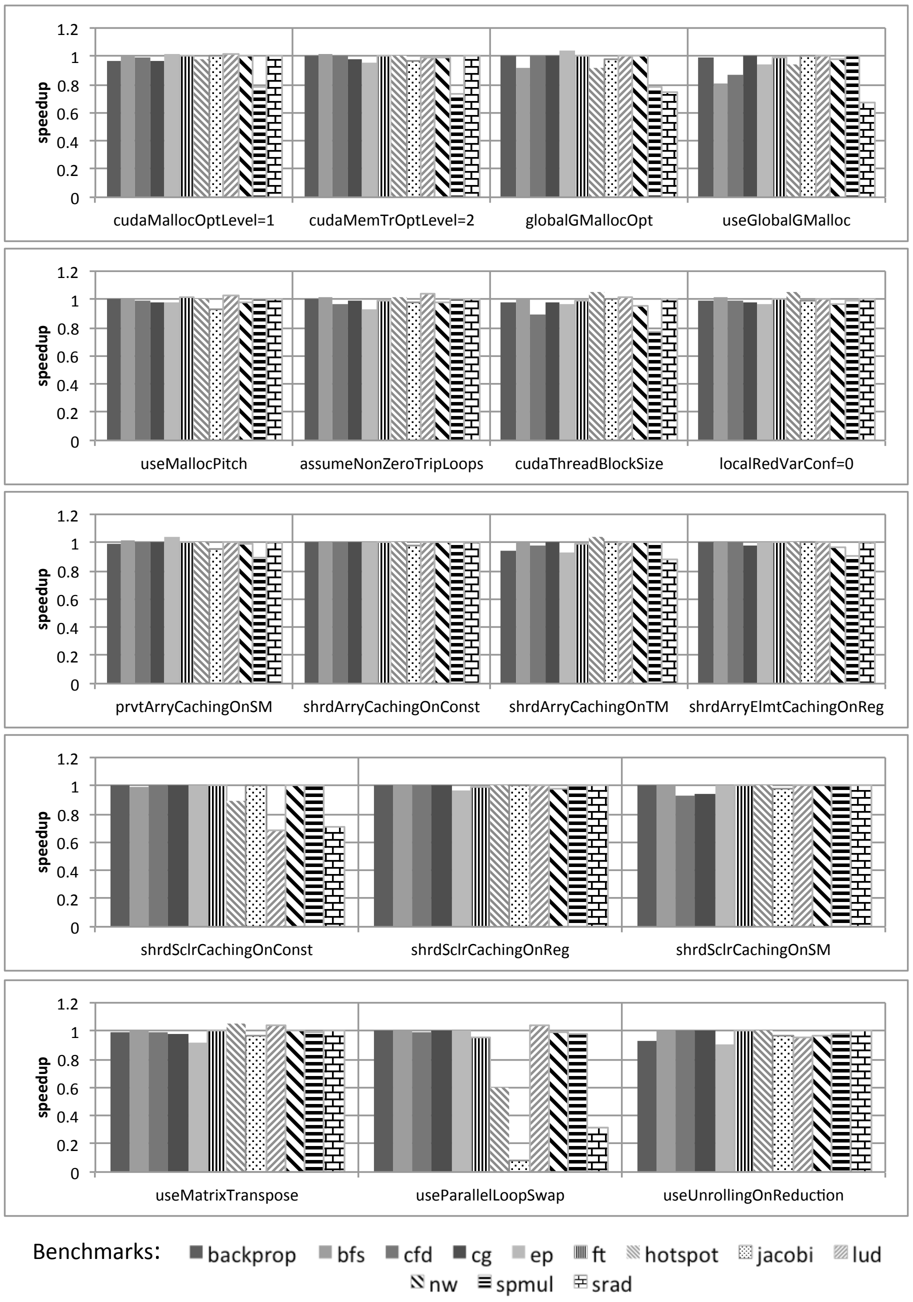


Figure 1.4.: Individual impacts of the 18 optimizations. Bars show normalized performance of the benchmarks after disabling the selected optimization. A large drop in performance indicates high impact.

Table 1.4.: Tuning Time Comparison of Pruned Exhaustive Vs. Modified IE Algorithm

Benchmark	Tuning Time (mins)	
	Pruned Exhaustive Tuning	Modified IE Tuning
SRAD	538	23
FT (S)	2345	23
CG (S)	1108	17
CFD (97k)	1083	210
FT (A)	3680	97
Jacobi (12288)	98	55

from the best tuning options set, so as to understand the effects of the individual optimization in terms of the slowdown incurred. The bigger the slowdown, the larger is the benefit of the optimization. We analyze the results in Fig 1.4 with respect to the techniques shown in Table 1.1.

Some benchmarks like SRAD, Jacobi, SPMUL depict high benefits obtained due to compiler techniques. However, some others like Backprop show relatively small effects. The effectiveness of our Modified IE tuning algorithm can be gauged from the observation that switching off an individual technique with respect to the best tuning optimization set has never improved the performance beyond 3%, which can be attributed to the computation variations.

Memory transfer optimization-based techniques show high impact on many GPU programs. Similarly, the techniques that change data access strides can be highly beneficial since they help coalesce memory accesses. *useParallelLoopSwap* and *useMatrixTranspose* are some such techniques.

Exploiting GPU specific memories for caching both the scalar and array variables can be highly beneficial. GPUs have on-chip cache and shared memories and off-chip constant and texture memories. The current OpenMPC setup tries to put all the variables (either scalar or arrays) on one of these memories, depending upon the tuning option provided. However, since these memories may not be large enough to hold the complete data sets, the compilation of such programs may fail (in which case the current tuning system ignores the option). We foresee a methodology to adaptively exploit all the GPU specific memories.

1.5 Chapter Takeaways

We have analyzed the performance of GPU optimization techniques present in the OpenMPC translation and tuning system. Our main findings indicate that the compiler engineer who wishes to translate a program in a given language into a CUDA program should consider the following optimizations:

1. Memory transfer optimization-based techniques are essential for offloading-based programming models.

2. Exploiting special memories on GPUs can yield significant speedups.

3. Transformations that change the memory access strides are of great importance in GPU programs.

4. Tuning is important. With its help, *standard* OpenMP programs can be translated effectively and efficiently into CUDA/GPU code.

5. Explicit GPU programming (without tuning support) needs to make use of CUDA-extensions (above items 1, 2) for best performance. It is important for emerging standards, such as OpenMP (4.0) [13] and OpenACC [47] to support these features. Above items 1 and 3 should be applicable to a wide range of accelerators. Item 2, however, is CUDA specific, but is necessary to obtain best performance.

We also proposed a new empirical tuning algorithm for GPU programs called Modified IE (MIE), which significantly reduces tuning time. MIE addresses and is able to tolerate runtime variations caused by memory transfer between GPU and CPU. As a result, MIE performs 11% better, on average, than the original OpenMPC tuning system [41], while maintaining polynomial tuning time.

One key learning from this chapter is that the cost of data transfers can be detrimental to the overall accelerator program performance. Pipelining is a well known technique to mitigate this cost, as the computation can be overlapped with the data transfers. We address this issue in the next chapter.

2. SCALING LARGE-DATA COMPUTATIONS ON MULTI-GPU ACCELERATORS

2.1 Introduction

Accelerators have become the forerunners of high-performance computing. Many supercomputers now use GPUs as accelerators. Among many open issues are those related to programming models and program optimization. This chapter addresses some such issues.

First, even though accelerators can be used as independent computational devices, they commonly serve as co-processors. Eligible computation is offloaded from the CPU - either explicitly by the programmer, or implicitly by the system software/hardware. Offloading involves data transfer from the CPU to the accelerator, which causes significant overhead – up to 95% of the program execution time, in our experiments. The primary contribution of this chapter is an automatic pipelining generation technique that reduces this overhead. To do so, the pipelining technique overlaps data transfer with computation. It creates opportunities for such overlap by transforming the computation into multiple chunks and transferring the data for chunk '(i+1)' while executing chunk 'i'. Our technique contrasts with those that reduce data transfer overhead by eliminating redundant memory transfers [7,48] and by advancing or delaying the data copy operations [49].

Second, the pipelining technique builds on an enabling technique that deals with another important issue in accelerators: The accelerator's memory space, which is discrete from the host CPU's, is limited in size; computation that fits in the CPU's memory may exceed the accelerator's capacity. The simple-most form of the technique splits computation into blocks that fit in memory. The same technique can be a key enabler for optimizations that tend to increase memory demand. Examples of such optimizations are data privatization, prefetching, and our pipelining technique. The difficulty in designing the technique is to model the increase in memory demand and determine the maximum chunk size that fits in the device memory. While most common accelerator benchmarks use data sizes that fit in memory, researchers have encountered the issue of limited memory sizes as well. For example, Liang [50] et al. discuss an example of an out-of-card FFT computation; a set of map-reduce based systems [29, 30] need to handle large data sizes that can exceed the GPU memory size. With the evolution of big-data systems, we expect the computing focus to move to large datasets. Our technique is the first to automatically tailor computation to the available memory space while considering optimizations that increase memory demand. The ability of splitting the computation also provides an opportunity to perform multi-device mapping of the program. Multiple GPUs attached to a single computation node are becoming an architectural reality, especially in supercomputing environments. Several techniques have been proposed [18, 19] to port custom applications to multi-GPUs. Our framework automates this process.

The third issue addressed in this chapter is the programmability of accelerators. An important issue is the design of a suitable high-level programming model, with one of the key questions being what architectural details need to be exposed to the programmer. Among many proposed models [1, 7–9, 51, 52], recently, the idea of using OpenMP extended with directives for accelerators, has gotten traction. We will integrate and evaluate our techniques in one of the most advanced compilers that has been pursuing this idea, OpenMPC [7]. In doing so, we introduce a novel component that addresses a fundamental problem in high-level programming environments for accelerators: The architectural complexity of the CPU-Accelerator systems exacerbates the difficulty of advanced compilers in making optimization decisions that need runtime information. These architectural intricacies are detrimental to the portability of the code. An optimally tuned program on one platform may perform poorly on another [15]. The most advanced compilers make use of *offline tuning* techniques to obtain the optimal choices of system-specific parameters. By contrast, we describe an adaptive runtime tuning mechanism that learns the architectural details in the initial phase of the program in a short time and determines the most suitable pipeline stage size that is used in the rest of the computation to attain best performance.

To summarize, in this chapter we make the following contributions:

- We design and implement an automatic pipelining technique that reduces CPU-accelerator data transfer overhead by overlapping the data transfers with computation. We will demonstrate the efficacy of this technique by displaying the performance benefits achieved on a set of benchmarks, including kernels and applications. On average, pipelining achieves a speed-up of 1.49x over the baseline OpenMPC codes.

- We design and implement an automatic computation splitting technique, *COSP*, that fits large computations into the accelerator's device memory. In doing so, it considers program transformations that increase memory demand, including our pipelining technique. We will show that large, out-of-card data sizes that can not be otherwise handled by OpenMPC/CUDA can be successfully run.

- We describe a low-overhead (less than 3% execution time) adaptive runtime tuning method that chooses a good split size for a problem and the underlying hardware so as to approach the best pipelining performance.

- We describe a system containing our novel techniques that automatically translates an input OpenMP code into a multi-device CUDA code that can employ multiple GPUs attached to the same host node. We evaluate the performance of benchmarks executed on multi-GPU systems.

The remainder of the chapter is organized as follows : Section 2.2 provides background information about GPGPU programming and OpenMPC. Section 2.3 analyzes the benefits of *COSP* and describes implementation details. Section 2.4 explains how pipelining and multi-GPU code generation are enabled by computation splitting and provides compiler design details for both. Section 2.5 describes the proposed adaptive runtime tuning system. Finally, Section 2.6 evaluates our system on a set of benchmarks from the StreamIt benchmark suite, CUDA SDK and Rodinia benchmarks.

2.2 Preliminaries

This section describes the GPU system architecture and the CUDA programming model. We also describe the OpenMPC compiler system that translates an input OpenMP program into CUDA code.

2.2.1 GPUs and CUDA

Typically, one or more GPUs are connected to a Host CPU via a PCIe bus. The CPU can offload computation kernels onto the GPU(s). Since the CPU and the GPU have different memories, input data must be copied from the CPU to the GPU before the start of a kernel. This operation is called *copy-in*. Similarly, copying the outputs of a kernel from the GPU to the CPU is termed *copy-out*. CPU-GPU data transfers are initiated and governed by the CPU.

GPUs are SIMD units with a large number of processors. NVIDIA GPUs have termed these processors *Streaming Multiprocessors* or *SMs*, each being an SIMD processor. CUDA [1] is a multi-threaded, SIMD programming model. The threads on the GPU device are divided into *ThreadBlocks*. Each *ThreadBlock* consists of a set of threads, each executing the same code. When the CPU launches a kernel, it prescribes the number of threads in a *ThreadBlock* along with the number of *ThreadBlocks* to launch. The set of *ThreadBlocks* launched by a kernel is called a *grid*.

CUDA has a complicated memory hierarchy. Each thread has its own local memory and a set of registers. Local memory contains a stack which is primarily used to spill the registers. A *ThreadBlock* has its on-chip local storage in the form of *shared memory*. The off-chip global memory is accessible to all threads in the grid. Further, on-chip *constant* and *texture* memories can act as read-only buffers.

Various factors impact the performance of a GPU kernel: coalesced memory accesses, register and shared memory usage, number of threads in a *ThreadBlock*, shared memory bank conflicts etc., making it difficult to predict the GPU performance [53–55]. To obtain good performance results, tuning is essential for GPU programs.

2.2.2 OpenMPC

OpenMPC [7] is a programming framework that synthesizes CUDA programs from OpenMP codes. The framework includes an extended OpenMP programming interface, a source-to-source translator [56], and an automatic compiler-assisted tuning system. The programming interface extends OpenMP with a new set of directives and environment variables for controlling CUDA specific parameters and optimizations. OpenMPC applies various code transformations and CUDA extensions to the input OpenMP code.

We chose OpenMPC as the underlying compiler for our system implementation because of the following reasons : (a) OpenMP-like programming models are becoming popular, especially with the advent of OpenACC [9]. The OpenMP standard itself is in the process of being extended to support accelerator devices [57]. OpenMPC is a research framework pursuing the same idea. (b) OpenMPC automatically performs many safe and beneficial code transformations so as to coalesce memory accesses and map data to different kinds of memories available on the GPU. OpenMPC also implements sophisticated CPU-GPU live variable analysis to remove or hoist the redundant memory transfers. (c) Considering the complexity of the CUDA programming model, its not always possible for a compiler to find the performance optimal CUDA parameters. OpenMPC provides OpenMP extensions for the programmer that can be used to set the CUDA related parameters. (d) The OpenMPC translator is realized on top of the Cetus [58] compiler, which provides an efficient implementation infrastructure.

2.3 COSP - An Enabler Technique

COSP, or *Computation Splitting*, divides a given problem into smaller, homogeneous subproblems. It acts as an enabler technique to other optimizations. In this section, we describe the nature of *COSP* and the optimizations it enables. While doing so, we establish an analytical upper bound on the size of a problem that can be run on an accelerator with limited memory. We provide a lower bound on the number of splits that the problem must undergo in order to fit in the device memory. We also describe the implementation of *COSP*.

2.3.1 Catering to Arbitrary Device Memory Sizes

COSP reduces the runtime device memory requirement of a problem; every subproblem requires less memory than the overall computation. We now analyze the factors that impact the device memory requirement. The total device memory requirement of a kernel depends upon the following:

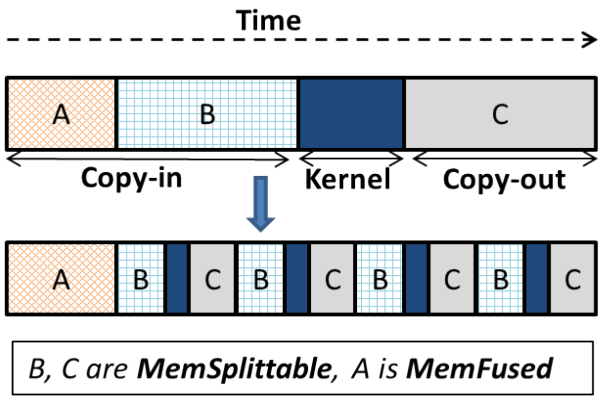


Figure 2.1.: *COSP* - *MemSplittable* data can be split since only a part of them is required per chunk of computation, *MemFused* is the part of the data accessed by every chunk.

- *COSP* converts a large problem into smaller subproblems. An example of such operation is shown in Fig. 2.1. In this example, data A, B, and C are a part of OpenMP *'shared'* data objects i.e. all threads work on a single copy of the data. In the original computation, data A and B are copied-in. The kernel uses these elements to generate data C, which is copied out of the device memory. The kernel, even after splitting, requires all data A for generating even a part of data C; we call A *MemFused* type of data. On the other hand, data B and C can be split, and each subproblem only requires a part of these elements. We call B and C *MemSplittable* data. If the splitting is perfect, amongst *numSplits* subproblems, the original runtime device memory space for the shared data goes down from ($MemSplittable + MemFused$) to ($MemSplittable/\ numSplits + MemFused$). However, a subproblem may also use the data of another (mostly neighboring) subproblem. We model this extra data requirement overhead by $SplitOverlap$.

- Every OpenMP *Private* data element in the input OpenMP program has to be allocated per thread in the computation. If the size of the private element is small (e.g. scalars), the element is generally stored in the device register. However, if the size of the private element is large (e.g. for arrays), the element needs to be placed in the local memory of the thread. Therefore, if P is the size of all such elements together and $NumThreads$ is the number of total threads, the device memory requirement is $P \times NumThreads$. Further, if the problem is split amongst $numSplits$ subproblems, the device memory requirement for the private data would be $P \times NumThreads/numSplits$.

- Memory prefetching is an important technique in accelerator programming. Device memory prefetching advances the copy-in operation for the ready data so as to overlap the copying time with the CPU computation. Similarly, for the data that is not immediately required by the CPU, the copy-out can be delayed while overlapping the copy-out time with the CPU computation. Both these optimizations require device memory to hold the buffered data. We denote this size as $PrefBuffer$.

To ensure that the device memory, $DevMem$, can fit all of the above components, the following constraint must be met:

$$\begin{aligned} DevMem \geq \Bigg(& P \times \frac{NumThreads}{numSplits} + PrefBuffer + \\ & MemFused + \frac{MemSplittable}{numSplits} + SplitOverlap \Bigg) \end{aligned}$$

$$\begin{aligned} & \Rightarrow numSplits \geq \\ & \left\lceil \frac{(MemSplittable + P \times NumThreads)}{(DevMem - PrefBuffer - MemFused - SplitOverlap)} \right\rceil \end{aligned} \tag{2.1}$$

Equation 2.1 provides a lower bound on the number of splits required on the input problem to make it fit in the device memory. Equation 2.1 also indicates that the benefits of *COSP* are multi-faceted. *COSP* can enable an accelerator program to run successfully even when the OpenMP *private* elements push the device memory requirements beyond available. *COSP* can also partition the computation into appropriate subproblems so as to enable prefetching in the device memory.

2.3.2 COSP Through a Code Example

Listing 2.1: Input Scalar Product OpenMP Code

```
#pragma omp parallel for shared(D, E, F) \\
private(vec, pos, sum)
for(vec = 0; vec < NUM_VECTORS; vec++) {
  sum = 0;
  for(pos = 0; pos < NUM_ELEMENTS; pos++) {
    sum += D[NUM_ELEMENTS * vec + pos ] *
    E[NUM_ELEMENTS * vec + pos];
  }
  F[vec] = (float)sum;
}
```

Section 2.3.1 provided a discussion on the lower bound of the number of splits required by the computation so as to make it fit in the device memory. We now describe a mechanism to achieve *COSP* through a compiler transformation. If all the parameters in Eq.2.1 are known, the compiler can choose the number of splits required to make the computation fit in the device memory. As we would show later in Section 2.4, *COSP* creates an opportunity for pipelining the subproblems. The number of subproblems required to obtain efficient pipelining can be less than the upper bound provided by Eq.2.1. Our compiler therefore abstracts out the number of splits as a variable.

OpenMP programs usually encapsulate parallelism using large *for* loops. We introduce the *COSP* mechanism through an example of *Scalar Product* code, as shown in Listing 2.1. This program generates scalar products for a set of vectors. The split version of the code generated by our compiler is shown in Listing 2.2. The input *parallel for* loop is split into many small loops. The upper bound for the inner *for* loop is set to be *SplitSize*. SplitSize is left as a parameter that can be set by the user or the automated tuning system. The inner loop body represents a subproblem of the initial large problem. We hereafter refer to the inner loop as a ***Split*** and the outer loop as ***Split Loop***.

Listing 2.2: Split Parallel Region for Scalar Product

```
for (split=0; split < NUM_VECTORS/SplitSize; split = split+1) {
  #pragma omp parallel for shared(D, E, F)
  private(vec, pos, sum) shared(split, SplitSize)
  for (vec = 0; vec < SplitSize; vec++ ) {
    sum=0;
    for (pos = 0; pos < NUM_ELEMENTS; pos++ ) {
      sum+=(D[(pos + (NUM_ELEMENTS *
      (vec + split*SplitSize)))]*
      E[(pos + (NUM_ELEMENTS * (vec + split*SplitSize))]);
    }
    F[(vec + split*SplitSize)] = (float)sum;
  }
}
```

For the *Scalar Product* program, the total number of splits created is equal to $NUM\_VECTORS/SplitSize$. This number needs to be larger than the *numSplits* calculated by Eq. 2.1 in Section 2.3.1 so as to meet the device memory size constraint. The number of splits in the problem is controlled by the SplitSize. Further, SplitSize governs the data size required by the kernel as well as the number of GPU threads synthesized by the OpenMPC system. Note that the *COSP* version in Listing 2.2 does not explicitly formulate the data partitions per Split. We defer the data partitioning analysis till the next section.

COSP can lead to extra data copy overheads in cases where the computation splitting is not perfect. As an example, consider stencil programs where computing each output requires an array element along with its neighbors. In such cases, *COSP* would require to account for the storage of boundary elements of each split, increasing the amount of overall data copied from the CPU to the GPU.

Since the subproblems provided by *COSP* are devoid of data dependences, they can be run in parallel. *COSP* can therefore generate opportunities for pipelining the slow host-device memory channel in an accelerator-based system. It also offers an easy- to-distribute program structure to the compiler system, which can then map subproblems to different devices.

2.4 Pipelining

Pipelining, in general, is a throughput-enhancing technique that overlaps the execution phases of one computation with another so as to make the best use of the available computational resources. We introduce an example of pipelining in Fig. 2.2. After performing *COSP*, one Split's kernel execution is overlapped with the next Split's memory copies; i.e. while the first subproblem kernel is working on the already copied-in data B, the copy-in for the next subproblem kernel's data B is taking

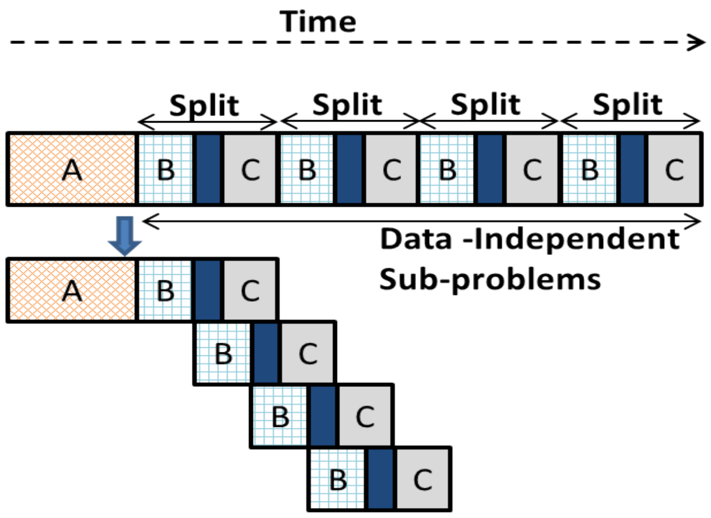


Figure 2.2.: Pipelining Opportunity Generated by *COSP* : Individual Splits are independent; they can be pipelined

place. It is important to note that to achieve successful pipelining, there should be two buffers present for each *MemSplittable* data so as to obtain the necessary prefetching.

In the case at hand, the three resources that we propose to pipeline include the CPU-GPU channel, GPU-CPU channel (if different than the first) and the GPU computational units. This section focuses on the pipelining code generation of the compute-split code. It also explains the strategy to perform multi-device code mapping and implementation mechanisms adopted by our compiler.

2.4.1 Achievable Speedup from Pipelining

Maximum attainable speedup from pipelining is restricted by the number of pipelining stages. In the proposed scheme, there are three pipeline stages : (i) Memory channel between the CPU and the GPU (ii) Memory channel between the GPU and the CPU (iii) GPU Computation cores. Hence, the pipelining speedup can be at most three. Most new GPUs support overlaps between the computation and transfers. Some advanced GPUs, such as Tesla M2090, also support overlaps in the memory copy operations in different directions, since they have different channels for copying in each direction. For GPUs without dedicated copy engines, the speedup would be restricted by two.

$$Speedup = \frac{t_{compute} + t_{MemFused} + t_{co} + t_{ci}}{t_{MemFused} + max(t_{compute}, t_{co}, t_{ci})} \tag{2.2}$$

where

$t_{compute}$ = Time spent in kernel computation

$t_{MemFused}$ = Time spent in transferring MemFused data

t_{ci} = Time spent for copy-in of MemSplittable data

t_{co} = Time spent for copy-out of MemSplittable data

Equation 2.2 provides an upper bound on the pipelining speedup, assuming the *COSP* overhead is zero. It also assumes the presence of different memory copy channels for copy-in and copy-out operations.

All outputs of a parallel program fall under the *MemSplittable* category unless they are reductions. Although *COSP* can always 'split' the outputs of a program, it may not be always able to do so for its inputs, if the inputs fall under the *MemFused* category. The worst case scenario would be a program wherein computation of a single output element requires input data of type *MemFused* that has a size larger than the device memory. In such a case, algorithmic change in the program structure is the only alternative.

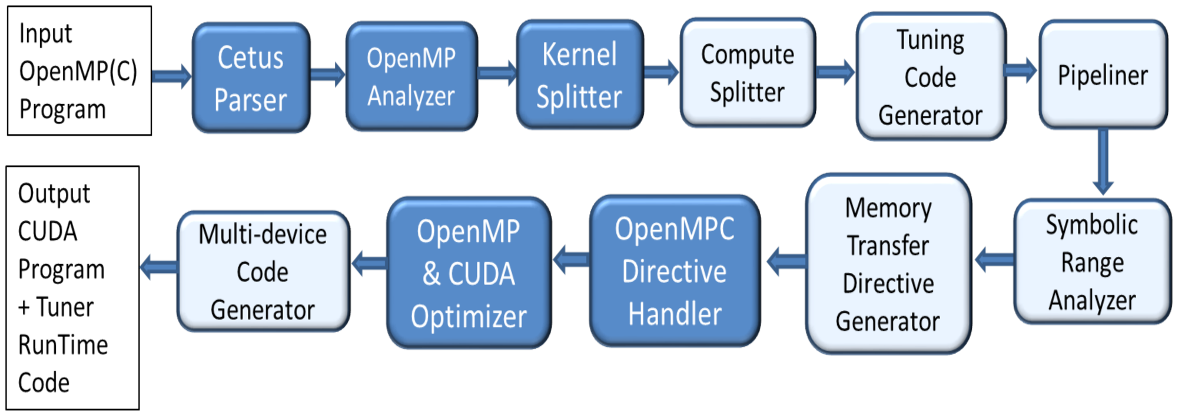


Figure 2.3.: Overall System Flow : Darker boxes indicate the base OpenMPC passes.

2.4.2 Compiler Organization

Figure 2.3 shows the structure of our compiler. It builds on OpenMPC, representing the program using Cetus IR [58]. After parsing the source code, OpenMPC system-internal decision making is performed to identify the kernels to be offloaded to the GPU. *COSP* is performed next on the kernel regions recognized by the OpenMP analyzer. For each eligible kernel region, tuning code is generated. The pipelining pass is applied next. Pipelining is optional in the sense that a user could simply generate only the compute-split code. Communication generation for each pipeline chunk is handled by the advanced symbolic range analysis [59] stage, which encodes its results as OpenMPC memory transfer pragmas. Once the OpenMPC directive handler and CUDA optimizer have finished their work on the IR, the multi-device code generation pass maps the pipelines to their respective devices. For the correct functioning of the multi-device code generation pass, the pipelining pass is made multi-device aware, in the sense that it keeps a map of the pipelines to devices, which is utilized later on by the multi-device code generation pass.

2.4.3 Generating Pipelined Code

At the heart of realizing the pipelining is the ability of the CPU to execute memory copy and kernel operations on the device asynchronously. The underlying CUDA model provides a mechanism called *'CUDA streams'* that realizes this asynchronous operation. A *'cudaStream'* [1] represents an instruction stream of computation on the host CPU that queues the launches of device commands i.e. kernel launches and memory transfers. All operations on a given *cudaStream* are launched sequentially, however, different *cudaStreams* can run independently of each other. Our pipelining implementation describes the strategy using *cudaStreams*.

The pipelining stage unrolls the Split Loop in Listing 2.2 twice. Listing 2.3 displays the intermediate code generated by our compiler. The unrolling factor of two corresponds to the two data buffers required; one for the currently executing Split and another for prefetching the inputs of the next Split. Each unrolled Split is transformed into a separate kernel and the required memory buffers are allocated for each kernel by the base OpenMPC system. Unrolling the Split Loop eases the software pipelining implementation.

Listing 2.3: Intermediate Representation for Pipelining the Compute Split Parallel Region

```
for (split=0; split<NUM_VECTORS/SplitSize; split=split+2) {
  //Determine starting points and ranges required per Split
  E_c2gstart_stream_0=((split*NUM_ELEMENTS)*SplitSize);
  E_c2grange_stream_0=(NUM_ELEMENTS*SplitSize);
  F_g2cstart_stream_0=(split*SplitSize);
  F_g2crange_stream_0=SplitSize;
  D_c2gstart_stream_0=((split*NUM_ELEMENTS)*SplitSize);
  D_c2grange_stream_0=(NUM_ELEMENTS*SplitSize);
  E_c2gstart_stream_1=((NUM_ELEMENTS*SplitSize)+
  ((split*NUM_ELEMENTS)*SplitSize));
  E_c2grange_stream_1=(NUM_ELEMENTS*SplitSize);
  F_g2cstart_stream_1=(SplitSize+
  (split*SplitSize));
  F_g2crange_stream_1=SplitSize;
  D_c2gstart_stream_1=((NUM_ELEMENTS*SplitSize)+
  ((split*NUM_ELEMENTS)*SplitSize));
  D_c2grange_stream_1=(NUM_ELEMENTS*SplitSize);

  //Code to be launched by cudaStream 0
  #pragma omp parallel for  private(pos, sum, vec) shared \
  (NUM_ELEMENTS, NUM_VECTORS, split, SplitSize, D, E, F)
  #pragma cuda gpurun \
  g2cmemtr(F[F_g2cstart_stream_0: F_g2crange_stream_0]) \
  c2gmemtr(D[D_c2gstart_stream_0: D_c2grange_stream_0]) \
  c2gmemtr(E[E_c2gstart_stream_0: E_c2grange_stream_0])
  #pragma cuda gpurun noc2gmemtr(F) \
  nog2cmemtr(NUM_ELEMENTS, NUM_VECTORS, D, E)
  for (vec=0; vec<SplitSize; vec ++ ) {
    sum=0;
    for (pos=0; pos<NUM_ELEMENTS; pos ++ ) {
      sum+=(D[(pos+(NUM_ELEMENTS*
      (vec + split*SplitSize)))]*
      E[(pos+(NUM_ELEMENTS*(vec + split*SplitSize)))]);
    }
    F[(vec + split*SplitSize)]=((float)sum);
  }

  //Code to be launched by cudaStream 1
  #pragma omp parallel for  private(pos, sum, vec) shared \
  (NUM_ELEMENTS, NUM_VECTORS, split, SplitSize, D, E, F)
  #pragma cuda gpurun \
  g2cmemtr(F[F_g2cstart_stream_1: F_g2crange_stream_1]) \
  c2gmemtr(D[D_c2gstart_stream_1: D_c2grange_stream_1]) \
  c2gmemtr(E[E_c2gstart_stream_1: E_c2grange_stream_1])
  #pragma cuda gpurun noc2gmemtr(F) \
  nog2cmemtr(NUM_ELEMENTS, NUM_VECTORS, D, E)
  for (vec=0; vec<SplitSize; vec ++ ) {
    sum=0;
    for (pos=0; pos<NUM_ELEMENTS; pos ++ ) {
      sum+=(D[(pos+(NUM_ELEMENTS*
      (vec + (split+1)*SplitSize)))]*
      E[(pos+(NUM_ELEMENTS*(vec + (split+1)*SplitSize)))]);
    }
    F[(vec + (split+1)*SplitSize)]=((float)sum);
  }
}
```

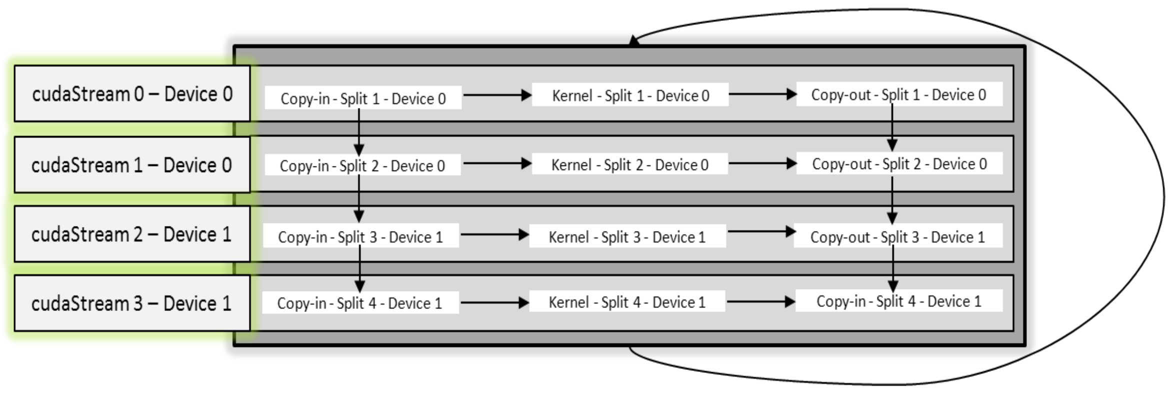

cudaStream 0 – Device 0
cudaStream 1 – Device 0
cudaStream 2 – Device 1
cudaStream 3 – Device 1
Copy-in - Split 1 - Device 0
Kernel - Split 1 - Device 0
Copy-out - Split 1 - Device 0
Copy-in - Split 2 - Device 0
Kernel - Split 2 - Device 0
Copy-out - Split 2 - Device 0
Copy-in - Split 3 - Device 1
Kernel - Split 3 - Device 1
Copy-out - Split 3 - Device 1
Copy-in - Split 4 - Device 1
Kernel - Split 4 - Device 1
Copy-in - Split 4 - Device 1

Figure 2.4.: System Strategy to generate and run programs on Multiple Devices - In this case, the number is 2. Straight arrows depict dependences, the curved arrow represents Split loop

Bounds for the data required by each Split need to be precisely calculated to avoid transferring more than necessary data (lines 8-13 in Algo. 2). We develop an aggregate data sections analysis using the advanced symbolic range analysis techniques offered in Cetus. This algorithm determines the starting locations and data lengths per Split required for all aggregate data elements.

It also categorizes the data into *MemSplittable* and *MemFused* categories. The algorithm determines the range of access for each usage of the given variable. It then performs a union operation on the individual ranges to get the comprehensive access range as well as the starting address of the data. Both these expressions are symbolic in nature and are parameterized by *SplitSize* and *Split* (Lines 3-17 in Listing 2.3).

The starting address and length for data copy are specified to the OpenMPC system using *c2gmemtr* and *g2cmemtr pragmas* that govern the generation of CPU-to-GPU and GPU-to-CPU communication, respectively. (Lines 23-25 and 42-44 in Listing 2.3). For *MemFused* data elements, for every Split, entire aggregate datatype transfer is required. In such cases, transfers for these data elements are hoisted outside the Split Loop.

Algorithm 2: Pipelined Code Generation from Compute Split Program

```
Input: Compute Split Loop ‘$Region$’
Output: Pipelined $CUDA$ code
1  Unroll $Region$ by a factor of 2;
2  Create $cudaStreams$ $stream0, stream1$;
3  $StreamMap$ = new Map($cudaStream$, $Split$);
4  $StreamMap$.insert($stream0$, first Split in $Region$);
5  $StreamMap$.insert($stream1$, second Split in $Region$);
6  foreach ($split \in Region$) do
       //  Static code contains only two Splits
7      foreach ($sharedVar \in split$) do
           //  sharedVar is OpenMP shared type
8          $range$ = rangeAnalysis($sharedVar$ , $split$);
9          if ($range$ contains $SplitSize$) then
10             $start$ = getStartingPoint($range$);
11             insertCopyPragmas($sharedVar$, $start$, $range$);
12         else
               //  this is a MemFused Variable, hoist memory
               //  transfers out of the Split Loop
13             insertCopyPragmas($sharedVar$, $Region$);
14 $cudaCode$ = translate($Region$, $StreamMap$);
   //  Format CUDA code to achieve desired
   //  queueing using streams
15 reOrganize($cudaCode$, $stream0$, $stream1$);
```

Since GPUs have at most one data copy engine in the CPU-to-GPU or GPU-to-CPU direction, it is necessary to schedule the operations on this channel wisely in order to avoid bottlenecks on the data copy engines. Correct scheduling is necessary to assure maximum overlapping benefits as well. The first Split in the unrolled Split Loop is assigned to *cudaStream 0*. That is, the kernel launches and memory transfers for this Split are handled by *cudaStream 0*. Similarly, the second Split is attached to *cudaStream 1*. In this manner, unrolling of the Split loop helps in (a) Creating private buffers on the device per *cudaStream* and (b) Providing an easy kernel-*cudaStream* mapping. Memory copy requests in a given direction from both *cudaStreams* get serialized. Hence, the corresponding queueing strategy performs copy-in from *cudaStream 0* for Split 1, then issues the kernel from *cudaStream 0* for Split 1 and simultaneously performs copy-in via *cudaStream 1* for Split 2. Next, the system launches the kernel from *cudaStream 1* and subsequently issues copy-out from *cudaStream 0* and *cudaStream 1* respectively. Listing 2.3 shows that alternate Splits go on different *cudaStreams*.

Our system maintains a mapping between a Split and its corresponding *cudaStream*. After the CUDA code is generated by OpenMPC, the compiler reorganizes the memory transfer and kernel launches so as to realize the queuing strategy. We portray the complete pipelining code generation algorithm in Algorithm 2.

2.4.4 Multi-GPU Code Generation

As seen in the previous section, to implement pipelining, two *cudaStreams* are required for a single GPU. For multi-device code generation, our system extends this strategy and assigns two *cudaStreams* per device, the overall number of *cudaStreams* used being twice the number of devices. In other words, the Split Loop is unrolled twice per device.

The next important stage in multi-device code generation is to perform work partitioning amongst devices. We use block-cyclic partitioning wherein two contiguous Splits are attached to the same device, since these Splits are more likely to access data elements in the vicinity of each other; this method improves spatial locality while performing CPU-GPU transfers.

Further, data elements that need to be present completely inside the device memory even for a single Split computation (*MemFused* elements) need to be made 'private' per device, and the memory copies for such elements need to be hoisted out of the Split Loop. An example with the overall multi-device code generation strategy for two devices is shown in Fig. 2.4. Note that the dependences are caused due to the queueing on the memory copy channels.

2.5 Adaptive Runtime Tuning System

SplitSize is the number of iterations of the parallel loop in a given Split. SplitSize can therefore be thought of as the size of the pipeline stage. Each iteration of the Split is mapped to a GPU thread by OpenMPC. Hence, the choice of SplitSize determines the number of $ThreadBlocks$ issued per Split, governing both the time required for computation and CPU-GPU communication. In this section, we describe how the choice of SplitSize impacts the pipelining performance. We then propose a heuristic tuning algorithm that selects the most suitable SplitSize.

2.5.1 Performance Variation with SplitSize

Equation 2.1 gave the maximum SplitSize that fits in the device memory. The SplitSize that yields the best performance results is usually less than this upper bound. Fig. 2.5 shows such execution time variation with different SplitSizes for two benchmarks. Experiments were run on Tesla M2090, which has separate memory copy engines in CPU-to-GPU and GPU-to-CPU directions. *Vector Add* is a memory copy-intensive application, whereas *Filterbank* is a compute-intensive one. *Vector Add* results were generated for problem size (iteration space) of 2^{27}; for *Filterbank*, the problem size was 2^{24}. Fig. 2.5 shows that the performance becomes better with increasing SplitSize for *Vector Add*, while better results can be achieved at lower SplitSize values for *Filterbank*. The choice of SplitSize is therefore important to achieve good performance, but selecting the correct SplitSize is not straightforward for the programmer. To build an intuition for choosing the SplitSize, we begin by analyzing the performance results on both memory copy-intensive and compute-intensive programs. The former spend most of their execution time on CPU-GPU transfers. The latter spend most of their execution time running the GPU kernels.

Fig 2.6 shows examples of the different behaviors of compute-intensive and memory copy-intensive programs. *Monte Carlo* is compute-intensive, while *Black Scholes* is memory copy-intensive. SplitSizes are chosen such that the number of *ThreadBlocks* launched by each SplitSize is a multiple of the number of SMs, *SMCount*, of the GPU. Note that the memory copy times show linear increase at smaller SplitSizes as compared to the kernel execution times. Since the pipelining benefits are higher when the pipelining stages are of equal size, relative increase in kernel execution time as compared to the memory copy time leads to better performance. Once the kernel execution starts to grow linearly with SplitSize, there is no more opportunity for higher benefits. This explains the performance variation with SplitSizes for *Vector Add* and *Black Scholes* benchmarks. Both these benchmarks have the highly memory copy-intensive *Type 1* overlapping pattern as shown in Fig. 2.7.

Compute-intensive programs like *Filterbank, Monte Carlo,* show the second type of overlap displayed in Fig. 2.7. In both these codes, the kernel execution time is larger

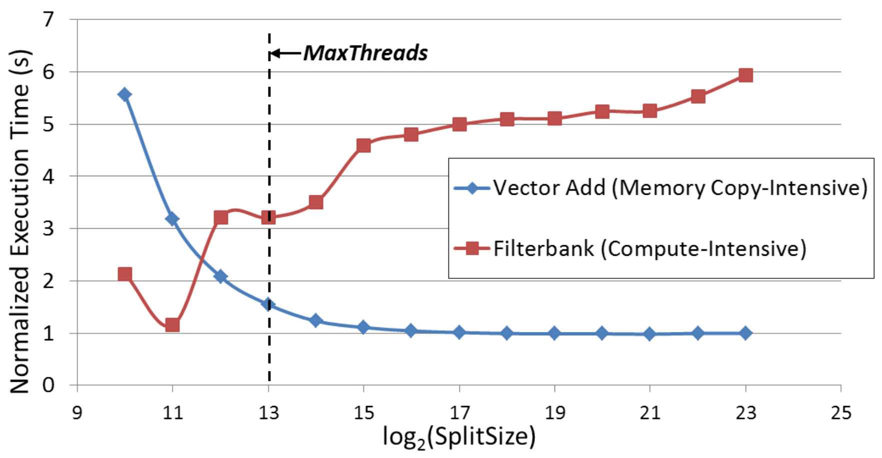


Figure 2.5.: Performance Variation with SplitSize : Performance is higher for smaller SplitSizes for compute-intensive benchmarks, while for memory copy-intensive ones, performance is higher for larger SplitSizes. *MaxThreads* is the maximum number of threads that can co-exist on the GPU.

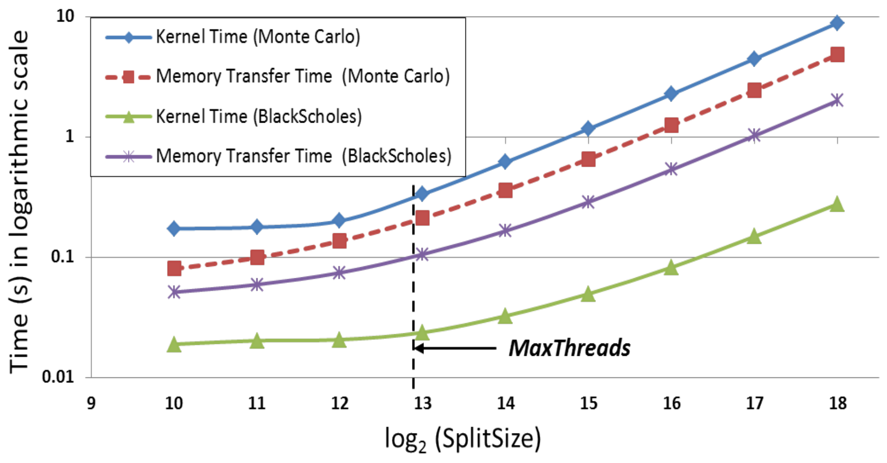


Figure 2.6.: Kernel execution and Memory copy times per Split for different SplitSizes. Monte Carlo is compute-intensive; Black Scholes is memory copy-intensive. Experiments were run on Tesla M2090.

than the sum of copy-in and copy-out times. A quick look at Fig. 2.7 shows that when the kernels are executing, the memory channel(s) are unused. One way to increase the kernel and memory copy overlap would be to run more *cudaStreams* so that the kernel time between the overlap of t_K^1 and t_K^2 (Fig. 2.7) can be further overlapped by memory transfers. Another alternative is to reduce the SplitSize and allow concurrent kernel

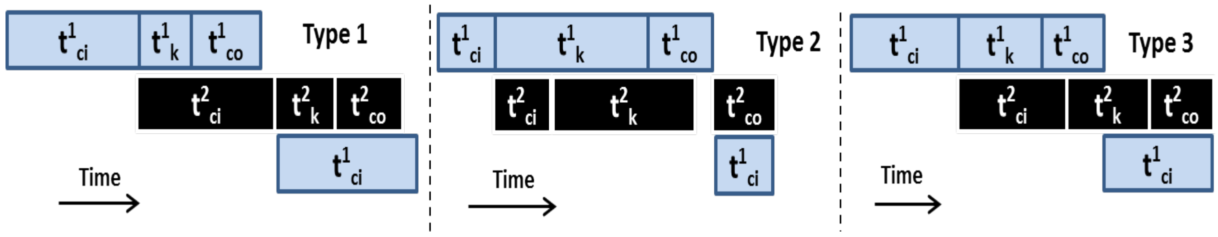


Figure 2.7.: Possible Overlap Types for GPUs with Distinct Copy-in and Copy-out Engines : t_{ci} is the copy-in time, t_{co} is the copy-out time and t_k is the kernel execution time. Number in the superscript represents the *cudaStream*. Type 1 is the highly memory copy-intensive type with $t_{ci} > t_{co} + t_k$. Note that t_{ci} and t_{co} are interchangeable. Type 2 is highly compute-intensive, with $t_k > t_{ci} + t_{co}$. Type 3 is neither ($t_{ci} > t_k > t_{co}$ & $t_k + t_{co} > t_{ci}$). Note that shown is one of the many cases of different program patterns that can fit in *Type 3*.

execution, supported by the newer GPUs. Kernel-kernel overlaps can easily occur in compute-intensive benchmarks. Due to the limited knowledge in GPU $ThreadBlock$ scheduling mechanisms in the presence of multiple kernels, kernel-kernel overlap performance is difficult to predict or model and the performance tuning would need explicit runs with different SplitSizes. However, increasing the SplitSize to be larger than the maximum number of threads that can coexist on a GPU ($MaxThreads$) would only lead to queueing up of $ThreadBlocks$ in the launching process. Hence, the SplitSize search space of interest in case of compute-intensive benchmarks is fairly small i.e. $SplitSize \leq MaxThreads$.

2.5.2 Adaptive Runtime Tuning Algorithm

With the aforementioned observations, we have developed a heuristic adaptive runtime tuning algorithm (Algorithm 4.5.2) for finding the SplitSize that would yield the best performance. First, the algorithm determines if the kernel is highly memory copy-intensive *(Type 1)* or highly compute-intensive *(Type 2)* or neither *(Type 3)* by running a pilot run with SplitSize equal to the $MaxThreads$.

Note that running just one Split (RUN_ONE_SPLIT, line 20 of Algo. 4.5.2) of the Split Loop is sufficient for the algorithm to determine the type of the kernel. The algorithm then generates a unique set of SplitSizes, named *CSet* or *configuration set*, depending upon the type of the program. Each SplitSize is chosen such that the number of $ThreadBlocks$ contained in it is a multiple of $SMCount$, so as to evenly distribute the $ThreadBlocks$ on SMs. The *CSet* generation is handled by the GENERATE_CONFIGS function (line 17 of Algo. 4.5.2). The largest SplitSize generated by the GENERATE_CONFIGS function is bounded by the device memory capacity. It is worthwhile to note that the algorithm is agnostic of the number of threads per $ThreadBlock$.

For *Type 1* programs, generated *CSet* contains SplitSizes $\geq MaxThreads$ in an increasing order. As explained earlier, a linear increase in the kernel execution time indicates the highest performance for memory copy-intensive programs. The algorithm therefore runs one Split per SplitSize in the *CSet* until the kernel time starts to grow linearly. Linear growth is established by comparing the kernel execution time of the previous SplitSize in the *CSet* (Line 23, in Algo. 4.5.2). This results in the selection of smallest SplitSize that would generate high performance while maintaining low device memory requirements.

For *Type 2* programs, algorithm generates the *CSet* with SplitSizes that are $\leq MaxThreads$. For each SplitSize, the runtimes for a single Split are noted, and the SplitSize with the best normalized overlapped time (Line 2, Algo. 4.5.2), t_o, is selected as the candidate.

Type 3 programs are neither highly compute-intensive nor memory copy-intensive. In such cases, the algorithm generates all possible SplitSizes (bounded by the device memory size) and runs one Split for each of them. The SplitSize with the best normalized overlapped time, t_o, is selected. *Type 3* denotes the worst case for the algorithm,

Algorithm 3: Heuristic SplitSize Tuning Algorithm

```
Input: $T$ is the Number of Threads per ThreadBlock
Input: $SMCount$ is the Number of SMs
Input: $B$ is the maximum ThreadBlocks per $SM$
Output: Best SplitSize configuration
// $t_{ci}$ is copy-in time, $t_k$ is kernel execution time
// $t_{co}$ is copy-out time, $t_o$ is the overlapped time
// $t_o = min[max((t_{ci} + t_k), t_{co}), max((t_{co} + t_k), t_{ci})]$
// timer is an array of structures storing the
// quadruplet $t_{ci}$, $t_{co}$, $t_k$, $t_o$
// $Normalizer(i) = MaxThreads/CSet(i)$
1  Function tuner()
2      $MaxThreads \leftarrow T \times SMCount \times B$ ;
3      $Type \leftarrow$ Nil ;
4      $(t_{ci}, t_k, t_{co}, t_o) \leftarrow RUN\_ONE\_SPLIT(MaxThreads)$ ;
5      if $(t_{ci} + t_{co} \leq t_k)$ then
6          $Type \leftarrow 2$;
7      else if $(t_{ci} + t_k \leq t_{co} \vee t_{co} + t_k \leq t_{ci})$ then
8          $Type \leftarrow 1$;
9      else
10         $Type \leftarrow 3$;
       // GENERATE_CONFIGS creates a sorted set of
       // SplitSize configurations
11     $CSet \leftarrow GENERATE\_CONFIGS(Type)$;
12     $min \leftarrow 1$;
13     for $(i = 1 \rightarrow length(CSet))$ do
14         $timer[i] = RUN\_ONE\_SPLIT(CSet[i])$;
15         if $(Type = 1)$ then
16             if $(i > 1)$ then
17                 $diff \leftarrow ((timer[i].t_k \times Normalizer(i)) - (timer[i-1].t_k \times$
                   $Normalizer(i-1))$;
18                 if $(|diff| \approx 0)$ then
19                     return $CSet[i-1]$;
20         else if
           $(timer[min].t_o \times Normalizer(min) < timer[i].t_o \times Normalizer(i))$
           then
21             $min = i$;
22     if $(Type = 1)$ then
23         return $CSet[i-1]$;
24     else
25         return $CSet[min]$;

26 Function RUN_ONE_SPLIT(SplitSize)
27     Run one Split from the Split loop;
28     Return corresponding $t_{co}, t_{ci}, t_k, t_o$;
```

since it requires a traversal in a larger space. However, most of the programs that we tried fell into either *Type 1* or *Type 2*.

Since the algorithm needs to run only a single Split to learn about the characteristics of a given SplitSize, the runtime overhead incurred is low.

2.5.3 Compiler Support for Tuning

The adaptive runtime tuning algorithm is automated in the compiler. The compiler automatically generates a tuning function for each kernel region. The tuning function contains an outlined copy of the parallel region, which is used to realize RUN_ONE_SPLIT function. This outlining is performed at the OpenMP level. Since the tuning function involves execution of some extra Splits, the original data may get modified. Outlining prevents this potentially harmful behavior to the kernel data during the tuning execution, as it forces the data elements in the tuning function to be allocated separately. Timer calls are inserted in this outlined copy to gather t_k, t_{ci} and t_{co}. Certain parts of the algorithm are inserted in this function, while others, such as GENERATE_CONFIGS, are implemented in a run-time library. The compiler automatically inserts the necessary calls to the runtime library functions. The tuning function is invoked only during the first call to the kernel; subsequent kernel calls use the SplitSize value that was generated during the first run.

Notice that the tuning system may not represent the best SplitSize if the control flow of the parallel region is divergent. However, if the parallel region is invoked only once, the first invoke runs the tuner which indeed finds the correct SplitSize.

2.6 Evaluation

This section evaluates the performance of the presented computation splitting, pipelining and multi-device code generation regimes. We study seven kernels and two applications. *DCT*, *FFT* and *Filterbank* are traditional streaming benchmarks from the StreamIt benchmark suite [60]. These are compute-intensive applications. Other kernels, such as *Scalar Product*, *Black Scholes* and *Vector Add*, are from the CUDA SDK [61] and are mostly memory copy-intensive. *Monte Carlo*, also from CUDA SDK, is compute-intensive. *SRAD* and *CFD* are two applications from the Rodinia benchmark suite [62] and both are memory copy-intensive. To explore out-of-card situations, the benchmarks were run with larger datasets than provided in the benchmark suites. The baseline, non-pipelined translation from OpenMPC is used as the comparison point.

We used an NVIDIA Tesla M2090 GPU for our experiments. The device has 16 Streaming Multiprocessors (SMs) and remarkably large 6GB of DRAM. The GPU is connected via an x16 PCIe link to a host system consisting of an AMD Opteron Processor 6282 with 16 cores, running at 2.6 GHz. The host system has 64GB RAM. Up to 4 GPUs were connected to the host using the same PCIe bus.

We evaluate the contributions of our system in the following manner : (a) We demonstrate the ability of our system to handle large out-of-card data sizes by performing *COSP*. We compare the scalability of our approach against hand-written CUDA and baseline OpenMPC programs. (b) We evaluate the efficacy of our tuning method by comparing its performance to a naive compiler-only strategy. We also measure the overheads incurred by the tuning system. (c) To evaluate the benefits of pipelining and multi-GPU code generation, we compare the results obtained by these techniques over baseline OpenMPC.

Benchmark	DataSize (Iteration Space)	CUDA Time (s)	OpenMPC Base Time (s)	%Copy-in Time(s)	%Copy-out Time(s)	%Kernel Time(s)	OpenMPC Pipelined Time (s)	Speedup Ideal	Speedup Achieved
Scalar Product	1024 x 1024	0.633	0.88807	52.19789	0.22676	47.57535	0.46297	1.91579	1.91822
	1024 x1024 x2	1.267	1.7723				0.91496		1.93703
	1024 x1024 x4	----	----				1.73067		
Monte Carlo	1024 x32	0.00537	0.00454	21.55109	13.71958	64.72933	0.0037	1.54489	1.22733
	1024 x1024 x32	***	1.79902				1.24699		1.44268
	1024 x1024 x64	***	3.5924				2.51465		1.42859
	1024 x1024 x128	----	----				5.02565		
Black Scholes	1024 x16	0.00105	0.00158	46.64777	41.9736	11.37863	0.00164	2.14373	0.96344
	1024 x1024 x128	1.8598	1.22591				0.87698		1.39786
	1024 x1024 x256	----	2.457				1.73985		1.41219
	1024 x1024 x384	----	----				2.60183		

Table 2.1.: Scalability of the *COSP* and Pipelining Mechanism : We compare the execution times of hand-written CUDA, baseline OpenMPC and compute split, pipelined OpenMPC programs. The ideal speedup is calculated using Eq. 2.2. In the table, '***' represent failure of the code due to larger-than-allowed grid sizes used. '—' represent code failure due to out-of-memory data size errors. Scalability of our approach can be gauged as arbitrarily large problems with out-of-card data sizes can be run and the speedup achieved for any large data size remains almost constant.

2.6.1 System Scalability

COSP allows large, out-of-card data sizes to run on GPUs with limited device memories. Table 2.1 shows the system scalability of the *COSP* approach for three different representative benchmarks and also compares the results with hand-written CUDA and baseline OpenMPC codes. We calculate the maximum achieveable speedup from pipelining and compare it with the achieved speedup. To generate these results, we used hand-written CUDA codes from the CUDA SDK. Asynchronous transfers between the CPU and GPU require the corresponding memory to be allocated in 'pinned' pages i.e. the OS pages that can not be swapped out of the host memory. This can be achieved using the CUDA *cudaHostAlloc* API call. Since OS pages allocated in this fashion improve the overall application performance, the hand-written and base OpenMPC-generated CUDA codes were modified, allocating the host memories using *cudaHostAlloc*, to produce consistent comparison results with the CUDA versions generated by our system.

The scalability problems faced by the hand-written CUDA programs and the base OpenMPC-produced programs can be clearly seen from Table 2.1, since whenever the data size goes out-of-card, both these codes fail. For hand-written CUDA programs, as seen for the *Monte Carlo* benchmark, the number of *ThreadBlocks*, or the grid size, launched by the code crosses the CUDA-imposed limit of 65536 in a single dimension and the code starts failing even when the memory space requirement is sufficiently small to fit in the device. This is a severe programmability issue, as the programmer must consider all possible input sizes and manage the grid formation accordingly. A high-level programming model, such as OpenMPC, can easily tackle this issue by launching two dimensional grids if one dimension exceeds the limit. Hand-written CUDA codes underperform the baseline OpenMPC-generated codes as the input sizes grow large for the *Black Scholes* and *Monte Carlo* benchmarks. In the *Black Scholes* hand-written CUDA code, the launched grid size is constant and is better suited for small data sizes. In the *Monte Carlo* hand-written CUDA code, a part of the parallel loop is left out from the kernel and is instead run on the CPU. Our system translates this entire loop for GPU execution. Further, the *constant* memory allocation in the hand-written *Monte Carlo* CUDA code is proportional to the data size, and *constant* memory can run out of space if the problem size increases. High-level programming models can alleviate these programmability issues, ensuring the scalability of programs for arbitrary data sizes.

Performance benefits of our pipelining scheme can be seen in Table 2.1. This table also dispays the effectiveness of our implementation by comparing the ideal speedup that can be achieved from pipelining against the one obtained by our system. Except for the smallest data sizes, pipelining speedup numbers closely follow the ideal speedups for the *Monte Carlo* and *Scalar Product* programs. The differences between the ideal and achieved speedups are mainly due to the underperformance of the bidirectional transfer overlaps on the PCI Express (PCIe). This effect becomes a performance limiter for programs like *Black Scholes*, which have large bidirectional data transfers. For the small dataset of *Monte Carlo*, the optimal SplitSize suggested by our tuning system was 8192, being just one fourth of the input iteration space, thereby lowering the obtained pipelining benefits. Similar is the case for *Black Scholes*. Constant speedups for any large data size demonstrate the scalability of our approach.

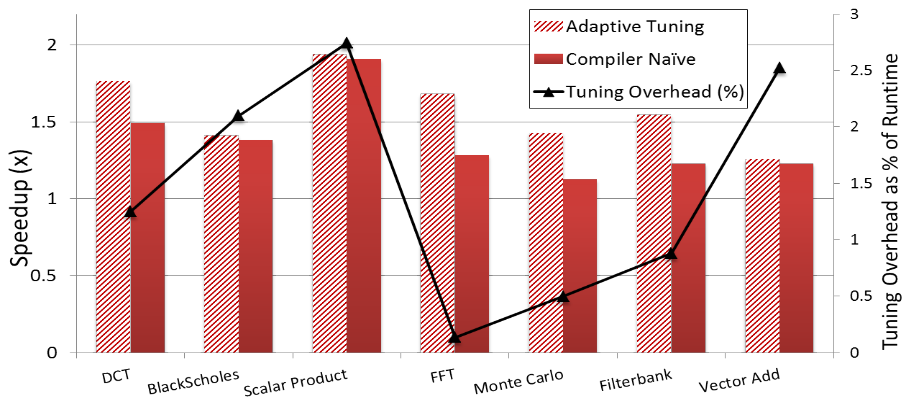


Figure 2.8.: Performance of Adaptive Runtime Tuning System : Speedups are with respect to the OpenMPC non-pipelined baseline. Higher performance of the tuned program versions over a naive pipelining approach emphasize the necessity of tuning.

2.6.2 Tuning System Performance

To measure the performance benefits gained from tuning the pipelined system, we compare the speedups achieved by our adaptive tuning method over the baseline OpenMPC codes against a *naive* compiler-driven splitting strategy. We found 1024 splits to be a good number that generated performance improvements and chose it as the “naive” reference point.

Fig. 2.8 displays the effectiveness of the adaptive runtime tuning system over this naive strategy. The superior performance of the tuned codes over the naive strategy indicates the importance of tuning; a static estimate of the number of splits can not provide the best performance. Further, a number of splits that yields good results for a given program may be suboptimal for another program. We also measure the overheads incurred by the adaptive runtime tuning system in terms of the percentage runtime spent in tuning. Because the tuning system runs only a single Split of the Split Loop per tuning configuration, the runtime overhead is low. The maximum overhead, measured as the percentage runtime of the total execution time, is less than 3%. Note that the tuning overhead decreases as the computation size grows.

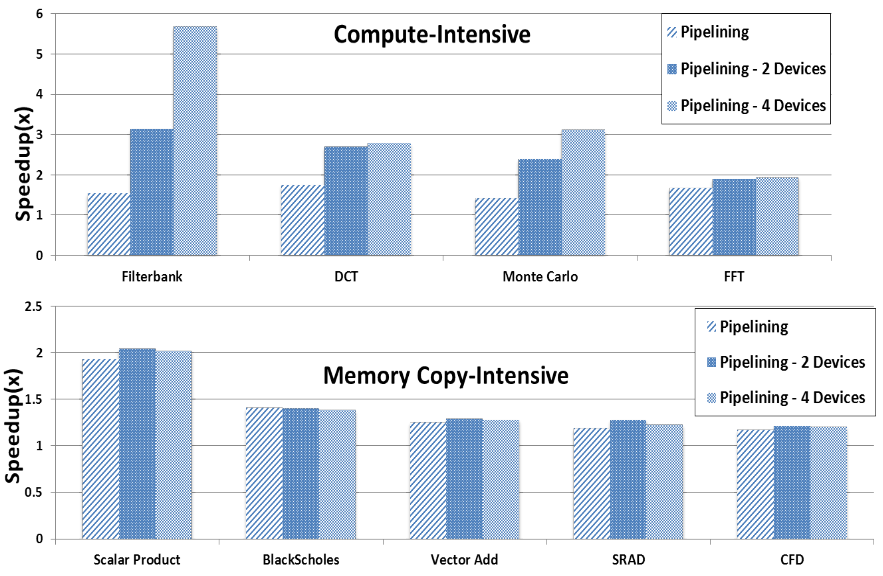


Figure 2.9.: Speedup over the Baseline OpenMPC Generated Codes (without pipelining) : Compute-intensive applications show good scalability with multiple GPUs. Memory copy-intensive programs scale poorly with multiple GPUs since PCIe bus forms a bottleneck.

2.6.3 Overall Performance Comparison

We now present comprehensive results over all benchmarks showing the effects of pipelining on 1, 2 and 4 GPUs in Fig. 2.9. Since the maximum speedup over the baseline system that can be achieved with pipelining is limited by three in our setup, the theoritical maximum speedup that can be achieved by pipelining, implemented on four GPUs, can be at most twelve.

Benchmarks like *DCT, FFT, Scalar Product, Monte Carlo, Filterbank* show large performance benefits since they have balanced computation and memory transfer contents, translating into equally sized pipeline stages. *SRAD, CFD and Vector Add* have a low computation-to-communication ratio and therefore show lesser overall speedups. Highly compute-intensive benchmarks like *Filterbank, Monte Carlo* show excellent scalability when multiple devices are used.

Secondly, not all benchmarks show large performance benefits when run with multiple GPUs; the reason being the bottleneck formed on the PCIe bus while transferring the data.

2.7 Related Work

An alternative to deal with problems of arbitrarily large data sizes can be to write the basic program flow in terms of a stream graph. Each kernel can then be made to have a granularity of a single iteration of the stream graph. Two recent systems propose mechanisms to convert streaming programs written in StreamIt [63] language into GPU codes. The first system, Sponge [24], suggests mechanisms to optimize the stream graph. It then splits the graph into multiple kernels. Huynh et al. propose another system [23] that takes a different approach and places an entire iteration of a stream graph on a single Streaming Multiprocessor (SM). This mechanism can face scalability issues due to the restriction of running just one iteration on an SM. Since each filter in a StreamIt graph consists of its own inputs and outputs, global memory accesses can be overwhelming. Both these systems therefore propose mechanisms to prefetch data into the GPU shared memory and optimize the shared memory usage. However, the behavior of these systems is undefined if the data size required is larger than the GPU memory. Secondly, the applicability of the stream programming model to large applications is still an open challenge. Further work by Huynh et al. [64] performs multi-level partitioning of the stream graph to overcome the scalability challenges in [23]. In this work, a mechanism to port StreamIt programs to multi-GPUs on the same system has also been proposed. This system heterogeneously executes kernels on several GPUs, owing to the outcome of the multi-level partitioning performed on the StreamIt graph. By contrast, our system partitions the work homogeneously amongst GPUs and yields the best possible pipelining code with the help of a fast tuning system.

A naive CUDA-provided solution to deal with out-of-card data sizes is to make use of *Zero Copy* memory. The *Zero Copy* memory mechanism allocates the corresponding buffers in the host memory instead of the device memory. The pointers to the buffers on the CPU and the GPU are mapped to each other. GPU reads the memory objects directly from the host during the kernel execution. Therefore, if the kernel has significant amount of reuse, the overheads of copying from the CPU memory would degrade the performance.

Pipelining benefits have been previously explored for GPUs to hide the global memory access latencies by generating a software pipeline that copies data into shared memory [21–24]. Aji et al. propose an approach [65] that deals with a network of nodes containing GPUs, aiming to overlap the communication between nodes with computations. Our work complements these techniques since our goal is to efficiently pipeline the CPU-GPU bus.

GPU performance tuning has been a research challenge; many approaches try to model and tune the GPU performance. While many proposed tuning systems are specific to the problem domains (3D Stencil [66], N-body simulations [67], 3-D FFT [68], SpMV [69]) , Ryoo et al. [54] propose generic performance metrics that help pre-select some potential parameter configurations to choose from. A distinguishing factor of our work is the tuning objective: In contrast to approaches that try to optimize the size of the *ThreadBlock*, our work attempts to optimize the grid size i.e. the number of *ThreadBlocks*. OpenMPC's inherent tuning system performs compiler-driven optimization option pruning to generate a set of parameter configurations. However, both the OpenMPC tuning system and the space pruning proposed by Ryoo et al. are run offline and require multiple complete runs of the entire programs to choose the best configuration.

Partitioning the application so as to make use of multiple GPUs attached to the same host CPU is an emerging research challenge. While some approaches target specific applications, e.g. Matrix multiply [18], fluid simulations [19], Kim et al. [20] provide an OpenCL based approach that provides a single device image of a multi-GPU system to the application developer. However, the scalability of this approach is bound by the total GPU memory sizes.

By contrast, ours is the first work that automatically deals with out-of-card data sizes and generates pipelined, multi-GPU code for generic applications, starting with a high-level program representation.

2.8 Chapter Takeaways

In this chapter, we have described a novel pipelining optimization that is able to scale large-data computations on multi-GPU accelerators. To this end, the technique splits a computation so it fits in the available memory resources and overlaps data transfer with computation. The execution can take advantage of multiple GPU devices. We have demonstrated that the technique can successfully execute out-of-card datasets that would fail without our optimization. For programs and datasets whose baseline versions succeed, our technique improves performance by 1.49x, on average, owing to the computation-communication overlap. We have implemented our optimization in one of the most advanced compilation platforms for GPGPUs: OpenMPC. This system converts OpenMP programs to CUDA, performing several advanced transformations. OpenMPC provided a state-of-the-art baseline for our measurements; it also allowed us to demonstrate our technique in the context of an important current language trend, which is the extension of OpenMP with accelerator directives.

This chapter dealt with multi-GPU systems, wherein multiple GPUs are attached to the same host node. However, many supercomputing environments contain a cluster of nodes, where each node has one or more GPUs. To exploit GPUs across a cluster, distributed programming models are necessary. Next chapter describes a MapReduce programming system that can automatically employ both CPUs and GPUs in the cluster.

3. HETERODOOP : A MAPREDUCE PROGRAMMING SYSTEM FOR ACCELERATOR CLUSTERS

3.1 Introduction

A growing number of commercial and science applications in both classical and new fields process very large data volumes. Dealing with such volumes requires parallel processing in cluster environments, and often on systems that offer high compute power.

For this type of parallel processing, the MapReduce paradigm has found popularity. The key insight of MapReduce is that many processing problems can be structured into one or a sequence of phases, where a first step (Map) operates in fully parallel mode on the input data; a second step (Reduce) combines the resulting data in some manner, often by applying a form of reduction operation. MapReduce programming models allow the user to specify these map and reduce steps as distinct functions; the system then provides the workflow infrastructure, feeding input data to the map, reorganizing the map results, and then feeding them to the appropriate reduce functions, finally generating the output.

The large data volumes involved may not fit on a single compute or storage node. Thus, distributed architectures with many nodes may be needed. Among the systems that support MapReduce on distributed architectures, Hadoop [25] has gained wide use. Hadoop provides a framework that executes MapReduce problems in a distributed and replicated storage organization (the Hadoop Distributed File System – HDFS). In doing so, it also deals with node failures.

Big-data problems pose high demands on processing and IO speeds, often with emphasis on one of the two. For general compute-intensive problems, accelerators, such as NVIDIA GPUs and Intel Xeon Phis, have proven their ability for an increasing range of applications. To obtain high performance, their architectures make different chip real-estate tradeoffs between processing cores and memory than in CPUs. A larger number of simpler cores provide higher aggregate processing power and reduce energy consumption, offering better performance/watt ratios than CPUs. In GPUs, intra-chip memory bandwidth is high and multi-threading reduces the effective memory access latency. These optimizations come at the cost of an intricate memory hierarchy, reduced memory size, data accesses that are highly optimized for inter-thread contiguous (a.k.a. coalesced) reference patterns and explicitly parallel programming models. Using these architectures therefore requires high programmer expertise.

While accelerators can perform well on compute-intensive applications, IO-intensive MapReduce problems may not always benefit. Previous research efforts on MapReduce-like systems employ either GPUs [29–31] alone, disregarding IO-intensive applications, or CPUs [25–28] alone, leaving out the GPU acceleration. In this chapter we present our *HeteroDoop*[1] system, which exploits *both* CPUs and GPUs in a cluster, as needed by the application. HetereoDoop makes four specific contributions, addressing the following challenges.

The first challenge in developing such a heterogeneous MapReduce system is the programming method. In a naive scheme, the programmer would have to write two program versions, one for CPUs and the second for GPUs. This need arises as accelerators rely on explicitly parallel programs, be it either low-level programming models such as CUDA [1] and OpenCL [6], or high-level ones, such as OpenACC [9] and

[1]https://bitbucket.org/asabne/heterodoop/wiki/Home

OpenMP 4.0 [13]. Although available high-level programming models relieve the user from having to learn model-specific APIs, such as in CUDA or OpenCL, they still require explicit parallel programming. On the other hand, in CPU-oriented MapReduce systems, programmers write only sequential code; the underlying framework automatically employs all cores in the cluster by concurrently processing the input data, which is split into separate files. Previous research on GPU uses for MapReduce has either relied on explicitly parallel codes with accelerator-specific optimizations [30, 31, 33, 70], and/or on specific MapReduce APIs [29, 30, 33, 34]. Programmability in both approaches is poor; the former requires learning low-level APIs and the latter necessitates application rewriting. To overcome these limitations, our contribution enables programmers to port already available sequential MapReduce programs to heterogeneous systems by annotating the code with *HeteroDoop directives*. Inserting such directives is straightforward, requires no additional accelerator optimizations, and leads to a single input source code for both CPUs and GPUs. Furthermore, the resulting code is portable; it can still execute on CPU-only clusters.

The second key challenge in exploiting accelerators is their limited, non-virtual memory space. The default parallelization scheme used in MapReduce/Hadoop engages multiple cores by processing separate input files in parallel – typically one per core, as an individual task. Data input is appropriately partitioned into separate *fileSplits*, which are fed to the different compute nodes and their threads. Simultaneous accesses by many threads to their fileSplits require a large memory. This size requirement is not a problem in today's typical CPUs with 4 to 48 cores and virtual memory support; however, in GPUs with several hundred cores and possibly thousands of threads, the available, non-virtual memory is insufficient. Our second contribution addresses this challenge by processing the data records *within* a fileSplit in parallel on accelerators, while retaining the default processing scheme on the CPU.

The third challenge is to translate the annotated, sequential source code in a way that obtains high accelerator performance. Our contribution is a MapReduce domain-specific optimizing source-to-source translator, assisted by a runtime system, that generates CUDA code and splits the work across CPUs and GPUs. Several issues arise in this process. The first one is load imbalance across GPU threads, as the records processed in parallel may be of different size. A global work-stealing approach would incur high overheads, due to excessive atomic accesses by the GPU threads. HeteroDoop overcomes this issue by using a novel record-stealing approach that partitions the records statically across GPU threadblocks but dynamically within threadblocks. Another issue is that GPU memory is statically allocated but the size of the map phase output, the key-value pairs, is not known a priori. Over-allocation would lead to inefficient sorting of the key-value pairs, which follows the map phase. To resolve this issue, HeteroDoop includes a fast runtime compaction scheme, resulting in efficient sort. Furthermore, the runtime system executes the sort operation on the GPU rather than on the slower CPU. Other optimizations include efficient data placement in the complex GPU memory hierarchy and automatic generation of vector load/store operations.

A final challenge in developing such a heterogeneous MapReduce scheme is to cater to the different processing speeds of CPUs and GPUs while performing work partitioning. Although prior work [71,72] has dealt with load balancing across nodes, intra-node heterogeneity has remained an issue. HeteroDoop's *tail scheduling* scheme addresses this issue. Our contribution is based on a key observation: the load imbalance only arises in the execution of the final tasks in a job; careful GPU-speedup-based scheduling of the tailing tasks can avoid this imbalance.

We have evaluated the HeteroDoop framework on eight applications, comprising well-known MapReduce programs as well as scientific applications. We demonstrate the utility of HeteroDoop on a 48-node, single GPU cluster with large datasets, and on a 64-node, 3-GPU cluster with in-memory datasets. Our main results indicate that the use of even a single GPU per node can speed up the end-to-end job execution by up to 2.78x, with a geometric mean of 1.6x, as compared to CPU-only Hadoop, running on a cluster with 20-core CPUs. Furthermore, the execution time scales with the number of GPUs used per node.

The remainder of the chapter is organized as follows: Section 3.2 provides background on GPUs, MapReduce, and Hadoop. Section 3.3 describes the HeteroDoop constructs, followed by the compiler design in Section 3.4. Section 3.5 describes the overall execution flow of the HeteroDoop framework and details the runtime system. The tail scheduling scheme is explained in Section 3.6. Section 3.7 presents the experimental evaluation. Section 3.8 discusses related work, and Section 3.9 presents the takeaways from the chapter.

3.2 Preliminaries

We introduce the basic terminology used in this paper for the GPU/CUDA architecture and programming model as well as the MapReduce and Hadoop concepts. We keep the discussion of GPUs and CUDA brief, assuming reader familiarity, but we do refer to introductory material [73].

3.2.1 CUDA Architecture and Programming

In the CUDA [1] GPGPU architecture, the many GPU cores are structured into multiple Streaming Multiprocessors (SM). CUDA threads execute in SIMD fashion, where a *warp* consisting of 32 threads executes a single instruction and multiple warps time-share an SM in multi-threading mode. Storage is separate from the CPU address space; it is structured into the global (or device) memory, the per-SM shared memory (user managed cache), and specialized memories. The latter include the read-only constant and texture memory as well as the registers.

CUDA programming is explicitly parallel with user-driven *offloading*; i.e, the user identifies compute-intensive code sections, *kernels*, which are to be run on the GPU (the device) and inserts data transfers between CPU and GPU. The threads are organized into threadblocks. All threads within a threadblock execute on the same SM. The programmer manages most of the storage hierarchy explicitly, including fitting the data into the limited-size memories. There is no virtual memory support.

3.2.2 MapReduce and Hadoop

In the MapReduce model, programmers write a *map* and a *reduce* function, with the system organizing the overall execution workflow. The input data is placed on a distributed file system, such as the HDFS [74] (Hadoop Distributed File System). HDFS stores the input in blocks, or *fileSplits*. A job consists of a set of map and reduce tasks. In Hadoop, one node usually acts as master and the others as slaves. The master node runs a *JobTracker*, while each slave runs a *TaskTracker* – together they orchestrate the necessary map and reduce tasks in a way that exploits data locality, engages all available nodes and cores, and provides fault tolerance. The total number of concurrent tasks in a Hadoop cluster is typically the same as the number of available cores.

In Hadoop, each map task processes one fileSplit. The map function applies a map operation to each data record in this fileSplit. The map task emits a set of <key, value> pairs (or *KV pairs*). The framework puts these KV pairs into different partitions, each partition targeted at a particular reduce task. To form these partitions, the KV pairs are sorted by their keys and split by a default or user-provided partitioning function.

Each partition is then sent to its target reduce task. This task typically resides on a different node, making this a costly step. To reduce the cost, a task-local, user-provided reduction operation, known as the *combiner*, is applied first on each partition, minimizing the size of the communicated data. This communication is also known as the *shuffle phase*.

In the next phase, the *sort phase*, each reduce task merges the incoming partitions into a sorted list. Then the *reduce phase* applies the reduce function to these data. The output is written back to the HDFS. As the data volumes involved are typically very large, the intermediate results between map and reduce are typically written to the local disk.

Hadoop uses a *heartbeat* mechanism for communication between the JobTracker and TaskTracker. TaskTrackers send heartbeats to the JobTracker at regular intervals. These heartbeats include items such as status of tasks and free cores or *slots*. If the JobTracker finds that a TaskTracker has free slots, it schedules new tasks on the particular TaskTracker in the heartbeat response.

Hadoop is written in Java, and therefore the baseline system supports map, combine and reduce functions written in Java. Hadoop Streaming [75] is an extension that supports other languages for writing MapReduce programs. It is implemented so that the map, combine, and reduce functions obtain their input from standard input and write the output onto standard output; map, combine, and reduce can be written as unix-style "filter" functions, using a language of choice.

3.3 HeteroDoop Directives

Recall that the HeteroDoop directives are designed to exploit both CPUs and GPUs in a cluster with a single source program. They identify the *map* and *combine* functions and their attributes in a serial, CPU-only MapReduce program. From this information, our translator generates code that exploits both the CPUs and GPUs available in the nodes of a distributed architecture.

While the concept of HeteroDoop directives is language-independent, our HeteroDoop prototype supports programs written in C. Our implementation makes use of the Hadoop Streaming framework, which we extend to enable efficient execution on both CPUs and GPUs.

3.3.1 HeteroDoop Directives with an Example

A key insight used in the design of HeteroDoop constructs is the observation that both map and combine functions iterate over a non-predetermined number of records. The bulk of the computation of the map and combine functions is performed inside a *while* loop. HeteroDoop directives identify these loops and express attributes that allow the translator to generate efficient parallel code for the GPUs.

Listing 3.1 shows an example *map* code, written in C, for *Wordcount*. This application counts the occurrences of each word in a set of input files. The code reads each input line and applies to it an elementary map operation. For each word in the line, this map operation emits a KV pair $<$ word, 1>. Notice that the input is read from STDIN using the *getline* function, while the output is written to STDOUT via *printf*.

In general, the elementary map operation is applied to every record. By default a record is a line of input. The bulk of the map computation lies within the loop

Listing 3.1: Wordcount Map Code with HeteroDoop Directives

```
 1 int main() {
 2   char word[30], *line;
 3   size_t nbytes = 10000;
 4   int read, linePtr, offset, one;
 5   line = (char*) malloc(nbytes*sizeof(char));
 6   #pragma mapreduce mapper key(word) value(one) \\
 7   keylength(30) kvpairs(20)
 8     while( (read = getline(&line, &nbytes, stdin)) != -1) {
 9     linePtr = 0;
10     offset = 0;
11     one = 1;
12     while( (linePtr = getWord(line, offset, word,
13         read, 30)) != -1) {
14       printf("%s\t%d\n", word, one);
15       offset += linePtr;
16     }
17   }
18   free(line);
19   return 0;
20 }
```

iterating over these records – lines 8–17 in the example. The *mapreduce* directive on line 6 with the *mapper* clause tells the compiler that this loop applies the map operation. The *key* and *value* clauses identify the variables used for emitting KV pairs. The *keylength* and *vallength* clauses indicate the lengths of the respective variables; the clauses are needed if these variables do not have a compiler-derivable type. Table 3.1 lists all HeteroDoop directives and clauses. Some of these clauses are optional; users need to provide them only for further optimizations and tuning of the generated GPU code.

Listing 3.2 shows an example *combine* code for the same *Wordcount* application. Recall that, before the combiner is run, the underlying HeteroDoop system sorts the KV pairs emitted by the map according to their keys and places them into different

Listing 3.2: Wordcount Combine Code with HeteroDoop Directives

```
int main() {
  char word[30], prevWord[30];
  prevWord[0] = '\0';
  int count, val, read; count = 0;
  #pragma mapreduce combiner key(prevWord) value(count)
  keyin(word) valuein(val) keylength(30) vallength(1)
  firstprivate(prevWord, count) {
    while( (read = scanf("%s %d", word, &val)) == 2 ) {
      if(strcmp(word, prevWord) == 0 ) {
        count += val;
      } else {
        if(prevWord[0] != '\0')
          printf("%s\t%d\n", prevWord, count);
        strcpy(prevWord, word);
        count = val;
      }
    }
    if(prevWord[0] != '\0')
      printf("%s\t%d\n", prevWord, count);
  }
  return 0;
}
```

partitions. The combiner function operates on each partition and sums up the occurrences of each word. Note that the *while* loop in the map function can execute in parallel, but the one in the combine function cannot, except for reduction-style parallelism. The only readily available parallelism in the combine and reduce executions exists across partitions. Typically, the number of partitions is not high enough to exploit the GPU completely. For this reason, HeteroDoop provides no directives for reduce functions and executes them on the CPUs only. For combine functions, however, as the data is already present in the GPU memory, HeteroDoop employs an economical way for GPU execution (Section 3.4.2).

Two extra clauses are necessary on the combiner function : *keyin* specifies the variable that receives the map-generated key and *valuein* does the same for the map-generated value.

3.3.2 Clauses for Memory and Thread Attributes

HeteroDoop directives also allow for clauses that improve performance by specifying the attributes of the data and threads. The following constructs exist:

Read-only variables:

The *sharedRO* clause lists read-only variables. The compiler places such variables in faster GPU memories, such as the constant memory and the texture memory. By default, our compiler places read-only scalars in the constant memory. Arrays whose sizes are known at compile time are placed in the texture memory, otherwise in the global memory. The *texture* clause forces placement in the texture memory. Placing data in the texture memory is especially useful for random array accesses, as this memory comes with a separate on-chip cache.

Firstprivate:

This clause specifies variables that can be privatized during the map or combine operation but are initialized beforehand. In the absence of this clause, the compiler tries to identify such variables automatically. It issues a warning if the analysis is inaccurate, e.g., due to aliasing.

Notice that in the MapReduce programming model, all written variables are privatizable during the map and combine operations. There is no shared written data. The translator performs the privatization without the need for user directives.

Table 3.1.: HeteroDoop Directives

Clause	**Arguments**	**Description**	**Optional**
mapper		Specifies that the attached region performs map operation	No
combiner		Specifies that the attached region performs combine operation	No
key	Variable name	Specifies the variable that contains the key	No
value	Variable name	Specifies the variable that contains the value	No
keyin	Variable name	Specifies the variable that contains the incoming key. Valid only on the combiner	No
valuein	Variable name	Specifies the variable that contains the incoming value. Valid only on the combiner	No
keylength	Integer variable	Specifies the length of the key being emitted	No
vallength	Integer variable	Specifies the length of the value being emitted	No
firstprivate	A set of variable names	These variables are initialized before being used in the attached region	No
sharedRO	A set of variable names	These are read-only inside the map or combine regions	Yes
texture	A set of variable names	These variables are read-only and hence can be placed in GPU texture memory	Yes
kvpairs	Integer variable	This is an optional clause on map region specifying the maximum KV pairs that can be emitted from a single record	Yes
blocks	Integer variable	Specifies the number of threadblocks to use	Yes
threads	Integer variable	Specifies the number of threads in a threadblock	Yes

Space Allocation for KV Pairs:

The space needed for the KV pairs output by map is not known at translation time. The translator allocates all free GPU memory for storing these KV pairs. In practice, this leads to an over-allocation. To reduce the memory required for storing these KV pairs, the users can indicate the maximum number of KV pairs emitted by each record using the *kvpairs* clause. This reduction in the storage space required for the KV pairs improves the aggregation efficiency for this storage, as will be described in Section 3.4.3.

Thread Attributes:

The *blocks* and *threads* clauses allow the programmer to choose the number of threadblocks and number of threads in a threadblock on the GPU, respectively. These clauses help tune the map and combine kernel performance.

3.4 Compiler

This section describes the HeteroDoop source-to-source translator, which converts an input MapReduce code annotated with HeteroDoop directives into a CUDA program. It is built using the Cetus [58] compiler infrastructure. Translating directly into CUDA, rather than a higher-level model such as OpenACC or OpenMP 4.0, provides direct access to GPU-specific features. For the code running on the CPUs, the same Hadoop Streaming code is compiled using the *gcc* backend compiler. In this manner, a single MapReduce source is sufficient for the execution on both CPUs and GPUs. The HeteroDoop compiler carries out various MapReduce-specific optimizations. The next two subsections describe the key translation steps of generating GPU kernel code for the map and the combine functions, respectively. Section 3.4.3 also describes the code generation for the host CPU, which offloads the kernels.

Algorithm 4: Handling Variables in Generated Kernels

```
Input: $region$ - Region attached to the annotation
Input: $newGPUKernel$ - Extracted region copy (kernel)
Input: $sharedROSet$ - Set of shared read-only variables inside
       $newGPUKernel$
Input: $textureSet$ - Set of shared read-only variables to be placed on texture
       memory
Input: $firstPrivateSet$ - Set of firstprivate variables
Output: Correct placement of variables in $newGPUKernel$, along with
        necessary GPU memory allocation and data transfer generation
1  Function handleVariables($region$, $newGPUKernel$, $sharedROSet$,
   $textureSet$, $firstPrivateSet$)
2      $usedVars$ = $newGPUKernel$.getUsedVars();
3      foreach $var \in usedVars$ do
4          if ($sharedROSet$ contains $var$) then
5              if ($var$ is scalar) then
                   //  //gets placed on constant memory
6                  $newVar$ = addParameter($var, newGPUKernel$);
7              else
                   //  //a shared read-only array
8                  $newVar$ = addParameter($var$, $newGPUKernel$);
9                  insertMallocAndCpyIn($region$, $newVar$, $var$);
10         else if ($textureSet$ contains $var$) then
11             $tex$ = createNewTexture($var$);
12             $newVar$ = addParameter($var$, $newGPUKernel$);
               //  cudaBindTexture
13             bindTexture($tex$, $newVar$, $region$);
               //  inserts cudaMalloc and cudaMemCpy
14             insertMallocAndCpyIn($region$, $newVar$, $var$);
15         else
               //  a private variable
16             $newVar$ = addPrivateVar($newGPUKernel$, $var$);
17             if ($firstPrivateSet$ contains $var$) then
18                 newFPCopy = addParameter($var$, $newGPUKernel$);
19                 if ($var$ is array) then
20                     insertMallocAndCpyIn($region$, $newFPCopy$, $var$);
21                 insertInKernelCopyCode($newFPCopy$, $newVar$);
```

3.4.1 Map Kernel Generation

Each thread in the map kernel fetches a record, performs the elementary map operation, and stores the generated KV pairs into its portion of a central GPU storage, called the *global KV store*. The process repeats until all records are done.

The compiler begins by locating the annotation with the *mapper* clause. The annotated *while* loop is the target region for kernel generation. The compiler extracts this region into a new function, *newGPUKernel*, which contains the kernel code. A key step in doing this is the transformation of the different variable types. Algo. 4 shows the procedure. From the user annotations, the compiler generates *sharedROSet*, which lists the shared read-only variables. *TextureSet* consists of read-only array variables that are to be placed in the texture memory. *FirstPrivateSet* contains all variables that are firstprivate, either specified by the programmer or identified by the compiler. The scalar *sharedRO* variables are passed as arguments to the kernel; the underlying CUDA compiler places these arguments in *constant* memory (lines 5–6). A pointer to each array *sharedRO* variable is passed through a kernel parameter (lines 8–9); storage is allocated on the GPU and the array data is copied from the CPU into this space. The transformation for using the texture memory is essentially the same as for *sharedRO* arrays (lines 11–15). The functions *addParameter* and *addPrivateVar* create new variables and automatically rename the previous variable names in the *newGPUKernel* respectively.

The remaining variables inside the *newGPUKernel* are private (lines 17–24). For each private variable, a new variable is created inside the kernel body using the *addPrivateVar* function. For firstprivate variables, there is an extra step. For scalars, the initial value is passed through a kernel parameter. For arrays, storage is allocated and a reference is passed via kernel parameter. The initial values of the array elements are copied into the corresponding memory locations before the kernel. Inside the kernel body, each thread copies these values into the private spaces using the *insertInKernelCopyCode* function.

Listing 3.3 shows the translated CUDA map kernel for *Wordcount* (listing 3.1). The kernel generation procedure adds internal parameters for bookkeeping. *Ip* represents the input file buffer; *ipSize* is the input file size. *RecordLocator* is a data structure that keeps a list of starting addresses of all input records. *DevKey* and *devVal* are the GPU variables for keys and values of the global KV store, respectively. *StoresPerThread* holds the storage size allocated to each thread in the global KV store. *DevKvCount* array keeps a count of the KV pairs emitted by each thread.

The first GPU-specific translation step is the insertion of a *mapSetup* function call (line 9) for the map execution. This function sets up internal variables for GPU execution e.g. *tid*, which stores the thread ID. Next, the algorithm replaces the CPU record input function, such as *getline*, with a GPU equivalent *getRecord* function (line 11). Similarly, the KV-emitting function, which is *printf* in the CPU code, is replaced with a GPU equivalent *emitKV* function (line 18). The *emitKV* function puts the generated KV pairs into the global KV store. The translation scheme replaces the calls to all C standard library functions with GPU counterparts. Since the current GPUs do not support all C standard library calls, their equivalent implementations are provided by the runtime system. The runtime system also includes other functions used in the translated code, such as *mapSetup*, *getRecord*, and *emitKV*. At the end of the map kernel execution, the *indexArray* holds the locations of individual KV pairs in the global KV store. This array is useful in indirectly accessing the global KV store.

The *mapFinish* function (line 25) performs bookkeeping after a thread is done with the map execution. Most importantly, it keeps a count of the total number of KV pairs emitted by each thread.

Listing 3.3: Translated Wordcount Mapper Code

```
__global__ void gpu_mapper(char *ip, int ipSize,
int *recordLocator, char * devKey, int * devVal,
int storesPerThread, int * devKvCount,  int keyLength,
int valLength, int * indexArray, int numReducers) {
  char gpu_word[30];
  int gpu_read, gpu_one, gpu_offset, gpu_linePtr;
  int index, tid, start;
  __shared__ unsigned int recordIndex;
  mapSetup(&start, &tid, &index, ipSize, storesPerThread,
  ip, devKvCount, numReducers, &recordIndex);
  while( ( gpu_read = getRecord(ip, &recordIndex, &start,
        recordLocator) )!= - 1) {
    gpu_linePtr = 0;
    gpu_offset = 0;
    gpu_one = 1;
    while( (gpu_linePtr = getWord(ip + start, gpu_offset,
          gpu_word, gpu_read, 30)) != -1) {
      emitKV(gpu_word, &gpu_one, devKey, devVal, &index,
      devKvCount, keyLength, valLength, numReducers,
      storesPerThread, indexArray);
      gpu_offset += gpu_linePtr;
    }
  }
  mapFinish(index, storesPerThread, devKey, keyLength,
  indexArray, numReducers, devKvCount);
}
```

Record Stealing:

The execution time taken by the map operation for each record can vary greatly among different records in certain MapReduce applications due to the differences in the amount of data in each record. For example, in the *kmeans* application, where each record contains a list of movie ratings, some records have fewer reviews than others. This leads to load imbalance among threads if the records are statically partitioned. A better strategy is therefore to perform dynamic record distribution, or *record stealing*. However, a global record distribution scheme would require global atomic functions on the GPU, which are expensive. We therefore devise a scheme where the records are statically and equally split among the threadblocks used in the map kernel, and threads of a given threadblock steal a new record from the threadblock's pool. As it is common for larger records to get distributed across threadblocks, record stealing implemented at the threadblock level is effective. The variable *recordIndex* (line 9, listing 3.3) acts as a counter for the records used by the threads of a threadblock. This variable is placed in the GPU shared memory for fast atomic increment operations. The maximum record stealing that a thread can perform is limited by the *storesPerThread* it has in the global KV store.

Using Vector Data Types:

For array keys/values, the generated code uses an optimization of internally using CUDA-specific vector data types, such as *char4*, which increase the memory accessing performance. The functions that exploit such a vectorization include *emitKV*, and string functions called within the map code, e.g. *strcpy*.

3.4.2 Combine Kernel Generation

As the combine function operates on one partition at a time, there is no explicit parallelism. However, different partitions can be processed in parallel. Unfortunately, we have found that the number of partitions i.e., the number of reducers can be low in certain MapReduce applications. Therefore, the degree of exploitable parallelism can be low, leading to underutilization of the GPU. Our scheme therefore exploits in-partition, reduction-style parallelism, while sacrificing full functional equivalence with respect to the CPU code. The idea for this approach is simple: e.g. for the *Wordcount* code, a particular partition received the following KV pairs <a,1>, <a,1>, <a, 1>,

<b,1>. The output from a CPU combiner would be <a,3>, <b,1>. However, if two different threads were to operate on the partition, with the first operating on the first two KV pairs, and the second on the last two, the intermediate output would be <a,2>, <a,1>, <b,1>. In this manner, the functional equivalence of the combiner is traded off for parallelism. Due to the presence of the global reducers, however, this trade-off is legal; the global reducer will eventually produce the same output. In practice, as the number of KV pairs in each partition is typically high, this small dissimilarity has a negligible impact on the communication volume.

A second design choice in the combine kernel generation deals with the fact that the degree of exploitable in-partition parallelism is still usually much less than the number of GPU threads. The compiler-generated code forces all threads in a warp to execute the same combine function redundantly. This way, intra-warp thread divergence is eliminated. Listing 3.4 shows the generated kernel for the *Wordcount* combine code (listing 3.2). The compiler inserted a number of parameters: *Keys* and *values* hold the KV pairs emitted by the map for the particular reducer. *OpKey* and *opVal* store the combine-emitted KV pairs. The lengths of keys and values for both map and combine functions are passed as parameters. The *handleVariables* pass (algo. 4) adds two parameters *prevWordFP* and *countFP* for firstprivate variables.

Similar to the map setup function, the *combineSetup* function (line 11) initializes the internal variables of the combiner operation. The compiler performs an optimization of placing the private array variables in the faster shared memory for each warp. In this example, *gpu_prevWord* and *gpu_word* are placed in the shared memory. The scalars are placed in the thread registers. The compiler replaces calls to *scanf* that read in the KV pairs with a GPU-specific *getKV* call (line 18). The *keyin* and *valuein* clauses are necessary for this function. The original *printf* function for outputting the KV pairs is replaced by the *storeKV* function (lines 24, 34). *FinalCount* keeps track

Listing 3.4: Translated Wordcount Combiner Code

```
__global__ void gpu_combiner(char *keys, int *values,
char *opKey, int *opVal, int *indexArray, int *finalCount,
int size, int mapKeyLength, int mapValLength,
int combKeyLength, int combValLength,
char *prevWordFP, char *countFP) {
  int laneID, kvsPerThread, warpID, ptr, gpu_val;
  int high, kvCount, index, gpu_read, gpu_count;
  //WARPS_IN_TB = number of warps in a threadblock
  __shared__ char gpu_prevWord[WARPS_IN_TB][30];
  __shared__ char gpu_word[WARPS_IN_TB][30];
  combineSetup(kvsPerThread, &laneID, &warpID, &ptr,
    &high, &kvCount, &index, size);
  for(int i=0; i < 30; i++) { //init firstprivate data
    gpu_prevWord[warpID][i] = prevWordFP[i];
  }
  gpu_count = countFP;
  while(getKV(gpu_word, keys, &gpu_val, values, ptr,
   high, indexArray, mapKeyLength, mapValLength)!= -1) {
    if(strcmpGPU(gpu_word, gpu_prevWord[warpID],
        mapKeyLength)==0){
      gpu_count += gpu_val;
    } else {
      if(gpu_prevWord[0] != '\0') {
       storeKV(gpu_prevWord[warpID], &gpu_count, &index,
       combKeyLength, combValLength, opKey, opVal,
       &kvCount);
      }
      strcpyGPU(gpu_prevWord[warpID], gpu_word,
        mapKeyLength);
      gpu_count=gpu_val;
    }
  }
  if (gpu_prevWord[0] != '\0') {
   storeKV(gpu_prevWord[warpID], &gpu_count, &index,
   combKeyLength, combValLength, opKey, opVal, &kvCount);
  }
  finalCount[warpID]=kvCount;
}
```

of the KV pairs emitted by each thread so that the final output can be combined and written back to the CPU.

While each thread in the warp executes the combiner function redundantly, for *getKV* and *storeKV* the threads perform vectorized loading and storing of KV pairs, respectively. This method loads/stores one element in an array key/value from one thread. It improves performance as it enables coalesced memory accesses. This same optimization is also performed on string functions, such as *strcpy* in the combiner kernels. Note that in order to dynamically switch between the vector and non-vector modes, all threads in the warp must be active for the entire code execution, making redundant execution in non-vectorizable code sections necessary. This redundant execution comes without any side-effects, owing to the warp-level SIMD model. If neither the key nor the value is an array, the compiler would generate a code where only a single thread per warp is active.

3.4.3 Host Code for Offloading

Since the GPU code execution is orchestrated by the host, the compiler must generate the necessary host code. Fig. 3.1 displays a flowchart for the structure of the generated code. First, the code copies the input fileSplit from HDFS into the GPU memory. A GPU kernel is then launched to collect and count the records in the input. Next, necessary storage is allocated on the GPU for map and combine kernels. To allocate the global KV store, all available GPU memory is used in the absence of the *kvpairs* clause. Otherwise, the global KV store memory allocation can be reduced, because the total number of records is already known. Each GPU thread in the map kernel generates KV pairs in its own portion of the global KV store. Note that each thread may not completely use its own portion of the global KV store, leading to *whitespaces*, which are the empty slots in the global KV store.

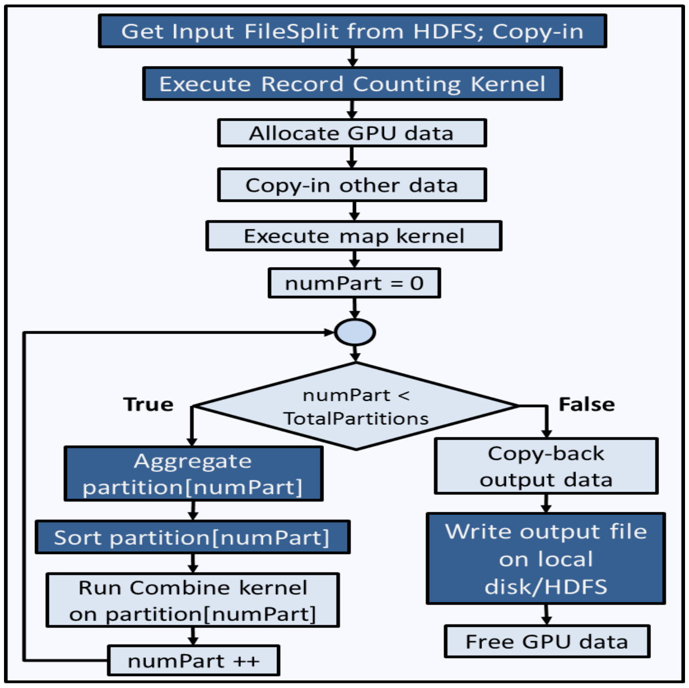

Get Input FileSplit from HDFS; Copy-in
Execute Record Counting Kernel
Allocate GPU data
Copy-in other data
Execute map kernel
numPart = 0
numPart < TotalPartitions
True
False
Aggregate partition[numPart]
Sort partition[numPart]
Run Combine kernel on partition[numPart]
numPart ++
Copy-back output data
Write output file on local disk/HDFS
Free GPU data

Figure 3.1.: Flowchart for the driver code on the host CPU : Dark boxes indicate functions provided by the runtime system

This leads to a scattering of the KV pairs belonging to each partition. These KV pairs must be aggregated by removing the whitespaces before they are sorted in the next phase, reducing the sort size. The indirection array (*indexArray*) is useful for performing this aggregation as the KV pairs do not need to be shuffled directly. Note that smaller global KV store size results in better aggregation efficiency. Next, sort, followed by the combine kernel, are run on each partition. The generated output is written back to a local disk. For map-only jobs, the output is written directly to the HDFS. Finally, the allocated memories are freed.

3.5 Overall Execution and Runtime System

This section describes the overall flow of the HeteroDoop framework and the runtime system. The runtime system assists the compiler-optimized code to run on a

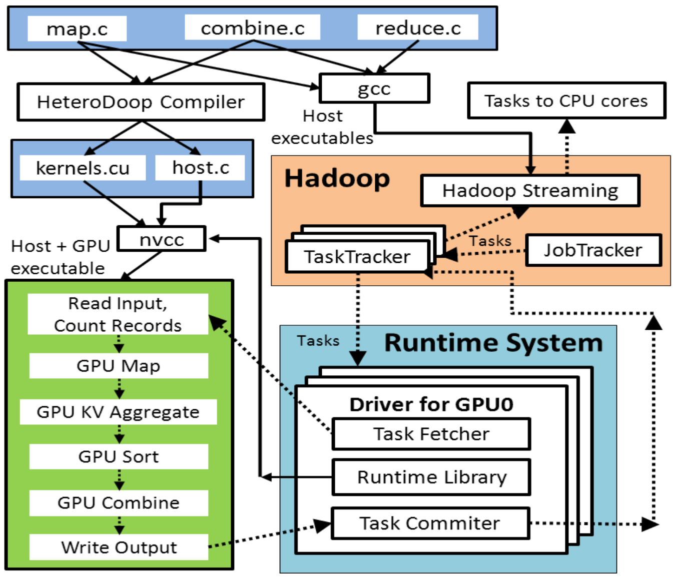


Figure 3.2.: Execution Scheme of HeteroDoop : Dashed arrows represent the execution flow; solid arrows represent the compilation flow

GPU by providing a number of library functions (Section 3.5.2), and helps implement the MapReduce semantics (Section 3.5.3).

3.5.1 Overall Execution

Fig. 3.2 shows the overall workflow in HeteroDoop and the roles of the runtime system. First, the user-written map and combine functions are translated by the HeteroDoop source-to-source compiler. Next, the generated CUDA code is compiled by the *nvcc* compiler for the GPU executable. The original source is compiled by *gcc* for the CPU executable. The executables are inserted into the Hadoop Streaming mechanism. When the execution starts, the JobTracker sends tasks to TaskTrackers, which schedule them either on the CPU cores, using native Hadoop Streaming, or on a GPU. The latter is coordinated by the *GPU driver* of the runtime system. The GPU driver fetches new tasks and invokes the GPU executables to perform map, combine and other MapReduce semantic-specific operations. Upon completion of a task, the GPU driver informs the TaskTracker about the task completion details, which comprise execution time, task log, etc.

Hadoop Integration and Fault Tolerance:

Fault tolerance of HeteroDoop for GPU tasks manifests itself in two ways: i) a task failure is communicated to the Hadoop scheduler so that it can reschedule the task; ii) the failed GPU is revived so that future tasks can still be issued to it. To obtain this fault tolerance and to achieve issuing of tasks to GPUs, we have modified Hadoop's *MapTask* class implementation to signal a new task to be run on the GPU to the GPU driver. TaskTrackers on each slave keep one slot reserved per GPU. Note that these slots simply offload the tasks on GPUs; no CPU time is consumed. Scheduling decisions on the GPU are described in the next section. Internally, the driver runs one thread for each GPU on the node. The driver assures that only a single task runs on the GPU at a time. If a thread's execution fails, the error is communicated to Hadoop TaskTracker, and the driver restarts the thread. In this manner, the GPU driver is made fault tolerant.

3.5.2 Assisting GPU Execution

The following runtime library functions help facilitate the GPU kernel execution.

File Handling:

HDFS does not follow the POSIX API for file handling; directly reading the input fileSplit from a C-code is not supported. The runtime system uses libHDFS [76], which provides a C/C++ function for fetching the input fileSplits from HDFS. The output of the map + combine execution is written to the local disk in a Hadoopcompatible binary format (*SequenceFileFormat*). For map-only jobs, the output is written directly to the HDFS, using libHDFS.

Record Handling:

The records in an input file must be pre-determined to support record stealing. The runtime system implements a GPU kernel that pre-determines and counts the records in the input file. This kernel is executed before the map kernel. The runtime system provides a thread-safe function, *getRecord*, which can be called by each thread of the map kernel to fetch a new record.

3.5.3 Implementing MapReduce Semantics

Performing Partition Aggregation:

After the map execution, KV pairs in each partition of the global KV store must be compacted to get rid of the whitespaces, resulting in an improved sorting efficiency. The runtime system provides an aggregation function that operates in parallel. It utilizes the count of the KV pairs emitted by each thread of the map kernel and an indirection array to locate KV pairs in the global KV store. A fast, parallel scan method for GPUs [77] is used to compute the prefix sum of the emitted KV pairs for each thread. From this information and the count of KV pairs emitted by each thread, a GPU kernel converts the old indirection array into a new one, wherein the generated KV pairs are aggregated.

Intermediate Sort:

Recall that the MapReduce model sorts the KV pairs after the map operation. HeteroDoop uses a modified version of the efficient GPU merge sort implementation [78]. The original implementation operates on fixed, short-length integer keys and values, making efficient use of the GPU shared memory. For long keys and values, this method would make inefficient use of the shared memory and thus restrict the partial merge size. Our implementation therefore forgoes direct key-based sorting. We instead modify the original method [78] to employ indirection for accessing the KV pairs. The use of indirection avoids expensive movements of the KV pairs in the device memory. We chose this implementation over an alternative merge sort approach [79] for variable-length keys. This alternative approach hashes the first few characters of the variable-length keys, breaking ties by comparing the next characters. We chose not to use this method, since hashing requires additional memory on the GPU.

3.6 Tail Scheduling

Hadoop's default scheduler does not take into account the disparity in the computational capabilities of the processors. When employing both CPUs and GPUs, task execution times vary substantially, requiring more advanced scheduling decisions. This section describes the HeteroDoop scheduler, aiming at optimal execution on accelerator clusters where each node has a CPU and one or more GPUs.

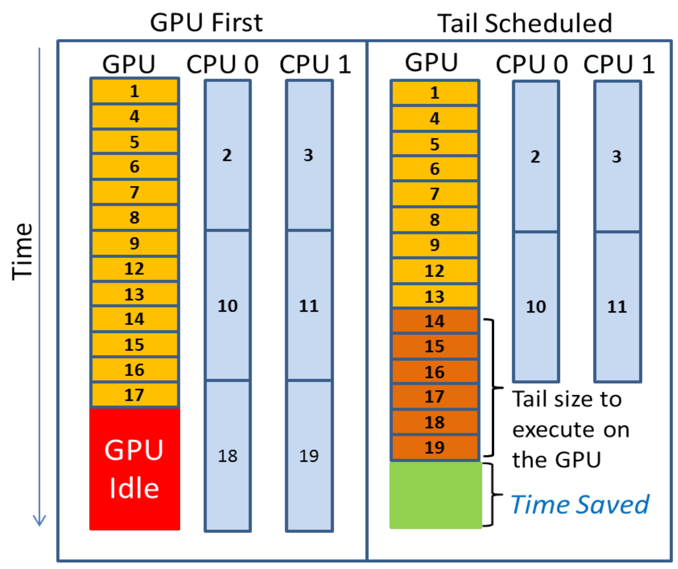


Figure 3.3.: Key Idea of Tail Scheduling

3.6.1 GPU-First Execution vs Tail Scheduling

A simplistic scheduler for CPUs and GPUs would act as follows. Whenever a new task is issued on a node, the task is scheduled on a GPU if such a device is free; otherwise, the CPU is chosen. We refer to this method as "*GPU-first.*"

Algorithm 5: Tail Scheduling on JobTracker and TaskTracker

```
Function TailScheduleOnJT(numGPUs, maxSpeedup, numSlaves)
    // this function executes on the JobTracker before
    // sending heartbeat response
    $jobTail = numGPUs \times maxSpeedup \times numSlaves$;
    // identifying tail
    if (jobTail ¡ getNumRemainingMaps()) then
        // assures fairness
        taskSet = scheduleNumGPUTasksAtMax();
    else
        taskSet = useHadoopDefaultScheduling();
    numMapsRemainingPerNode = getNumRemainingMaps()/numSlaves;
    heartBeatResponse.add(numMapsRemainingPerNode, taskSet);

// TaskTracker calculates GPU speedup over CPU for each task
Function TailScheduleOnTT(task, numGPUs, aveSpeedup,
numMapsRemainingPerNode)
    $taskTail = numGPUs \times aveSpeedup$;
    if (taskTail ¡= numMapsRemainingPerNode) then
        // tail is forced on GPU
        forceGPUexecution(task);
    else
        useGPUFirst(task);
```

GPU-first scheduling keeps all CPU and GPU cores busy until the final tasks show up. Consider the example scenario in fig. 3.3, which uses, 1 GPU and a 2-core CPU for scheduling a total of 19 tasks. The GPU slot is 6x faster than the CPU slot. The figure shows the sequence number of each task. In GPU-first scheduling, shown on the left, the 1st task would go on the GPU and the 2nd and 3rd on the CPU. Since the GPU finishes early, the 4th task would go again on the GPU. Continuing in this manner, the 17th task goes on the GPU, while the 18th and 19th stay on the CPU. Since the CPU tasks are much slower, the faster GPU remains idle at the end stage, which is sub-optimal. A smarter scheme, as shown on the right side of fig. 3.3, would force tasks 18 and 19 to execute on the GPU, saving on overall execution time. This is the key idea of tail scheduling.

A key challenge in realizing tail scheduling in HeteroDoop is for the slaves to know when their tails begin. As this information is not available to the slaves in the baseline Hadoop, in our algorithm, the JobTracker will estimate the number of remaining tasks for each node and communicate this information to the TaskTrackers, as described next.

We implement tail scheduling in HeteroDoop at two levels : on the JobTracker and on the TaskTracker. Algorithm 5 describes the mechanism. The TaskTracker continually calculates the average GPU slot speedup over the CPU slot and informs the JobTracker. The Hadoop heartbeat mechanism has been modified to carry this information. The JobTracker remembers the maximum speedup from the TaskTrackers. For the TaskTracker, the tail size (*taskTail*) is the number of GPU tasks that execute in the same amount of time as one CPU task. It is computed as the number of GPUs on the current node (*numGPUs*) multiplied by the average GPU speedup (*aveSpeedup*) seen on that TaskTracker.

The JobTracker notifies each TaskTracker about the remaining number of map tasks to be scheduled (*numMapsRemainingPerNode*). As task scheduling on the JobTracker is on a first-come-first-serve basis, it does not exactly know how many tasks would go on a given TaskTracker. The JobTracker estimates *numMapsRemainingPerNode* as the total number of remaining tasks divided by the number of slaves (line 8). Whenever *taskTail* is greater than *numMapsRemainingPerNode*, *GPU-first* scheduling is used (lines 13–17). Otherwise, all the next tasks - across all slots of the

TaskTracker - are forced for GPU execution. As all slots on a TaskTracker force their tasks on the GPU(s) once the *taskTail* begins, queuing might occur on the GPU(s). To counter this effect, the JobTracker only schedules at most *numGPUs* tasks on a TaskTracker per heartbeat once the *jobTail* begins. *JobTail* is the total number of tasks all GPUs in the cluster can finish in the same amount of time as a single CPU core. It is estimated as $numGPUs \times maxSpeedup \times numSlaves$, where *numSlaves* is the total number of slave nodes in the cluster and *maxSpeedup* is the maximum GPU speedup seen across the cluster. *ScheduleNumGPUTasksAtMax* replaces the original Hadoop scheduling scheme, which schedules tasks up to the number of empty GPU and CPU slots per heartbeat (lines 3–7).

3.7 Evaluation

This section evaluates the HeteroDoop system. We present the overall performance achieved on our heterogeneous CPU-plus-GPU system versus a CPU-only scheme, i.e. baseline Hadoop. To analyze the performance in further detail, we show individual task speedups of GPU over CPU execution. We also present the performance benefits of HeteroDoop compiler optimizations on individual kernels.

3.7.1 Benchmarks

We used eight benchmarks for our experiments. *Grep*, *wordcount*, *kmeans*, *classification*, *histmovies* and *histratings* are taken from the PUMA benchmark suite [80]. They represent typical MapReduce applications, including both IO-intensive and compute-intensive ones. Apart from these, we evaluate two scientific computation applications, *blackScholes* and *linear regression*, that have been shown to be amenable to MapReduce [30, 70]. *Grep* and *wordCount* are well-known MapReduce applications,

Table 3.2.: Description of the Benchmarks Used

Benchmark	**%Exec. Time Map + Combine are Active**	**Nature (Intensiveness)**	**Combiner**
Grep (GR)	69	IO	Yes
Histmovies (HS)	91	IO	Yes
Wordcount (WC)	91	IO	Yes
Histratings (HR)	92	Compute	Yes
Linear Regression (LR)	86	Compute	Yes
Kmeans (KM)	89	Compute	No
Classification (CL)	92	Compute	No
BlackScholes (BS)	100	Compute	No

Table 3.3.: Description of the Benchmarks Used

Benchmark	**Total Reduce Tasks**		**Total Map tasks**		**Input Size (GB)**	
	Cluster1	**Cluster2**	**Cluster1**	**Cluster2**	**Cluster1**	**Cluster2**
Grep (GR)	16	16	7632	2880	902	340
Histmovies (HS)	8	8	4800	640	1190	159
Wordcount (WC)	48	32	5760	1024	844	151
Histratings (HR)	5	5	4800	2560	591	160
Linear Regression (LR)	16	16	2560	3840	714	356
Kmeans (KM)	16	16	4800	NA	923	NA
Classification (CL)	16	16	4800	3200	923	72
BlackScholes (BS)	0	0	3600	5120	890	210

and both are IO intensive. *Kmeans* is an iterative clustering application. While *kmeans* is a centroid-based clustering algorithm, other variants such as distribution-based or density-based clustering are popular as well. All these clustering algorithms are known to be compute intensive. *Classification* benchmark is derived from statistical classification algorithms. It is similar to *kmeans*; however, there is no clustering involved. The application ends after classifying the input dataset to respective centroids in a single iteration. Histograms are common in big-data applications. *Histmovies* and *histratings* are two such applications. *Histmovies* averages the review ratings for each movie in a given dataset and puts these averages into bins. *Histratings* directly bins each review rating for all movies in the dataset. Since the combiner receives larger data to operate on, *histratings* becomes more compute intensive than *histmovies*. *BlackScholes* is a financial pricing model, with explicit parallelism. Other financial applications with similar workloads include *binomial options* and *SOBOL quasi random number generator*. *Linear regression* is a special case of polynomial regression, and is commonly used in curve-fitting applications. For *blackScholes*, we ran 128 iterations per option. For *linear regression*, each input file contained data for 12 regressors, with 32 rows each. The details of the benchmarks, data sizes and tasks are presented in tables 3.2 and 3.3.

3.7.2 Cluster Setups

Table 3.4 shows the configurations of the two clusters used in this evaluation. Cluster1 is our primary platform. Each node has one GPU, which is a newer generation device. To evaluate the scalability of HeteroDoop on a multi-GPU system, we use Cluster2, which includes three older-type GPUs per node. This cluster is an in-memory system. There are no disks; input, output and temporary storage all exist in the main memory.

Table 3.4.: Cluster Setups Used

	Cluster1	**Cluster2**
#nodes	48 (+1 master)	32 (+1 master)
CPU	Intel Xeon E5-2680	Intel Xeon X5560
#CPU cores	20	12
GPU(s)	Tesla K40 (Kepler)	3×Tesla M2090 (Fermi)
RAM	256GB	24GB
Disk	500GB	none
Communication	FDR InfiniBand	QDR InfiniBand
Hadoop Version	Hadoop 1.2.1	Hadoop 1.2.1
CUDA Version	CUDA 6.0	CUDA 5.5
HDFS Block Size	256MB	256MB
HDFS Replication Factor	3	1
Max. Map Slots Per Node	20 (+1 for GPU runs)	4 (+1/GPU for GPU runs)
Max. Reduce Slots Per Node	2	2
Speculative Execution	Off	Off
% map tasks to finish before reduce starts	20	20

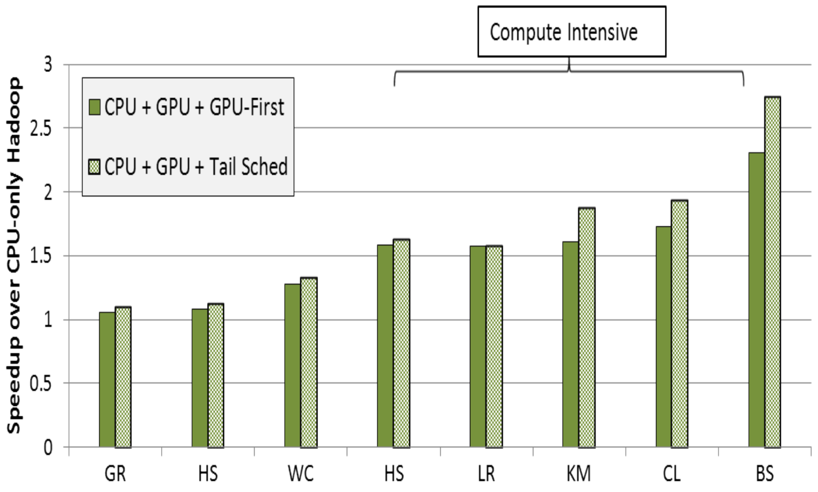


Figure 3.4.: Performance gains of HeteroDoop normalized to CPU-only Hadoop on Cluster1

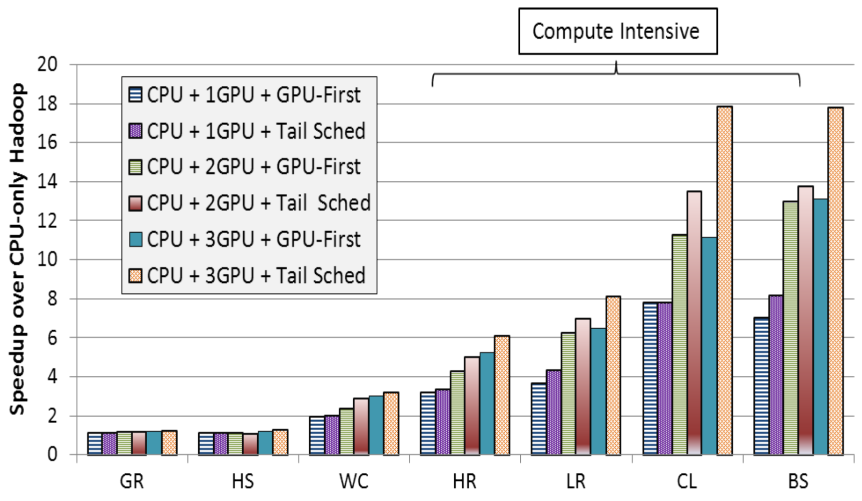


Figure 3.5.: Performance gains of HeteroDoop normalized to CPU-only Hadoop on Cluster2

3.7.3 Overall Improvements

Fig. 3.4 and Fig. 3.5 show the overall performance of HeteroDoop over the CPU-only Hadoop baseline for Cluster1 and Cluster2, respectively. We ran each experiment three times, and report the best run. The variation across runs was low (¡5%). The benchmarks are arranged by increasing speedup. As expected, HeteroDoop greatly outperforms CPU-only Hadoop on compute-intensive benchmarks owing to the GPU acceleration. For IO-intensive benchmarks, the use of CPUs brings about most of the achievable performance, indicating that execution on CPUs is essential in IO-intensive applications. In these benchmarks, HeteroDoop speedups are higher for Cluster2 than for Cluster1 because i) Cluster1 uses more CPU cores than Cluster2, and ii) Cluster2 is an in-memory system, reducing the IO-intensiveness of the applications. Fig. 3.5 shows multi-GPU scalability results of HeteroDoop. *KM* is not shown, as the memory requirement exceeds the capacity of Cluster2. The figures also show the effectiveness of *tail scheduling* over *GPU-first* scheduling. Note that tail scheduling improves performance only if the GPU(s) go idle at the end of the execution. For *LR* on Cluster1, this imbalance does not arise. For the IO-intensive benchmarks, the GPU speedup over a CPU core is low, resulting in only marginal benefits of HeteroDoop and tail scheduling.

3.7.4 Detailed Analysis

We present detailed analyses of the performance obtained on our primary cluster, Cluster1. Cluster2 showed similar trends. Fig. 3.6 shows the speedups obtained by a GPU task over a CPU task run by a single core, with the compiler-translated baseline code and with the optimizations. The optimizations include vectorization of map and combine kernels, using texture memory, record stealing and performing KV

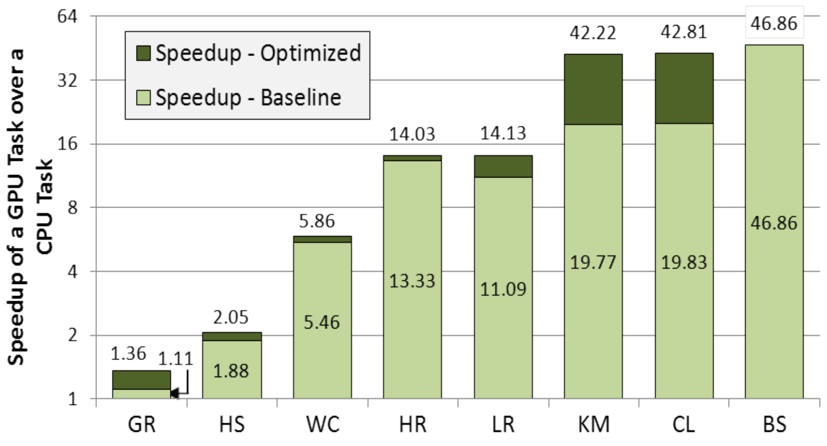


Figure 3.6.: Speedup of a single GPU task over a CPU task : Optimizations have high impact on three benchmarks

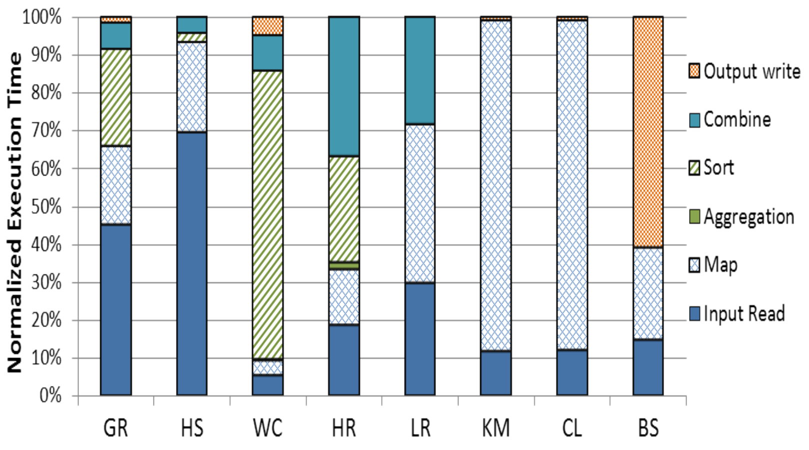


Figure 3.7.: Execution time breakdown of a GPU task

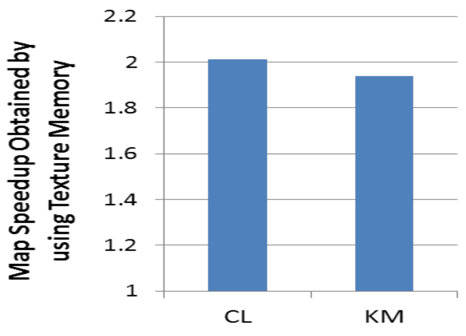


Figure 3.8.: Effects of using Texture Memory

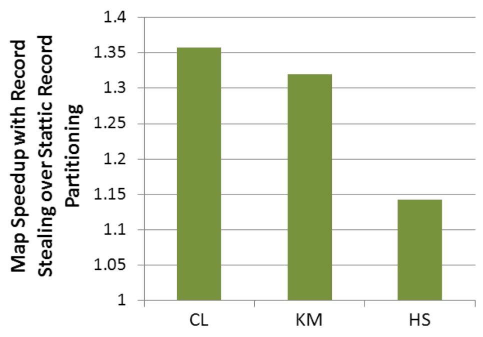


Figure 3.9.: Effects of Record Stealing on map kernels

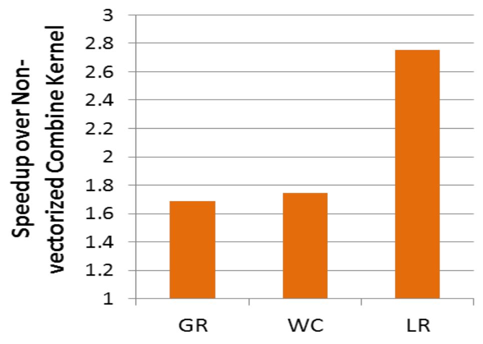


Figure 3.10.: Effects of Vectorized Read-Write on combine kernels

pair aggregation prior to sort. The GPU task comprises input copy, record counting, map, aggregate, sort, combine, and output write operations. The benchmarks are sorted by increasing speedups of their tasks. We attribute the increasing speedup to higher compute intensiveness. Each bar in the figure shows the speedup achieved by the baseline-translated code and the additional benefit of the optimizations. In *GR*, *KM*, *CL*, and *LR*, the optimizations gain substantial additional performance. Note that even for IO-intensive applications, such as *GR* and *HS*, the GPU achieves speedups over a single CPU core, and therefore, executing MapReduce tasks on GPUs along with CPU cores is still beneficial in these applications.

Fig. 3.7 breaks down the execution time for a single GPU task on each benchmark. Input reading time is the time for reading the HDFS file. Output writing time

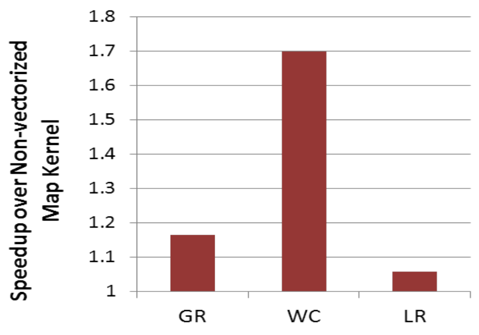


Figure 3.11.: Effects of Vectorized Read-Write on map kernels

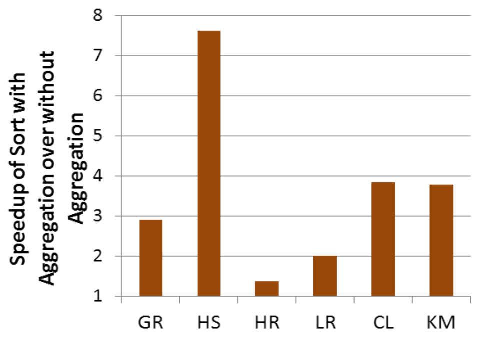


Figure 3.12.: Effects of KV Pair Aggregation on sort kernels

measures the time required for formatting the generated GPU output in Hadoop binary format, calculating the checksum, and writing the data on the HDFS/local disk, depending on whether or not the application is map-only. Fig. 3.7 shows that different stages of the MapReduce scheme can form a bottleneck on the GPU for different benchmarks. Note that the partition aggregation times are negligible in all benchmarks. Both these experiments (fig. 3.6, fig. 3.7) made use of input data-local map + combine tasks. These figures indicate that a single-task speedup can be as high as 47x for *BS*, which is our most compute-intensive application. The GPU task of *BS* spends 62% of its execution time writing the output, which is up from 1% in a CPU task. Evidently, the high computational power of GPUs moves the bottleneck from computation to disk write. *CL* and *KM* are both map-heavy operations. *Wordcount* shows an interesting case where most of the execution time is spent in sorting since it emits many long-length keys. *HR* and *LR* spend substantial execution time in the combine operation.

Fig. 3.8, 3.9, 3.10, 3.11 shows the effects of individual optimizations. The first is the usage of texture memory, which can speed up the map kernel in *KM* and *CL* benchmarks by 2x (fig. 3.8). Vectorized read and write of the KV pairs in the combine kernel can improve this operation by 2.7x, as shown in fig. 3.10. Vectorized read and write in the map kernel can yield gains of up to 1.7x (fig. 3.11). A speedup of up to 1.36x is gained by the record stealing scheme on map kernels as shown in fig. 3.9. Aggregating KV pairs in each partition prior to sorting can improve the sort kernel's performance by up to 7.6x (fig. 3.12).

3.8 Related Work

Single Node Systems: MapReduce has been studied as a programming model for several architectures. Two approaches [81, 82] target multi-cores. Catanzero et.

al. [31] proposed a MapReduce system for GPUs. Mars [29] uses MapReduce as a programming model for a single GPU node. MapCG [34] offers a MapRedue API that is portable across CPUs and GPUs. But all these systems lack a distributed framework, and do not scale across a cluster.

Distributed Systems: GPMR [30] uses CUDA + MPI as a platform for writing applications, but does not employ CPUs in a cluster. As both CUDA and MPI require significant programmer expertise to organize the parallelism and manage communication, programmability of GPMR is low. MITHRA [70] is a system that shows the effectiveness of using Hadoop and GPUs together in two scientific applications. Unlike HeteroDoop, MITHRA relies on the programmer to write CUDA code for the application. MITHRA uses Hadoop Streaming as well, and supports only the GPUs in a cluster.

Employing CPUs and GPUs: HadoopCL [83] presents a Java-based approach, where the programmer must write the map and reduce functions in GPU-friendly classes. HadoopCL performs bytecode translation of the input Java program into an OpenCL program, for both CPU and GPU. A key difference between HadoopCL and HeteroDoop is the automatic parallelization on the CPU cores; only a single task is run across all the CPU cores, because it is executed as an OpenCL program. While this strategy works for a single job on a cluster, it would require significant changes in the Hadoop scheduler when jobs have to share a cluster. In contrast to HeteroDoop, HadoopCL lacks architecture-specific clauses that can improve the program performance. Unlike HeteroDoop, HadoopCL does not have a runtime system to handle MapReduce semantics, and therefore, they must be handled on the CPU. Glasswing [33] uses another approach to utilize both CPUs and GPUs for MapReduce. It requires the input map and reduce kernels to be written with OpenCL and offers a unified API to feed these kernels to the framework.

Scheduling Hadoop Tasks on a CPU-GPU Cluster: Shirahata et. al. [84] present a strategy where map tasks can be run on both CPUs and GPUs. The programmer is expected to provide the corresponding CPU/CUDA codes. This work employs a mathematical performance model for a heterogeneous scheduling strategy that minimizes the overall execution time for the given constraints. However, such minimization requires evaluation of the performance model throughout the job execution. By contrast, tail scheduling affects only the final tasks, reducing the scheduling overhead.

GPU-Specific MapReduce Optimizations: Chen et. al. present a system [85] and optimizations for running MapReduce on CPU-GPU coupled architectures, where the CPU and GPU share the system memory. Another approach [86] presents a system that makes better use of the on-chip shared memory of the GPU for running reduction-intensive applications. Both these systems present important optimizations for MapReduce as a programming model on a single node. However, they are not directly applicable to distributed MapReduce since the map and reduce stages may run on different nodes. Also, GPU-shared-memory based optimizations for storing intermediate KV pairs would be precluded.

3.9 Chapter Takeaways

This chapter has presented HeteroDoop, a MapReduce framework and system that employs both CPUs and GPUs in a cluster. It has introduced HeteroDoop directives, that can be placed on an existing sequential, CPU-only, MapReduce program for efficient execution on both the CPU and one or more GPUs on each node. The optimizing HeteroDoop compiler translates the directive-augmented programs into efficient GPU codes. HeteroDoop's runtime system assists in carrying out the compiler optimizations and handling MapReduce semantics on the GPUs. HeteroDoop is built on top of Hadoop, the state-of-the-art MapReduce framework for CPUs. The HeteroDoop runtime system plays a vital role in this integration. A novel scheduling scheme optimizes the execution times of jobs on CPU-GPU heterogeneous clusters. Our experiments with HeteroDoop in eight benchmarks demonstrate that using a single GPU per node can achieve speedups of up to 2.78x, with a geometric mean of 1.6x, compared to a cluster running CPU-only Hadoop on 20-core CPUs. The proposed tail scheduling scheme works well for intra-node heterogeneity. We leave handling of extreme inter-node heterogeneity to future work, where the trade-off between data locality and execution speed will be an important additional consideration.

This and the previous chapters have dealt with issues of GPU program tuning and automatic scaling to single-GPU, multi-GPU, and multi-node systems. All of these chapters have shown optimizations that work in either intra-kernel or inter-kernel manner. In the next chapter, we are going to look at the problem of control flow structuring, which occurs at an even finer level – at the granularity of each warp in a kernel.

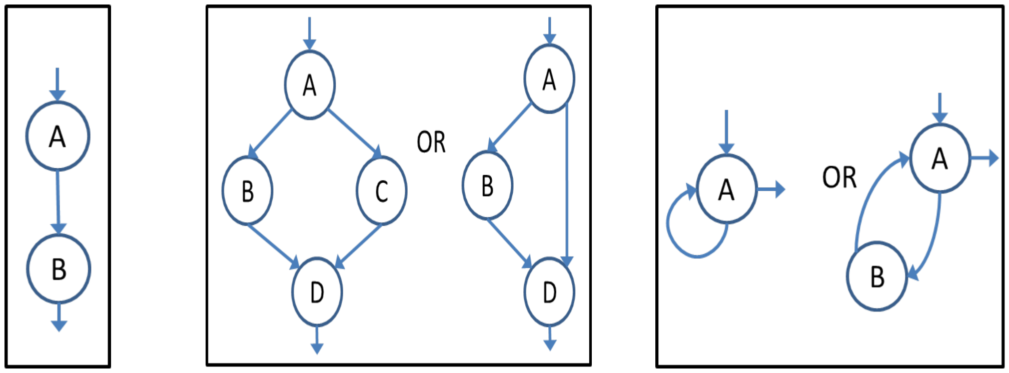
4. FORMALIZED CONTROL FLOW STRUCTURING WITHOUT CODE EXPLOSION

This chapter proposes a compiler technique to lower the penalty of divergent execution on accelerators with SIMD execution mechanism. This penalty gets exacerbated on "unstructured" control flow graphs (CFGs). The chapter first formalizes the notion of structured CFGs, and then presents a mechanism to convert unstructured CFGs into structured ones.

4.1 Introduction

Structured programming is a paradigm where programs are written using just three base constructs [37, 87], namely, i) sequence of statements, ii) **if**-then-**else** blocks, and iii) loops. Structured programming was sought for many reasons, two important ones being readability of the program [37, 40] and ease of analyzing the control flow [88]. While optimizing programs, compilers operate on the control flow graph (CFG) of the program, which is usually built upon an intermediate representation (IR), and not on the program source itself. A CFG is a directed graph in which the nodes represent basic blocks and the edges represent control flow paths [88]. As compilers work on CFGs, structuredness of CFGs becomes an important consideration. A key insight of this paper is that structured CFGs do not follow directly from structured program source codes.

The common notion of structured CFGs considers three base patterns [89, 90], similar to the structured program constructs mentioned above. They comprise a sequence (Fig. 4.1a), a selection (Fig. 4.1b), and a loop (Fig. 4.1c). We argue that



(a) Sequence (b) Selection (c) Loop

Figure 4.1.: Base Structured Patterns

such a pictorial depiction of base patterns alone is insufficient and leads to imprecision. The "definition" fails to clearly distinguish a structured CFG from an unstructured one. Consider the CFGs in Fig 4.2, which do not show any obvious matching to the base patterns. Creating structured CFGs contrasts with structured programming, where just by looking for the presence of unstructuring-causing constructs, such as `goto` and `break` statements, unstructuredness can be easily detected. The difficulty in doing the same in CFGs arises because the pictorial representations of the base structured patterns do not show how to compose the patterns into larger CFGs or decompose CFGs.

This paper formalizes the notion of base structured patterns by providing definitions for each of them and presenting a conceptual framework, called *folding*, that replaces base patterns by single nodes. The repeated application of folding determines whether a given CFG is structured or not.

Structured CFGs are desirable for a number of reasons. A key reason is the ease of program analysis, as structured CFGs cannot contain *irreduciblity* [1]. Irreducibility is a condition where one or more loops in a CFG have more than one entry points. Ir-

[1]We recommend e-reading to benefit from the hyperlinks

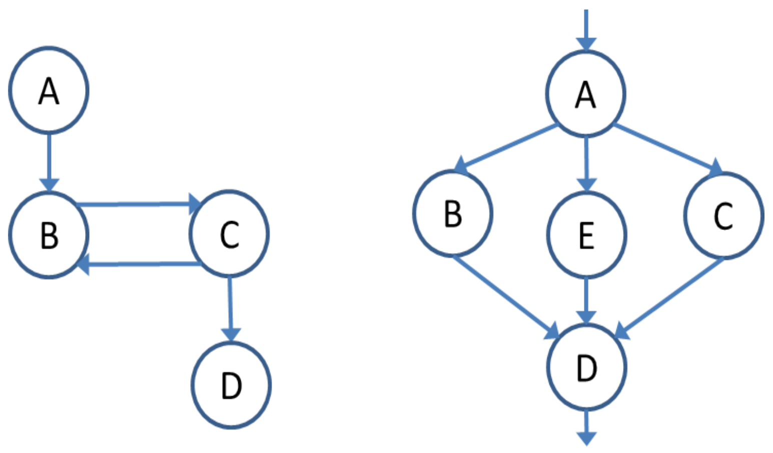


Figure 4.2.: Are these CFGs structured? The base patterns in Fig. 4.1 fail to answer.

reducible CFGs are difficult to analyze. Many compiler analyses and transformations take place only if the CFG is reducible [91–97]. Structured CFGs are considered easier to understand. A measure of control flow complexity is the number of *knots* [98]. A knot is an unavoidable crossing of two edges in a CFG. Structured CFGs have no knots, and hence are less complex. Unlike unstructured CFGs, structured CFGs can be translated back into high-level source codes, e.g., a Java bytecode with irreducibility, which is unstructured, cannot be translated into Java source, since `goto` statements are not supported in Java. Third, SIMD architectures such as GPUs have poor or no support for unstructured CFGs, which may lead to control flow divergence [35] resulting in performance degradations.

For the above reasons, techniques to remove unstructuredness are necessary. Many prior proposals have offered mechanisms to convert unstructured programs into structured ones [36–40]. They work on the program source to eliminate constructs that cause unstructuredness, such as `goto` and `break`, with an assumption that the output structured programs are always converted into structured CFGs by the underlying compiler. Table 4.1 (Section 4.8.3) will show that this indeed is not the case in practice; compiler front-ends can generate unstructured CFGs from structured programs. Furthermore, compiler optimizations can introduce unstructuring/irreducibility as well, e.g, short-circuiting, inlining [35], jump threading, and tail-call elimination [99]. Therefore, techniques to remove unstructuredness that work directly on CFGs are required.

Another issue with existing structuring techniques, both those that operate on program source [36–40, 100–103] and those that operate on CFGs [104, 105], is that they resort to a technique named *Node Splitting* [106] to remove irreducibility. Node splitting operates by duplicating parts of the code. It has been shown that Node Splitting-based mechanisms may lead to exponential code blowup [107], rendering any CFG analysis impossible. Conventional compilers, such as `llvm` and `gcc`, forgo Node Splitting, and disregard optimizations on irreducible loops. This is not the best approach though; presence of a single irreducible loop has been shown to restrict compiler optimizations on other reducible loops, due to loop nesting [108].

The exponential code expansion aspect of irreducible CFGs has found interest in the software obfuscation field. Software obfuscation intends to make it practically impossible for an attacker to statically determine program properties. To do so, it changes the nature of the code while retaining the functionality. A common obfuscation technique inserts dummy edges in the CFG to make it irreducible [107, 109, 110]. Since the equivalent reducible CFG would be exponentially larger, attacker's tools would fail to perform reasonable analyses.

This chapter presents a novel algorithm to remove unstructuredness from CFGs. It operates directly on arbitrary CFGs to generate structured CFGs. The algorithm avoids code duplication in *all circumstances*. The algorithm is based upon two novel transformations that reorganize the control flow by inserting additional variables that act as predicates. An important result proven in this chapter is that, unlike the Node

Splitting-based mechanisms, the predication-based technique proposed in this work does not face blowup and restricts the code expansion to only a polynomial function of the original code size. Furthermore, we show that the additional control flow inserted by our mechanism does not restrict traditional compiler analyses. This is a word of caution to obfuscators that operate by introducing irreducibility. The chapter presents experimental evaluation showing that while traditional compilers fail to analyze the CFGs obfuscated by this approach, application of the proposed structuring mechanism unlocks such analyses.

To summarize, the contributions of this chapter are:

- The chapter formalizes the notion of structured CFGs by presenting precise definitions of the three base patterns that constitute a structured CFG. It introduces *folding*, a conceptual framework that helps determine if a given CFG is structured.

- It proposes two novel transformations and subsequently an algorithm that converts an arbitrary unstructured CFG into a structured CFG, without requiring code duplication.

- The chapter proves that this algorithm avoids exponential blowup in even those cases where the input CFG is irreducible.

- We experimentally show that obfuscation by irreducibility insertion is rendered ineffective with the help of the proposed structuring mechanism.

The rest of this chapter is organized as follows : Section 4.2 presents preliminary CFG concepts that are required to understand our transformations. Section 4.3 presents definitions that represent various phenomena in CFGs and theorems that prove certain properties of CFGs. Section 4.4 builds on them, and describes the two structuring transformations. Section 4.5 presents an algorithm that uses these transformations to convert an unstructured CFG into a structured one. Section 4.6 analyzes the growth in code size caused by this algorithm and proves that the code expansion is limited by a polynomial function of the number on nodes in the CFG. Section 4.7 describes related work. Section 4.8 presents experimental results, while Section 4.9 presents the takeaways from the chapter.

4.2 Preliminaries

This section presents common definitions and concepts that lay foundations of this work.

We consider CFGs with a single entry and exit nodes. There are no infinite loops, i.e, there exists a path from each node in the CFG to the exit node. Symmetrically, each node in the CFG can be reached from the entry node. Also, we assume that the maximum in-degree and out-degree is two. Generality is maintained since a CFG with any in-degree or out-degree can be converted into a CFG with a maximum in-degree and out-degree of two.

Definition 4.2.1 ***Path*** *: A path between nodes A and B is an ordered list of adjacent edges and vertices. The list begins with an out-edge of A, and ends with an in-edge of B.*

Definition 4.2.2 ***Region*** *: The region between two nodes (edges) A and B contains all nodes and edges that are present on any path from A to B. If the starting/ending points are nodes, they are included in the region too.*

Definition 4.2.3 ***Dominator*** *: A node (edge) P is a dominator of a node (edge) Q if every path from the entry node of the CFG that reaches Q has to pass through P.*

Each node or edge dominates itself. Each dominator of a given node (edge), except itself, is said to be a *strict* dominator of the node (edge).

Definition 4.2.4 ***Post-dominator*** *: A node (edge) Q is a post-dominator of a node (edge) P if every path from P to the exit node of the CFG has to pass through Q.*

Each node or edge post-dominates itself. Each post-dominator of a given node (edge), except itself, is said to be a *strict* post-dominator of the node (edge). The dominator (post-dominator) relationships allow construction of a dominator (post-dominator) tree of the CFG, wherein the parent of a node is its strict dominator (post-dominator). The parent node of a given node in the post-dominator tree is known as the immediate post-dominator (IPDOM) of that node.

Definition 4.2.5 ***Single-entry-single-exit (SESE) region*** *: The region between two nodes (edges) A and B is SESE if all of the following are true:*

- *A dominates B*
- *B post-dominates A*
- *Every cycle containing A also contains B and vice versa.*

A node (edge) is a SESE region by itself. Each base pattern in Fig. 4.1 represents a SESE region. For the selection pattern (Fig. 4.1b), node A is the entry and node D is the exit. For the loop pattern (Fig. 4.1c), node A is the entry as well as the exit.

Definition 4.2.6 ***Loop condition node, loop path*** *: A given condition node N (N has two out-edges) is said to be a loop condition node, if there is a simple path (all nodes along the path have in/out-degree of one) that originates and ends at N. We refer to any such path as a loop path.*

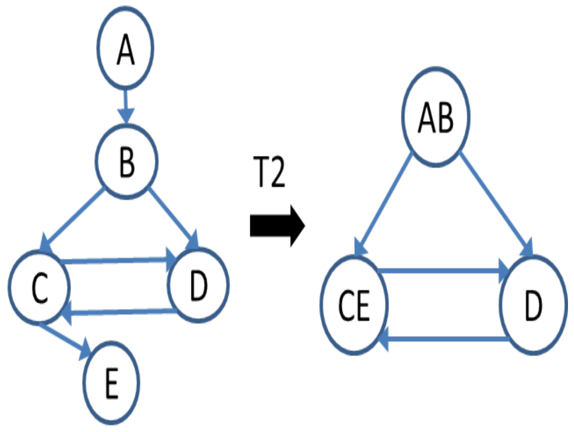


Figure 4.3.: Example of an irreducible CFG

Next, we describe the transformations that determine if a CFG is reducible or not. We attribute the definitions to Hecht and Ullman [111].

Definition 4.2.7 ***T1*** *: T1 is a transformation that removes an edge from a node onto itself.*

Definition 4.2.8 ***T2*** *: If a node B has a single predecessor node A, then transformation T2 replaces nodes A and B with a single node C. Predecessors of A become the predecessors of C. Successors of A or B become successors of C.*

Definition 4.2.9 ***Reducible CFG*** *: A CFG is reducible iff it becomes a single node through repeated applications of T1 and T2, otherwise, it is said to be* ***irreducible****.*

Fig. 4.3 shows an example of an irreducible CFG, where neither T1 nor T2 can be applied.

4.3 Analyzing Unstructuredness

This section formalizes the notion of structured CFGs by providing formal definitions for the structured base patterns. It introduces the framework of *folding*, which decomposes CFGs into base patterns. The section introduces terminology to define elements of a CFG, and presents a set of theorems on which the structuring algorithm described in the next sections is built upon.

We begin by defining structured selection and loop condition nodes (nodes with two out-edges).

Definition 4.3.1 ***Structured selection condition node, selection body*** *: A condition node N is a structured selection condition if for every path from N to its IPDOM, the region between the first and last edges is SESE. Therefore, the region between the structured selection condition and its IPDOM is SESE as well, which is said to be its selection body.*

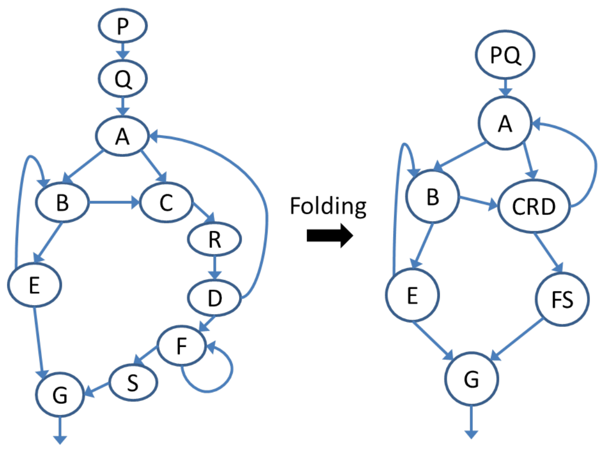


Figure 4.4.: Maximal folding

Definition 4.3.2 ***Structured loop condition node, loop body*** *: A structured loop condition node is a loop condition node where there exists a SESE region between one of its out-edges and in-edges. This SESE region is called the loop body.*

Definition 4.3.3 ***Unstructured condition node*** *: If a condition node is neither a structured selection condition node, nor a structured loop condition node, then it is an unstructured condition node.*

Examples: Node A in the base selection pattern (Fig. 4.1b) is a structured selection condition. Node A in the base loop pattern (Fig. 4.1c) is a structured loop condition, with edge A→A being both the entry and exit edge of the SESE region (left figure) or edge A→B being the entry edge and the edge B→A being the exit edge of the SESE region (right figure). For condition node A in Fig. 4.4, G is the IPDOM. A→B →E→G is one path from A to G, on which the region between the first and last edges is not SESE. Hence, A is not a structured selection node. Similarly, for no out-edge and in-edge pair of A, there exists a SESE region. Therefore, A is not a structured loop condition node either. Hence, A is an unstructured condition node.

Now, we present formal definitions for the **base structured patterns** shown in Fig. 4.1.

Definition 4.3.4 ***Sequence*** *: Two nodes, A and B, along with an edge $A \to B$ are said to form a sequence if B is the sole successor of A, and A is the sole predecessor of B.*

Definition 4.3.5 ***Selection*** *: The pattern of selection contains a structured selection condition node, its IPDOM, and the selection body. The selection body must contain at least one node, and any path from the selection condition node to the IPDOM can have at most one node.*

Definition 4.3.6 ***Loop*** *: The pattern of loop contains a structured loop condition node, the loop body, and the entry and exit edges of the loop body. The loop body can contain at most one node.*

To determine if a CFG is structured, we introduce a new concept, called *folding*.

Definition 4.3.7 ***Folding*** *: Folding is a process of replacing a base structured pattern with a single node in the CFG. During folding, any edge not belonging to the base pattern, but having its source (sink) node in the base pattern, is redirected so that the newly created single node is its source (sink).*

Definition 4.3.8 ***Maximal Folding*** *: Maximal folding repeatedly applies folding to a CFG until no more base structured patterns exist.*

Fig. 4.4 shows an example CFG and its maximally folded equivalent.

Definition 4.3.9 ***Completely Foldable CFG*** *: If a maximally folded CFG contains a single node, then the CFG is called completely foldable.*

Definition 4.3.10 ***Structured CFG*** *: A CFG is said to be structured iff it is completely foldable. Otherwise, it is called an* ***unstructured CFG****.*

Complete foldability, i.e., structuredness, implies that the CFG is composed of the base structured patterns. While reducibility eases CFG analysis, it does not determine if the CFG is structured or not. E.g., the CFG in Fig. 4.4 is reducible, but is unstructured.

Definition 4.3.11 ***Structured Region*** *: A completely foldable region is called a structured region.*

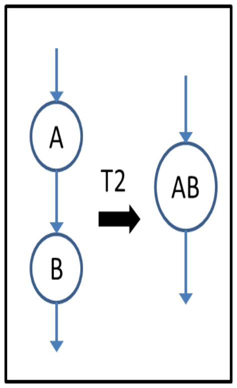
Theorem 4.3.1 *Structured CFGs are reducible.*

Proof Since a structured CFG is completely foldable, the idea here is to show that each base structured pattern is reducible. As shown in Fig. 4.5a, a sequence can be reduced into a single node by applying T2. Fig. 4.5b shows that the repeated application of T2 can reduce the base selection pattern, while Fig. 4.5c shows that application of T2 (if the loop is not a self loop), or T1 (if the loop is a self loop) can reduce the base loop pattern. The process of folding replaces a base structured pattern with a single node. Since each base pattern can be reduced by T1 and/or T2, it follows that instead of folding a base pattern, one can apply T1 and/or T2, and a continued application would result in a CFG containing a single node, implying reducibility. ■

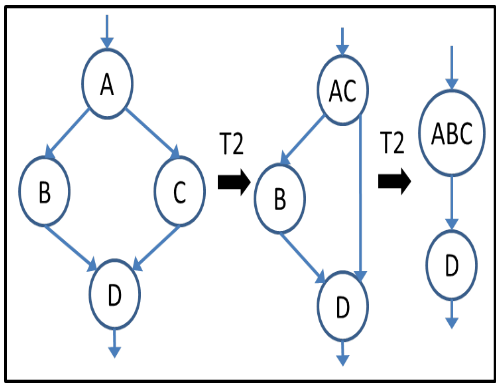
Corollary 1 *Every irreducible CFG is unstructured.*

Proof Follows directly from Theorem 4.3.1. ■

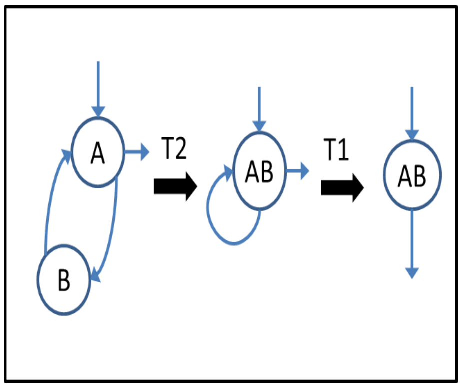
Lemma 1 *In a maximally folded CFG that can not be completely folded, there must be at least one unstructured condition node.*



(a) Sequence



(b) Selection



(c) Loop

Figure 4.5.: Reducibility of base patterns

Proof Let us assume that all condition nodes in a CFG are either structured selection conditions or structured loop conditions. Then, there must be some (innermost) condition node whose body does not have a condition node. There are two possibilities for this node:

Case 1: Structured Selection Condition : In this case, there can only be two distinct, simple paths originating at this node that reach its IPDOM, since there are no condition nodes in the selection body. By the definition of a structured condition node, nodes on each of these paths are dominated by their first edge, and hence none of their nodes can have two in-edges. Therefore both these paths can only contain nodes with a single predecessor and a single successor, however, as the CFG is maximally folded, there can at most be one node on either of these two paths. Hence, the selection condition node, its IPDOM, and, the selection body would match the base structured selection pattern.

Case 2: Structured Loop Condition : In this case, this node's loop body can only have nodes with a single predecessor and a single successor. With a similar argument as in case 1, the loop body can only have at most one node. Hence, the loop condition node, the loop body, and its entry and exit edges would match the base structured loop pattern.

Thus, in both cases, a base structured pattern would exist in the CFG, which is a contradiction. Hence, a maximally folded, but not completely folded CFG must have an unstructured condition node.

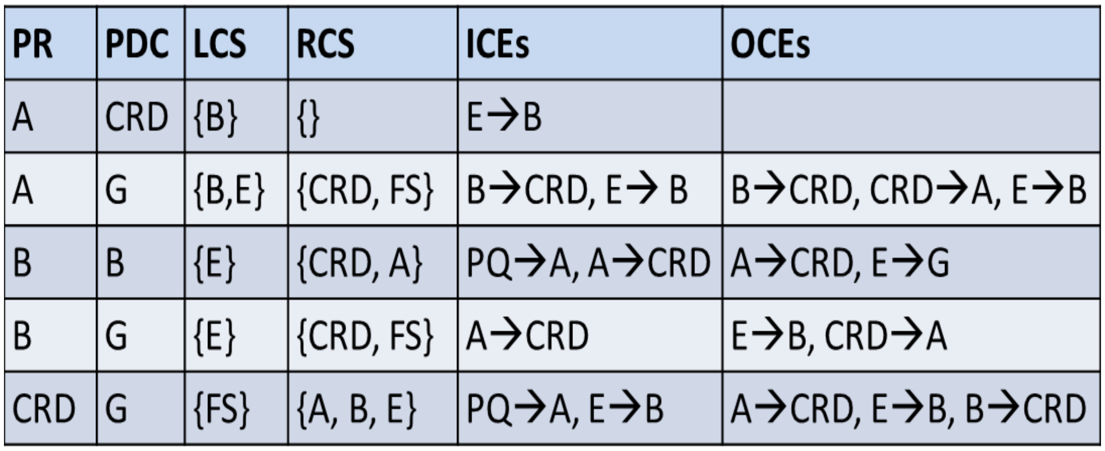
■

So far, this section has described the nature of structured CFGs, and how to distinguish them from unstructured CFGs. We now present new terminology, and present certain theorems that are crucial in converting unstructured CFGs to structured ones.

Definition 4.3.12 ***Pseudo Root (PR)****: PR is any condition node in the CFG.*

Definition 4.3.13 ***Post-dominator Candidate (PDC)****: PDC is a node that can be reached from both out-edges of a given PR through non-overlapping simple paths, neither of which contains the PR itself. These two paths are called* ***Left Path*** *and* ***Right Path****; the former starting at the condition true edge, and the latter beginning at the condition false edge.*

Definition 4.3.14 ***Candidate Subgraph (CS)****: CS is a set of nodes on the left and right paths. A set of nodes on the left path is called* ***LCS*** *and on the right path is called* ***RCS****.*



PR	PDC	LCS	RCS	ICEs	OCEs
A	CRD	{B}	{}	E→B	
A	G	{B,E}	{CRD, FS}	B→CRD, E→ B	B→CRD, CRD→A, E→B
B	B	{E}	{CRD, A}	PQ→A, A→CRD	A→CRD, E→G
B	G	{E}	{CRD, FS}	A→CRD	E→B, CRD→A
CRD	G	{FS}	{A, B, E}	PQ→A, E→B	A→CRD, E→B, B→CRD

Figure 4.6.: Combinations of PR-PDC pairs for the maximally folded CFG in Fig. 4.4

Definition 4.3.15 ***Contending Edge (CE)****: A CE is an edge that is neither on the left path nor on the right path, but a CS node is either its source or sink.*

Definition 4.3.16 ***Incoming Contending Edge (ICE)****: A contending edge that is an in-edge to a CS node.*

Definition 4.3.17 ***Outgoing Contending Edge (OCE)****: A contending edge that is an out-edge from a CS node.*

Examples: Fig. 4.6 shows the combinations of various PR-PDC pairs in the maximally folded CFG in Fig. 4.4. Certain properties of a maximally folded graph can be observed from this figure. First, for a given PR, multiple PDCs may exist, e.g. node A has two possible PDCs, CRD and G. A PR can be its own PDC, e.g. node B is its own PDC. The path from the left out-edge of B reaches itself through E, while the path from its right out-edge reaches B through CRD and A. A single edge could be both an ICE and an OCE for a given PR-PDC pair, e.g., for PR-PDC pair of A-G, edge (B $\rightarrow$ CRD) is both an ICE and an OCE.

Lemma 2 *In a maximally folded CFG, if a given PR is a structured selection/loop condition node, then there must exist an unstructured condition node in the selection/loop body of the PR.*

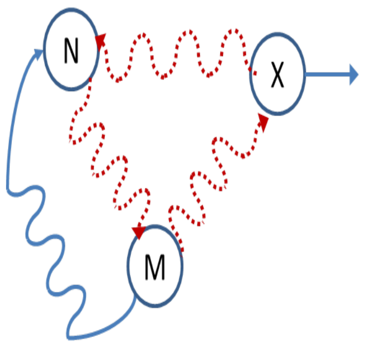
Proof Consider the selection/loop body for such a PR, which is a SESE region. If the PR is a structured selection condition, then without loss of generality, we can treat the PR, its IPDOM and the selection body as a sub-CFG that is maximally folded. Similarly, if the PR is a structured loop condition node, then without loss of generality, consider the loop body as a sub-CFG that is maximally folded. As this sub-CFG is not completely folded, by Lemma 1, it must contain an unstructured condition node. ■

Definition 4.3.18 ***Bounded loop condition node*** *: A loop condition node that has a loop path whose first edge is post-dominated by the last edge is called a bounded loop condition node. The corresponding path is called a* ***bounded loop path****.*

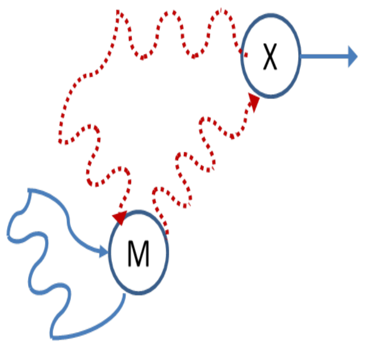
Lemma 3 *If a CFG contains a PR for which no PDC is present, then the PR must be a bounded loop condition node.*

Proof The IPDOM of a given condition node is its first strict post-dominator, i.e., each path from the condition node to the CFG exit node must pass through the IPDOM. If no out-edge of the condition node is post-dominated by itself, then there must be two non-overlapping simple paths from the condition node to its IPDOM, making the IPDOM a PDC. The only case in which a condition node can not have a PDC is if one of its out-edges is post-dominated by itself, i.e., the condition node must be a loop condition node where the last edge of the loop path post-dominates its first edge, meaning that the condition node is a bounded loop condition node.

■



(a) Bounded loop paths overlap



(b) Bounded loop paths do not overlap

Figure 4.7.: Bounded loop path ($\vec{P}$) has a bounded loop condition node (M) : $\vec{P}$ is represented by dotted lines

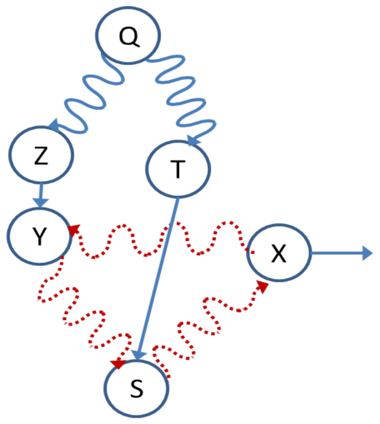
The next theorem, Theorem 4.3.2, plays an important role in our algorithm that converts an unstructured CFG into a structured one, as will be described in Section 4.5. This theorem guarantees that if a mechanism can generate a CFG where no PR-PDC pairs exist in the maximally folded CFG, then it must be a structured CFG.

Theorem 4.3.2 *In a maximally folded CFG, if there is a PR for which no PDC exists, then there must exist another PR in the CFG for which a PDC is present.*

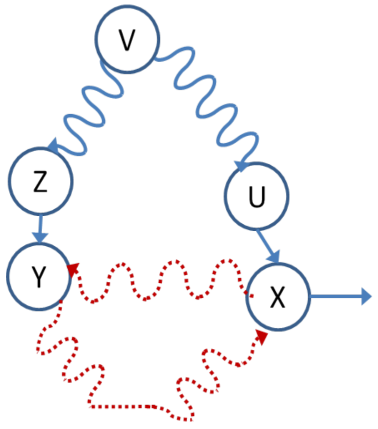
Proof Because there is no PDC for the given PR, the PR must be a bounded loop condition node, by Lemma 3. Let this PR be called X. Consider any bounded loop path, $\vec{P}$, of X. There exist the following two cases:

1. $\vec{P}$ **has at least one condition node** : Consider one such condition node M, which can be of the following two types:

   (a) **M is a bounded loop condition node (Fig. 4.7)**. Two cases arise: **Case i)** A bounded loop path of M has a node N in common with $\vec{P}$, as shown in Fig. 4.7a. Then, from one out-edge of M, a simple path reaches N



(a) Loop condition node has one in-edge



(b) Loop condition node has two in-edges

Figure 4.8.: Bounded loop path ($\vec{P}$) has no condition nodes : $\vec{P}$ is represented by dotted lines

through the bounded loop path of M. The other out-edge of M reaches N through $\vec{P}$. Hence, M-N form a PR-PDC pair. **Case ii)** No bounded loop path of M has a node in common with $\vec{P}$, as shown in Fig. 4.7b. Then, M-M form a PR-PDC pair, since one path from M reaches itself through one of its bounded loop path, while the other reaches M via $\vec{P}$.

(b) **M is not a bounded loop condition node** : A PDC must exist for M, by Lemma 3.

2. $\vec{P}$ **has no condition node (Fig. 4.8)** : Since $\vec{P}$ is not folded, and since it does not have any condition node, some node Y belonging to $\vec{P}$ must have two in-edges. Again, since $\vec{P}$ does not have a condition node, the source node of one of these edges, say Z, must not be on $\vec{P}$. The following two cases arise:

   (a) **X has only one in-edge (Fig. 4.8a)**: Note that a sub-path of $\vec{P}$ from Y (a single out-edge node) to X (a single in-edge node) is not folded. Hence, it must mean that there is some other node S on it, which has an in-edge from some node T that is not on $\vec{P}$. Let Q be the common nearest ancestor of Z and T in the dominator tree of the CFG. From Q, one simple path reaches S through $\vec{P}$ and through Z. The other reaches S without passing through $\vec{P}$, but through T. Hence, Q-S form a PR-PDC pair. Note that Z and T can be the same node, in which case Q will be that node as well. Also, it is possible that Q and Z are the same node, or Q and T are the same node.

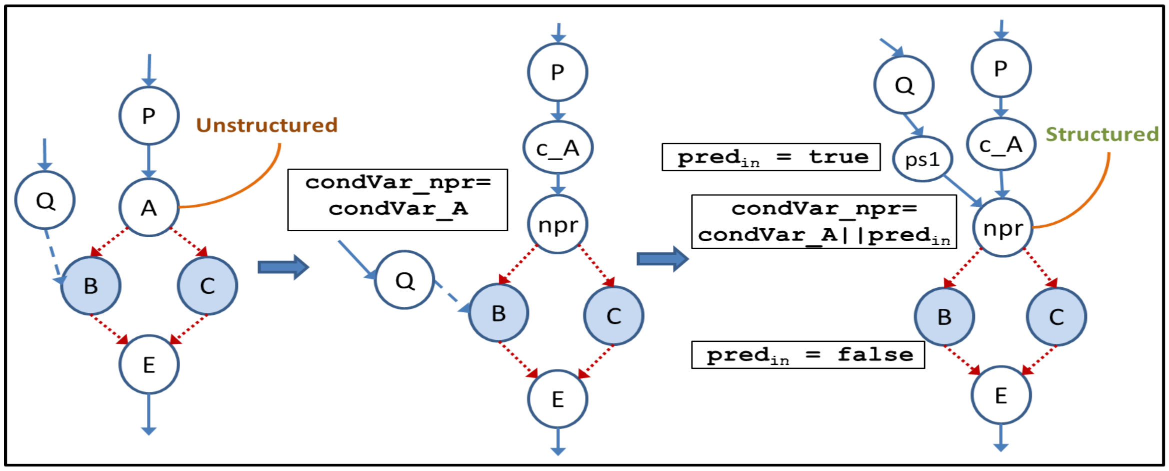
(b) **X has two in-edges (Fig. 4.8b)**: Let U be the predecessor of the in-edge of X that is not on $\vec{P}$. Let V be the common nearest ancestor of Z and U in the dominator tree of the CFG. From V, one simple path reaches X through U, without passing through $\vec{P}$. From V, another simple path reaches X though the edge Z$\rightarrow$Y, passing through $\vec{P}$. Hence, V-X form a PR-PDC pair. Z and U can be the same node, in which case V would be that node as well. Also, it is possible that V and Z are the same node, or V and U are the same node.

Therefore, some PR-PDC pair must exist in a maximally folded CFG that contains at least one condition node.

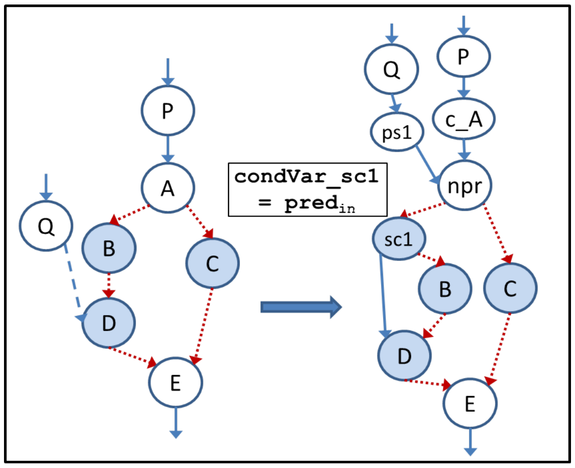
■

Theorem 4.3.3 *In a maximally folded CFG, the PR that is farthest away from the entry node of the CFG, and for which a PDC exists, is always an unstructured condition node.*

Proof We prove this theorem by contradiction. Let us assume that such a PR is a structured selection condition node or a structured loop condition node, and call it *PRO*. Then by Lemma 2, there must be an unstructured condition node inside



(a) Basic RICE technique



(b) RICE with short channelling

Figure 4.9.: RICE Transformation: Dotted edges and darkened nodes belong to left-/right paths. Dashed edge is an ICE.

the selection/loop body of PRO. Consider the selection/loop body of PRO, which is SESE, as the target sub-CFG. Now, in this sub-CFG, there must be a PR, say PRI, for which a PDC could be found, by Theorem 4.3.2.

Since PRO dominates its selection/loop body, any node inside the selection/loop body is farther away from the entry node of the CFG than PRO. Hence, PRI is farther away from the entry node, and also has a PDC, which is a contradiction. Therefore, a PR that is farthest away from the entry node, and for which a PDC exists, has to be an unstructured condition node. ■

Theorem 4.3.4 *In a maximally folded CFG, for a PR that is an unstructured condition node and has a PDC, there must be at least one CE.*

Proof We prove this theorem by contradiction. Let us assume that there is no CE for the PR-PDC pair with the PR being an unstructured condition node. Then, each node on the left and right path must have only one in-edge and one out-edge. Since the graph is maximally folded, on each path, there can only be one such node. However, the PR, the PDC and the left and right paths would now form the base selection pattern, which is a contradiction in a maximally folded CFG. Hence, for a PR which is an unstructured condition node and has a PDC, there must at least be one CE. ■

4.4 Structuring Transformations

This section presents two compiler transformations, *RICE* and *ROCE*, that operate on a maximally folded CFG and replace an unstructured PR with a structured selection condition. As a result, the selection body of the new PR becomes a structured region. Section 4.5 will explain how the repeated application of these transformations to each unstructured condition node creates a structured CFG.

Conventions: Each CFG node N contains code, *c_N*, followed by a jump at the end. If N is a condition node, the jump depends on a condition variable, *condVar_N*. Otherwise, the jump reaches the sole successor of the node. Without loss of generality, we assume that the CFGs are binary and *condVar_N* is a boolean variable. If *condVar_N* is true, the left target is executed, otherwise the right target is executed. Redirection of a node N containing an unconditional jump to some node X means that the jump target of N's jump instruction is changed to X. Similarly, if N were to contain a conditional jump with one of the targets being Y, then the redirection of edge N $\rightarrow$ Y to X means replacing the jump target containing Y by X.

4.4.1 Overview of RICE and ROCE

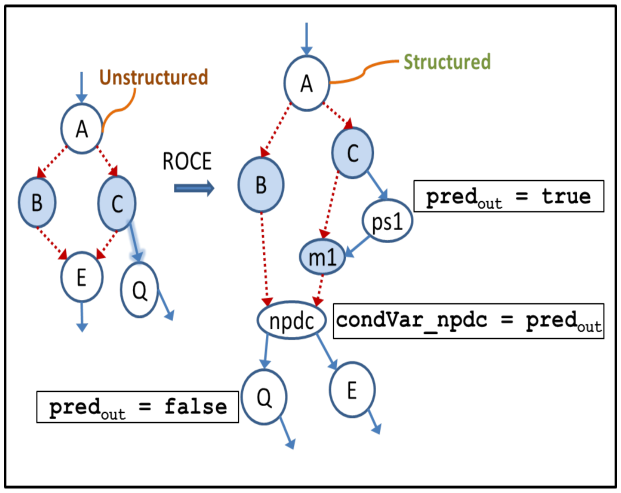
This section presents a high-level overview of the RICE and ROCE transformations. Sections 4.4.2, 4.4.3 describe the algorithmic details.

The key intuition in the RICE transformation is that it re-channels each ICE to come in through the PR, and then creates a path from the PR to the original sink node of the ICE. Fig. 4.9a shows a simple sub-CFG where for the PR-PDC pair of A-E, Q$\rightarrow$B is a single ICE. RICE splits the original PR into two nodes. The first node is *c_A* which contains the code in A followed by an unconditional jump to a new node, *npr*. The second node, *npr*, contains only the conditional jump in A. Next, to channel Q$\rightarrow$B via *npr*, RICE first redirects it to to a new *predicate setter* node, *ps1*, and directs *ps1* to *npr*. To ensure that B is reached from *npr* if the original ICE was taken, *cond_npr* must be modified. To that end, RICE creates a new predicate variable $pred_{in}$ and inserts a "$pred_{in}$=true" instruction in *ps1*. It then modifies *condVar_npr* to be *condVar_npr* $\vee$ $pred_{in}$. This way, if the original ICE were to execute, $pred_{in}$ will be true, and the control would reach B. Finally, RICE inserts instruction "$pred_{in}$=false" in the CFG entry node to initialize $pred_{in}$, and also in B, to reset it after the ICE is executed.

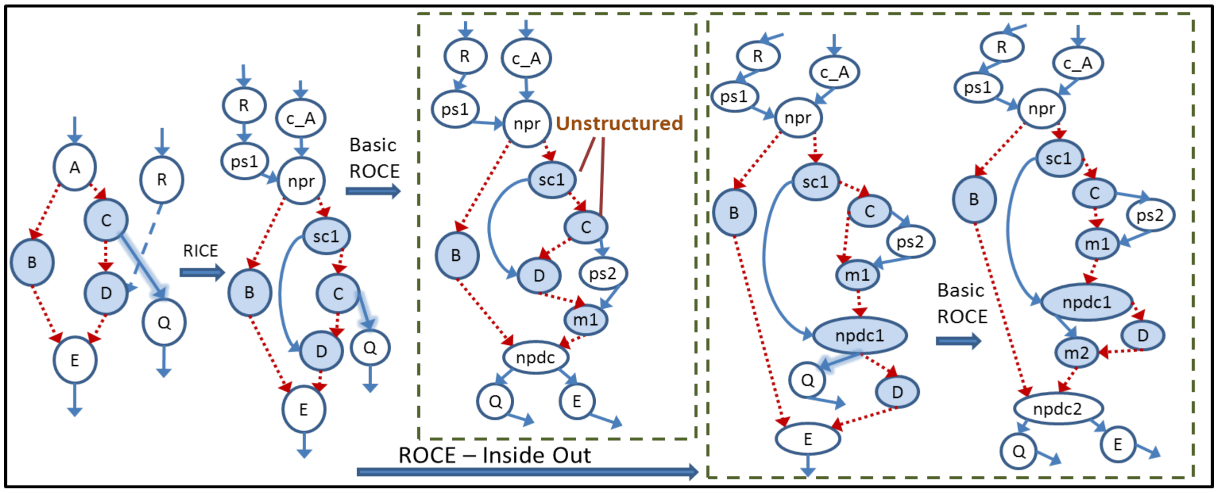
A difficulty arises when the ICE sink node is not the immediate successor of the PR. Consider Fig. 4.9b, where D has an ICE. In such a case, the above mechanism would incorrectly execute B in the changed ICE path. To resolve this issue, RICE inserts a condition node, called a *short-channel* node (*sc1*), as the PR's left child. RICE uses $pred_{in}$ as *condVar_sc1*, and sets the left target of *cond_sc1* to the original ICE sink (D), and the right target to the PR's original left child (B). Thus, if ICE is taken, B will be skipped, and control reaches D.

Analogously, the ROCE transformation re-channels an OCE to go out through the PDC node, and creates a path from the PDC to the original OCE sink node. Fig. 4.10a shows a sub-CFG where the edge C$\rightarrow$Q is the single OCE for the PR-PDC pair A-E. First, ROCE redirects the OCE to a new node, *ps1*. ROCE creates a new predicate variable, $pred_{out}$, and adds an instruction "$pred_{out}$=true" to indicate that the OCE is being taken inside *ps1*. Next, ROCE creates a *merge* node, *m1*, as the last node on the right path and directs *ps1* to *m1*. ROCE creates a new node, *npdc*, to act as the new PDC, and redirects the predecessors of E to *npdc*. ROCE sets *condVar_npdc* to be $pred_{out}$, with the left target being the sink node Q, and the right target being E. Finally, ROCE inserts "$pred_{out}$=false" in the CFG entry node to initialize $pred_{out}$, as well as in Q to reset it. Note that the sole purpose of the merge node is to restrict the in-degree of *npdc* to two.

A final issue may arise as ROCE is performed after RICE. RICE may insert condition nodes, i.e., short-channel nodes, on the left/right paths; ROCE must ensure their structuredness while removing OCEs. Consider the sub-CFG in Fig. 4.10b where an ICE and an OCE "cross", i.e., the ICE R$\rightarrow$D arrives at a node on the right path that is after the OCE source node C. The above described basic ROSE transformation



(a) Basic ROCE technique



(b) ROCE - performed inside-out, on crossing ICE and OCE

Figure 4.10.: ROCE Transformation: Dotted edges and darkened nodes belong to left/right paths. Dashed edge is an ICE, shadowed edge is an OCE. Nodes named sc* are *short-channel* nodes, m* are *merge* nodes, and ps* are *predSetter* nodes.

would create an edge from *ps1* to *m1*, rendering C and *sc1* unstructured. To overcome this problem, the ROCE transformation operates *inside out*. The key intuition here is that C→Q is now an OCE for the *inner* PR-PDC pair of *sc1*-D. Thus, the basic ROCE transformation is applied to this pair first, and then to the *outer* PR-PDC pair of *npr*-E.

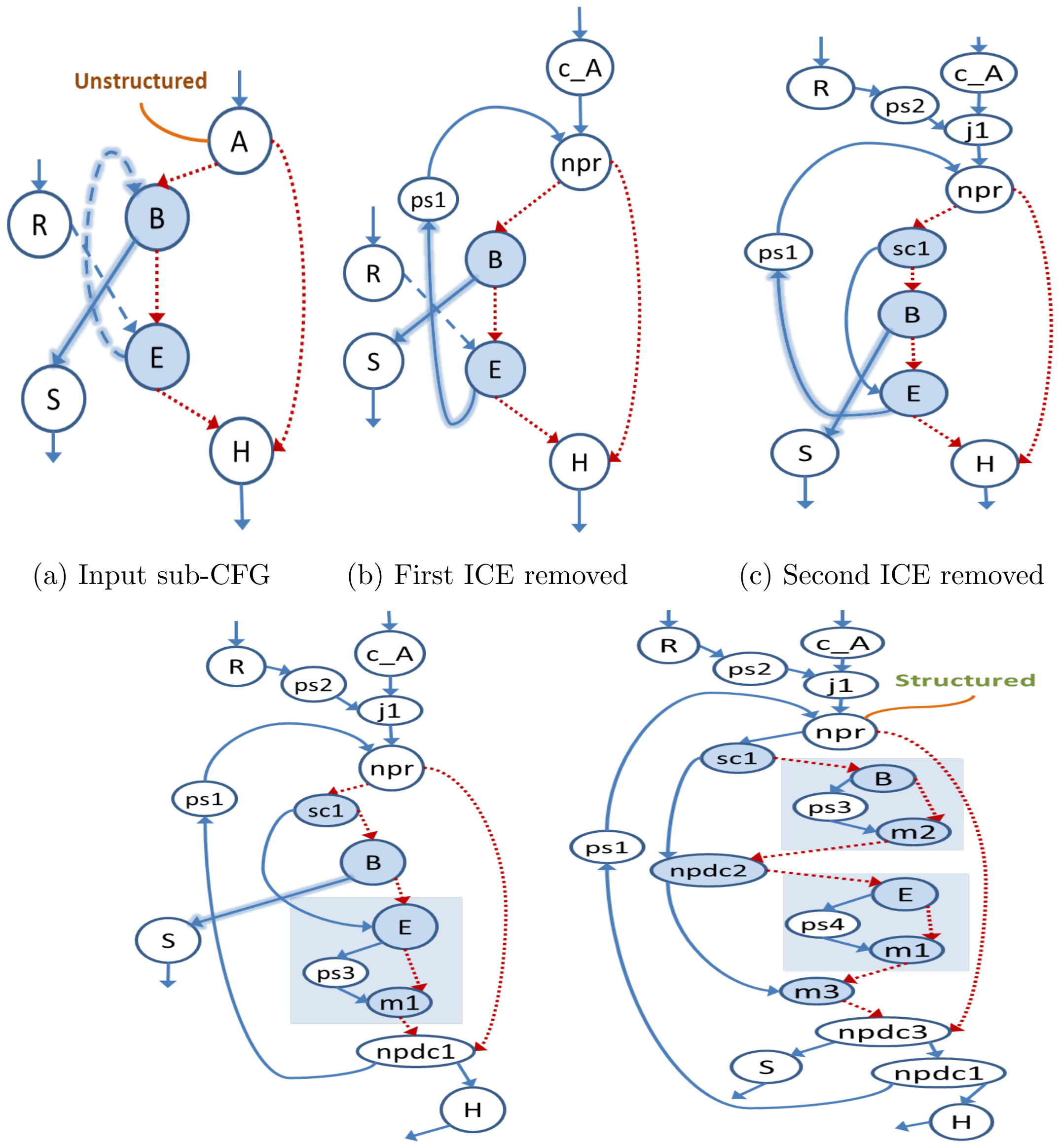
RICE and ROCE create a new PR-PDC pair wherein the respective regions between the first and last edges of both the left and right paths become SESE due to the removal of CEs. Therefore, the new PR becomes a structured selection condition node.

4.4.2 RICE - Removing Incoming Contending Edges

The RICE transformation removes all ICEs for a given PR-PDC pair. Algorithm 6 shows the RICE procedure. For a given PR-PDC pair, if there is at least one ICE, then the algorithm splits the PR node into two nodes, c_PR and *npr* (Lines 2-3).

The lists *LCS* and *RCS* contain nodes in the left and right paths respectively, in the order of their appearance, e.g., *LCS* for the CFG in Fig. 4.11a would be {B, E}. Lists *LCE* and *RCE* contain ICEs whose sink nodes are in *LCS* and *RCS*, respectively. They are arranged in the order of their sink nodes in the left/right paths, leading to a closer-to-PR-first order of ICE removal.

We now describe how the algorithm performs the high-level steps described in Section 4.4.1. RICE removes individual ICEs by traversing on both *LCE* and *RCE*. For each ICE, it inserts an instruction "$pred_{in}$=false" to initialize the new predicate variable at the beginning of the CFG entry node (Lines 12-13). It next creates a *predSetter* node with a "$pred_{in}$=true" assignment. It also inserts an instruction "$pred_{in}$=false" in the sink node of the ICE (Lines 21-22). RICE redirects the ICE to *predSetter* and directs *predSetter* to *npr*. RICE inserts a short-channel node, *sc*, if



(a) Input sub-CFG (b) First ICE removed (c) Second ICE removed

(d) First OCE removed (e) Second OCE removed

Figure 4.11.: Example of RICE and ROCE transformations: *A*-*H* is the PR-PDC pair. Dashed edges are ICEs, shadowed edges are OCEs. Dotted edges and darkened nodes belong to the left/right paths. Nodes named j* are join nodes, sc* are *short-channel* nodes, m* are *merge* nodes, and ps* are *predSetter* nodes. Shaded boxes indicate structured regions. After RICE and ROCE, the region between the first and last edges of the left path become SESE, making *npr* a structured condition node.

Algorithm 6: RICE - Removing Incoming Contending Edges in a given PR-PDC pair

```
Input: $PR$ - $PDC$ pair. $LCS$ is a list of all nodes on the left path, arranged
       in the order of their appearance. $RCS$ is symmetric. $LCE$ is a list
       containing the ICEs on the $LCS$ nodes, arranged in order of the
       appearance in the left path of their sink nodes. $RCE$ is symmetric.
       $OCEs$ is a set of all OCEs for the $PR$-$PDC$ pair
Output: $npr$ - The new PR node
$npr \leftarrow PR$;
if ($LCE.size() + RCE.size() > 0$) then
    $npr \leftarrow$ splitNode(PR); // separate code from jump
foreach $dir \in$ (left, right) do // do left and right paths
    if $dir$ = left then
        $curCS \leftarrow LCS$; $ICEs \leftarrow LCEs$
    else
        $curCS \leftarrow RCS$; $ICEs \leftarrow RCEs$
    foreach $edge \in ICEs$ do // edge loop
        $source \leftarrow edge$.getSource();
        $sink \leftarrow edge$.getSink();
        $pred_{in} \leftarrow$ createNewVariable();
        CFGEntryNode.insertAtBegin("$pred_{in} \leftarrow$ false");
        $predSetter \leftarrow$ createEmptyNode();
        if $npr.getNumPredecessors() = 2$ then
            $join \leftarrow$ createEmptyNode();
            foreach $predec \in npr.getPredecessors()$ do
                $predec$.replaceSuccessor($npr$, $join$);
        $source$.replaceSuccessor($sink$, $predSetter$);
        $predSetter$.setJumpTo($npr$);
        $predSetter$.insertAtBegin("$pred_{in} \leftarrow$ true");
        $sink$.insertAtBegin("$pred_{in} \leftarrow$ false");
        if ($OCE \leftarrow OCEs.find(edge)) \neq null$ then
            $OCE$.setSink($predSetter$)
        if $sink \neq curCS.getFirst()$ then // perform short chanelling
            $scn \leftarrow$ createEmptyNode();
            $npr$.replaceSuccessor($curCS$.getFirst(), $scn$);
            $scn$.setJumpTo($pred_{in}$, $sink$, $curCS$.getFirst());
            $curCS$.pushFront($scn$);
        if $dir$ = left then
            $npr$.setCondVar($npr$.getCondVar() $\vee$ $pred_{in}$);
        else
            $npr$.setCondVar($npr$.getCondVar() $\wedge \neg$ $pred_{in}$);
```

the sink node of the ICE is not the first node in *curCS* (Lines 25-29), and then marks *sc* as the first node of *curCS*. If the original ICE sink was in *LCS*, then RICE inserts an instruction to logically OR *condVar_npr* with $pred_{in}$, otherwise to logically AND it with NOT($pred_{in}$) (Lines 30-33).

A corner case arises if there are multiple ICEs on a left/right path. In such a case, after the first ICE is removed, *npr* would have two predecessors, and removal of the next ICE would add another predecessor to *npr*, breaking the maximum in-degree of two property. To resolve this issue, prior to the redirection of subsequent ICEs, RICE creates an empty node, *join*, to which it redirects the in-edges of *npr*, and directs *join* to *npr* (Lines 15-20). Another corner case arises if the ICE is also an OCE. As RICE redirects the ICE to *predSetter*, the sink node of the OCE is changed to be *predSetter* (Lines 23-24).

Correctness: The RICE algorithm maintains the execution order of the original CFG. We show that this holds after each ICE removal. If the ICE sink node was in *LCS*, and if the ICE is taken in the modified CFG at runtime, $pred_{in}$ will be set, and *cond_npr* would evaluate to true. Splitting the original PR serves the purpose of skipping *c_PR* when the ICE is taken. Control would reach the sink node of the original ICE either directly from *npr*, or through the *sc* node, since *condVar_sc* is set to $pred_{in}$ as well. On the other hand, if the ICE is not taken, then *condVar_npr* would simply carry the value of the condition variable of the original PR. Resetting $pred_{in}$ to false in the original ICE sink node ensures that if after executing the ICE, control were to return at *npr* through a non-ICE path, the original control flow is maintained. For this to occur, $pred_{in}$ must evaluate to false. Similarly, the control equivalence can also be shown for an ICE with a sink node in *RCS*.

Fig. 4.11 shows an example of the RICE transformation. In the input sub-CFG (Fig. 4.11a), *LCE* would contain {E → B, R → E}. Hence, in the first iteration of the edge loop (Line 9), edge E $\rightarrow$ B will be processed. The transformed CFG is shown in Fig. 4.11b. Note that no short-channel node was inserted for this node since B is the first node in *LCS*. In the second iteration, for the edge R $\rightarrow$ E, a short-channel node, *sc1*, is added (Fig. 4.11c). If the edge R $\rightarrow$ E was executed at the runtime in the original CFG, then in the new CFG, first the edge R $\rightarrow$ *ps2* will be executed. Then, $pred_{in}$ will be set to true in *ps2*, and the path *ps2* $\rightarrow$ *j1* $\rightarrow$ *npr* will be executed. As $pred_{in}$ is true, *cond_npr* will evaluate to true and the edge *npr* $\rightarrow$ *sc1* will be executed. Again, as $pred_{in}$ is true, *cond_sc1* will evaluate to true and the edge *sc1* $\rightarrow$ E will be executed, reaching the original ICE sink node E.

4.4.3 ROCE - Removing Outgoing Contending Edges

The ROCE transformation removes all OCEs for a given PR-PDC pair. Algorithm 7 describes the ROCE compiler transformation. The lists *LCS* and *RCS* are arranged to contain their nodes in the order of the left and right path, respectively. The lists *LCE* and *RCE* contain OCEs whose source nodes are in *LCS* and *RCS*, respectively. They are arranged in the reverse order of their source nodes in the left/right path, leading to a closer-to-PDC-first order of OCE removal.

We now describe how the high-level steps described in Section 4.4.1 are performed. For each OCE, ROCE creates a new predicate variable ($pred_{out}$) and an empty node *predSetter* (Lines 9-10). It inserts an initialization instruction "$pred_{out}$=false" in the entry node of the CFG, as well as in the OCE sink to reset $pred_{out}$. It inserts an assignment instruction "$pred_{out}$=true" in *predSetter* (Lines 11- 13).

The basic ROCE technique is applied next, in an *inside out* manner of the PR-PDC pairs. The corresponding `while` loop, called *in-out loop*, iterates until the OCE source is a part of *curCS*. This condition becomes untrue when the OCE source is the new PDC for *npr*. First, the basic ROCE technique creates new *npdc* and *merge*

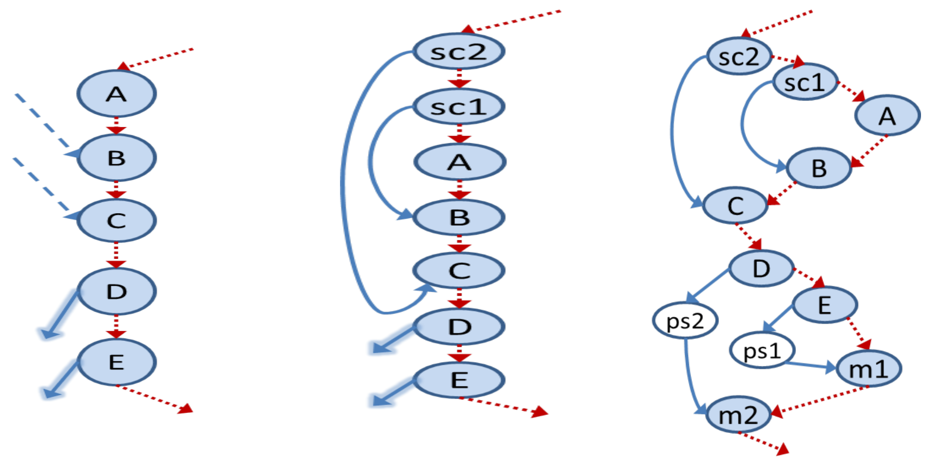
Algorithm 7: ROCE - Removing Outgoing Contending Edges in a given PR-PDC pair

```
Input: $PR$ - $PDC$ pair. $LCS$ is a list of all LCS nodes, arranged in order of
       the path from $PR$ to $PDC$, $RCS$ is symmetric. $LCE$ is a list
       containing the OCEs on $LCS$ nodes, arranged in reverse order of their
       source nodes in the $PR$ to $PDC$ path. $RCE$ is symmetric.
Output: $npdc$ - new PDC for the PR
foreach $dir \in$ (left, right) do // do left and right paths
    if $dir$ = left then
        $curCS \leftarrow LCS$; $OCEs \leftarrow LCEs$
    else
        $currentCS \leftarrow RCS$; $OCEs \leftarrow RCEs$
    foreach $edge \in OCEs$ do // edge loop
        $source \leftarrow edge$.getSource();
        $sink \leftarrow edge$.getSink();
        $pred_{out} \leftarrow$ createNewVariable();
        $predSetter \leftarrow$ createEmptyNode();
        CFGEntryNode.insertAtBegin("$pred_{out} \leftarrow$ false");
        $sink$.insertAtBegin("$pred_{out} \leftarrow$ false");
        $predSetter$.insertAtBegin("$pred_{out} \leftarrow$ true");
        $first \leftarrow$ true;
        while $curCS$.contains($source$) do // in-out loop
            $npdc \leftarrow$ createEmptyNode();
            $merge \leftarrow$ createEmptyNode();
            $lpdc \leftarrow curCS$.getNextTwoInEdgeNode($source$);
            if $lpdc$ = $null$ then // basic ROCE
                $lpdc \leftarrow PDC$;
                $curCS$.getLastNode().replaceSuccessor($lpdc$, $merge$);
                $curCS$.insertLast($merge$);
            else // inside-out ROCE
                $curCS$.getNodeBefore($lpdc$).replaceSuccessor($lpdc$, $merge$);
                $curCS$.insertBefore($lpdc$, $merge$);
                $curCS$.insertBefore($lpdc$, $npdc$);
            $lpdc$.redirectPredecessorsTo($npdc$);
            if $first$ = $true$ then
                $source$.replaceSuccessor($sink$, $predSetter$);
                $predSetter$.setJumpTo($merge$);
                $first \leftarrow$ false;
            else
                $source$.replaceSuccessor($sink$, $merge$);
            $npdc$.setJumpTo($pred_{out}$, $sink$, $lpdc$);
            $source \leftarrow npdc$;
            foldBetween($PR \rightarrow curCS$.getFirst(), $curCS$.getLast() $\rightarrow npdc$);
```

nodes (Lines 15-16). Next, it finds *lpdc*, which is the first node after the OCE source in *curCS* with two in-edges (Line 17). Presence of such a node means that there exists an inner PR-PDC pair; otherwise, the OCE belongs to only the outer PR-PDC pair (Lines 19-26). Next, ROCE redirects the predecessors of *lpdc* to *npdc*. ROCE redirects the OCE source edge is to *merge*, except for the first iteration of the in-out loop, where this redirection takes place via *predSetter* (Lines 28-33). ROCE inserts a jump instruction in *npdc*, and marks it as the new OCE source (Lines 33-34). Finally, ROCE performs a fold on the region between the first and last edges of the left/right path currently being operated. The folding removes two in-edge nodes that belong to a structured region.

Correctness: While removing each OCE, ROCE ensures that the original execution order of the control flow is retained. If an OCE is taken in the modified CFG, $pred_{out}$ is set to true, and then through one or more *npdc* nodes, the OCE sink is reached. This is ensured since all *npdc* nodes belonging to this OCE have $pred_{out}$ as their condition variable. On the other hand, if the OCE is not taken, then control traverses the original path to reach the original PDC, since all conditions in *npdc* nodes would evaluate to false.

Fig. 4.11c shows an example input sub-CFG to ROCE, on which the RICE transformation is already performed. The PR-PDC pair under consideration is *npr*-H. Here, *LCE* contains {E → *ps1*, B → S}. In the first iteration of the edge loop, (Line 6), the OCE E → *ps1* will be removed (Fig. 4.11d). The shaded box in the figure represents a structured region, which is folded. Let us call this folded node G. Note that no inside-out operation had to be performed here, since the left path from E to H did not contain a node with two in-edges. In the second iteration, the OCE B → S is removed (Fig. 4.11e). Since G had two in-edges, ROCE operates inside out, inserting *npdc2* and *npdc3*. After RICE and ROCE, the region between the first



(a) Example left path (b) RICE output (c) ROCE output

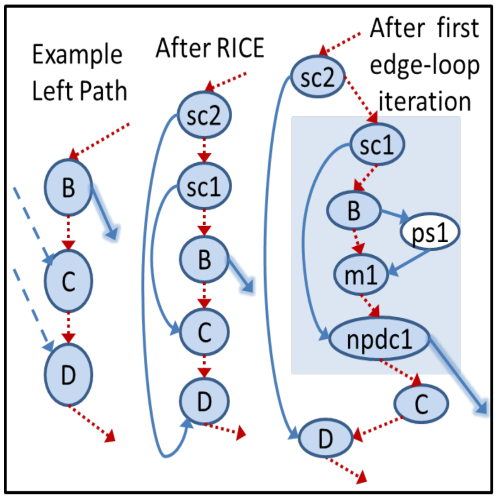
Figure 4.12.: Ensuring structuredness of all condition nodes on the left path when there is no crossing, i.,e., no OCE source is followed by an ICE sink: Dashed edges are ICEs, shadowed edges are OCEs. Dotted edges and darkened nodes belong to the left path. Nodes named sc* are *short-channel* nodes, m* are *merge* nodes, and ps* are *predSetter* nodes.

and last edges of the left path becomes SESE, making *npr* a structured selection condition.

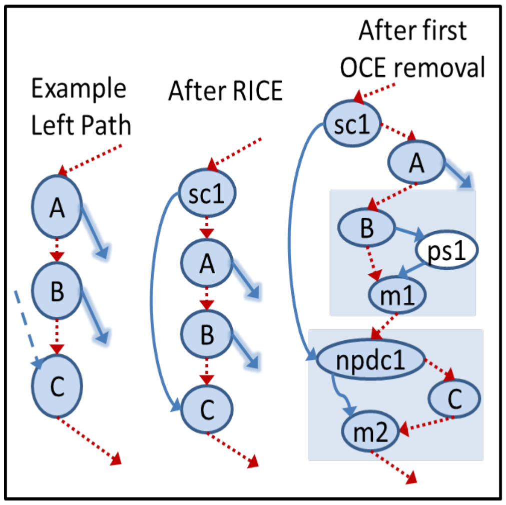
4.4.4 Ensuring Structuredness of the New PR-PDC Region

RICE and ROCE ensure that all condition nodes in the new PR-PDC SESE region are structured selection condition nodes. We start by showing that this is the case for a single ICE or OCE. Next, we extend it to multiple CEs of the same kind (ICEs/OCEs) on a given left/right path. Finally, we show the structuredness when multiple CEs exist on a path, with or without “crossing”, i.e., ICE sink node(s) are after OCE source node(s).

1. **Single CE**: Figure 4.9b shows the case of a single ICE. By construction, the inserted condition node, *sc1*, is structured, with D being its IPDOM. Note that, in general, on the subpath *sc1*→B→D of the left path, there could be additional



(a) Two ICEs after an OCE



(b) Two OCEs before an ICE

Figure 4.13.: Ensuring structuredness of all condition nodes on the left path when there is crossing, i.e., one or more OCE sources are followed by one or more ICE sinks: Dashed edges are ICEs, shadowed edges are OCEs. Dotted edges and darkened nodes belong to the left path. Nodes named sc* are *short-channel* nodes, m* are *merge* nodes, and ps* are *predSetter* nodes. Shaded boxes indicate structured regions.

nodes. However, since the path has a single ICE, all these nodes must have a single in-edge and out-edge. Removing a single OCE is essentially symmetric; ROCE constructs a *merge* node to be the IPDOM of the OCE source.

2. **Multiple ICEs/OCEs**: Since the RICE algorithm removes ICEs in closest-to-PR-first order, the short-channel node, its body, and IPDOM could be conceptually folded, producing the same structure as in the original CFG but with one ICE less. Consider the two ICEs in Fig. 4.12a. In the CFG generated by RICE (Fig. 4.12b), *sc1*, B, and *sc1*'s body can be folded into a single node, wherein *sc2* would be the single short-channel node, which is proven to be structured in the case above. A symmetric argument applies to ROCE, where the order of OCE removal is closest-to-PDC-first. An example can be seen in Fig. 4.12c.

3. **Non-crossing ICEs and OCEs**: The condition nodes introduced by RICE and ROCE, and their bodies, would be strictly sequential and would not overlap if no ICE sink follows any OCE source on the given left/right path. Such condition nodes, their bodies, and IPDOMs could be individually folded, indicating structuredness of all condition nodes on the path. Fig. 4.12a shows an example where no ICE sinks follow any OCE source; the sequential, non-overlapping nature of the condition nodes inserted by RICE and those by ROCE can be seen in Fig. 4.12c.

4. **Crossing ICEs and OCEs** : This is the case where the *inside out* operation of ROCE takes place. The base case here is of a single OCE source, followed by a single ICE sink, which was seen in Fig. 4.10b. From the output sub-CFG, it can be seen that all condition nodes on the right path, C, *npdc1*, and *sc1* became structured selection conditions. This result is generic; it would hold even when another node was present between *sc1*-C or between C-D. The following two cases suffice to see how structuredness is guaranteed when multiple CEs are involved:

   An OCE source is followed by two ICE sinks: Consider the example left path in Fig. 4.13a. After RICE, and the first iteration of the in-out loop, the OCE is removed from the inner PR-PDC pair of *sc1*-C. This turns *sc1* and B into structured condition nodes. Now, the shaded region in the figure can be folded into a single node to obtain a left path with a single ICE sink (D), following a single OCE source (*npdc1*), which is already proven to get structured.

   Two OCE sources are followed by an ICE sink: Consider the sub-CFG in Fig. 4.13b. After RICE, and the removal of the first OCE (on B), B and *npdc1* become structured condition nodes. The nodes, their IPDOMs, and their bodies, represented by shaded boxes, can be folded to obtain a left path with a single ICE sink (*npdc1*), preceded by a single OCE source (A), which is already proven to become structured.

The above two cases can be repeatedly applied to scenarios with multiple OCE sources followed by multiple ICE sinks, to prove the structuredness of all condition nodes on the left/right paths.

4.5 Unstructured to Structured CFG Conversion

This section describes an algorithm that uses RICE and ROCE transformations to convert an unstructured CFG into a structured CFG. Section 4.5.1 introduces a process that finds a PDC for a given PR. Section 4.5.2 describes the structuring algorithm and its time complexity.

4.5.1 Finding a PDC

The algorithm to convert an unstructured CFG into a structured one operates on PR-PDC pairs. It is imperative to find a PDC for a given PR. Recall that multiple PDCs may exist for a given PR. Recall also from Lemma 3 that some PR may not have a PDC, in which case the structuring algorithm would search for a PDC of a different PR. The presented algorithm (Algo. 8) returns the first PDC it finds, provided the PDC exists.

The algorithm follows descendants of the PR, starting from both the left and right children, in a breadth-first manner until it finds a common node, which will be the PDC. Unless the PR post-dominates one of the two paths, a PDC will be found (Lines 11-34).

After finding the PDC, the algorithm forms the respective *LCS* and *RCS*. *LeftLast* and *rightLast* are the left and right path predecessors of the *PDC*. Starting from

Algorithm 8: Finding a PDC for a given PR node

```
Input: $CFG$. $PR$ - PR node.
Output: $PDC$ - a PDC for the $PR$. $LCS$ and $RCS$ for the $PR$-$PDC$ pair.
searchQ ← getNewQueue();
leftSet ← ∅; rightSet ← ∅;
leftSet.insert($PR$.getLeftChild());
rightSet.insert($PR$.getRightChild());
searchQ.push($PR$.getLeftChild());
searchQ.push($PR$.getRightChild());
$PR$.getLeftChild().setParent($PR$);
$PR$.getRightChild().setParent($PR$);
$PDC$ ← null; $currDir$ ← null;
$leftLast$ ← null; $rightLast$ ← null;
while ($PDC = null \wedge$ !searchQ.isEmpty()) do
    curNode ← searchQ.pop();
    if ($node \in leftSet$) then
        currDir ← left
    else currDir ← right;
    if $node \neq PR$ then
        foreach ($child \in curNode.getChildren()$) do
            if $PDC = null$ then
                if ($currDir = left$) $\wedge$ ($child \in rightSet$) then
                    leftLast ← node;
                    rightLast ← $child$.getParent();
                    pdc ← $child$;
                else if ($currDir = right$) $\wedge$ ($child \in leftSet$) then
                    leftLast ← $child$.getParent();
                    rightLast ← node;
                    pdc ← $child$;
                else if ($child \notin leftSet$) $\wedge$ ($child \notin rightSet$) then
                    $child$.setParent(curNode);
                    if ($currDir = left$) then
                        leftSet.insert($child$)
                    else
                        rightSet.insert($child$)
                    if (!child.isIPDOMOf(PR)) then
                        searchQ.push($child$)
if $PDC \neq null$ then // generate LCS and RCS
    do
        $LCS$.pushFront(leftLast);
    while ( ( leftLast ← leftLast.getParent() ) $\neq PR$);
    do
        $RCS$.pushFront(rightLast);
    while ( ( rightLast ← rightLast.getParent() ) $\neq PR$);
```

these nodes, their parents are fetched until the *PR* is reached to form *LCS* and *RCS* (Lines 35-40).

4.5.2 Structuring Algorithm

The structuring algorithm (Algo. 9) is the main algorithm used for converting an unstructured CFG into a structured CFG. The *structuring loop* (Line 1) then executes until the CFG is structured, i.e., the number of nodes in the folded CFG is one.

Algorithm 9: Converting an unstructured CFG into a structured CFG

```
   Input: Arbitrary CFG
   Output: Structured CFG
1  while (CFG.size() > 1) do // structuring loop
2      fold(CFG); // maximally fold the CFG
3      prList ← getPRsInDeepestFirstOrder();
4      PDC ← null; PR ← null;
5      foreach (PR ∈ prList) do
6          if (PDC ← findPDC(CFG, PR, LCS, RCS)) ≠ null) then
7              break
8      (LeftICEs, RightICEs, LeftOCEs, RightOCEs) ← getCEs(CFG, PR, PDC,
       LCS, RCS);
9      npr ← RICE(PR, PDC, LCS, RCS, LeftICEs, RightICEs);
10     ROCE(npr, PDC, LCS, RCS, LeftOCEs, RightOCEs);
```

Each iteration of the structuring loop starts by calling *fold* function, which maximally folds the *CFG*. Folding is a conceptual operation; it does not actually modify the CFG. Next, the algorithm forms a list of PR nodes (*prList*). The list is sorted in the descending order of the depths of PR nodes from the CFG entry node. Then the algorithm traverses prList until it finds a PR-PDC pair. By Theorem 4.3.2, since the CFG is not completely folded, there must be a PR for which some PDC is present. Algo. 8 is used to find the PDC and the corresponding *LCS* and *RCS*. By Theorem 4.3.3, such a PR must be an unstructured condition node. There must be CEs for this PR-PDC pair, by Theorem 4.3.4. Function *getCEs* generates lists containing

ICEs and OCEs. The algorithm then performs the RICE transformation, which returns the new PR (*npr*). Consequently, with *npr* and *PDC*, the algorithm performs the ROCE transformation.

4.5.3 Termination Proof and Time Complexity of the Structuring Algorithm

An invariant of the structuring algorithm (Algo. 9) is the removal of at least one unstructured condition in each iteration of the structuring loop, owing to the conversion into a structured selection condition. After the RICE-ROCE transformations are applied to a given PR-PDC pair and the corresponding CS, a new PR-PDC pair is generated. The original PR is no longer a condition node. The new PR is a structured selection condition node, and its body is a structured region (Section 4.4.4). Therefore, the new PR-PDC SESE gets folded after the RICE-ROCE transformations are applied. All condition nodes added during the RICE transformation, i.e., the short-channel nodes, belong to the SESE region of the new PR-PDC pair. All condition nodes inserted during each OCE removal, i.e., *npdc* nodes, belong to this SESE region as well, except the final *npdc* node. The number of *npdc* nodes added outside of this SESE region is equal to the number of OCEs for the original PR-PDC pair. Let us assume that there were 'm' condition nodes that were the sources of the OCEs. Once the folding following the RICE-ROCE transformations has taken place, at least $m+1$ condition nodes would be folded, m for the sources of the OCEs, and 1 for the PR. And, at most m new condition nodes would remain in the CFG, the ones added by the ROCE transformation. Hence, overall, at least one condition node is reduced in each iteration of the structuring loop. Thus, the structuring loop can execute for at most n iterations. Symmetrically, it can also be shown that at least one node with two in-edges is reduced after each iteration.

Another invariant of the structuring algorithm is that the number of edges in the CFG reduces in each iteration. Let us define **outer edges** as those edges where neither sink nor source nodes are in the PR's selection body, after RICE and ROCE. For each ICE removal, the RICE transformation adds at most one outer edge, from node *join* to *npr*. The original ICE is merely redirected to *join* or *npr*. The ROCE transformation adds no outer edge; the OCE is redirected so that *npdc* becomes its new source. After each structuring loop iteration, at least two edges get folded, corresponding to the first edges of the left and right paths. Additionally, the out-edge of each ICE sink is folded as well, compensating for one outer edge RICE adds. Therefore, after each iteration of the structuring loop, the number of edges in the CFG reduces.

The algorithmic complexity of the structuring algorithm is as follows: The RICE and ROCE transformations have no impact on single in-edge, single out-edge nodes. In a CFG of n nodes, there can at most be $2n$ edges. The structuring loop can execute at most n iterations, which is a conservative over-estimation since it assumes all n nodes to be unstructured condition nodes. The *fold* function operates on each edge in the CFG. Each iteration of the structuring loop reduces the number of edges in the CFG. Therefore, the complexity of *fold*, called in Line 2, aggregated over all iterations of the structuring loop is bounded by the number of edges in the CFG, i.e., $\mathcal{O}(n)$. The *getPRsInDeepestFirstOrder* function is $\mathcal{O}(n)$, as it requires a BFS-style traversal of the CFG. Therefore, across all iterations of the structuring loop, its complexity is $\mathcal{O}(n^2)$. Similarly, the *findPDC* function, across all iterations of the enclosing *for* loop around it, traverses the entire CFG and is therefore $\mathcal{O}(n)$. Let k_i represent the number of nodes in the CS for the PR-PDC pair in the i^{th} iteration of the structuring loop. Then, the edge loop in the RICE transformation (Algo. 6), in the i^{th} iteration would execute for at most k_i times, which is the worst case behavior assuming that each CS node is an ICE sink. Summing up over all iterations of the structuring loop, the edge loop in the RICE transformation would execute at most $\sum_{i=1}^{n} k_i$ iterations. Since every iteration of the structuring loop reduces the number of two in-edge nodes (Section 4.5.1), $k_i < n$. Hence, the worst-case complexity of RICE aggregated across all iterations of the structuring loop is $\mathcal{O}(n^2)$, conservatively assuming that each CS node has an ICE. Similarly, the ROCE edge loop would execute at most k_i times, for the i^{th} iteration of the structuring loop, assuming that each CS node has an OCE. The maximum number of times the ROCE in-out loop can execute is limited by the number of ICEs for the PR-PDC pair. Hence, it can execute at most k_i times, assuming that each CS node has an ICE. Fold (Algo. 7, Line 18) called inside ROCE is $\mathcal{O}(k_i)$ as well. Hence, the time complexity of ROCE called in the i^{th} iteration is $\mathcal{O}(k_i^2)$. Summed up over all iterations of the structuring loop, the complexity of the structuring algorithm is $\sum_{i=1}^{n} \mathcal{O}(k_i^2)$, which in the worst-case is $\mathcal{O}(n^3)$, conservatively treating each CS node to have an ICE as well as an OCE.

4.6 Avoiding Exponential Blowup

This section describes how the presented structuring algorithm can convert an irreducible CFG into a reducible CFG without incurring exponential code expansion, known as *exponential code blowup* [107].

4.6.1 Removing Irreducibility with Node Splitting

The known Node Splitting technique [106, 112] duplicates nodes in the CFG so as to remove irreducibility. Consider the irreducible CFG in Fig. 4.14, which results from the repeated T1/T2 application to the original CFG in Fig. 4.3. Node Splitting operates on any node with at least two predecessors. This node is duplicated, and

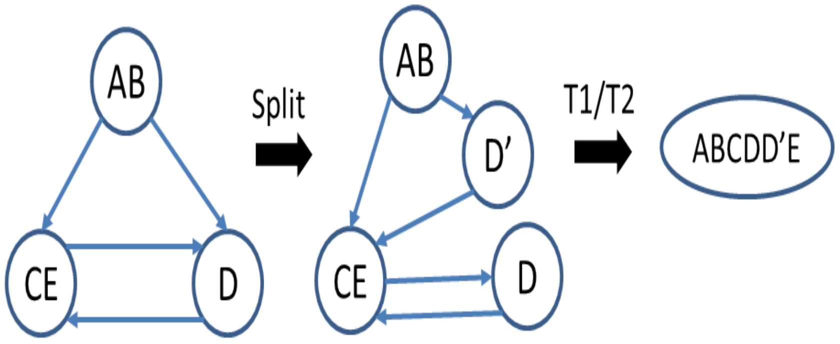


Figure 4.14.: Node Splitting: D' is a copy of D

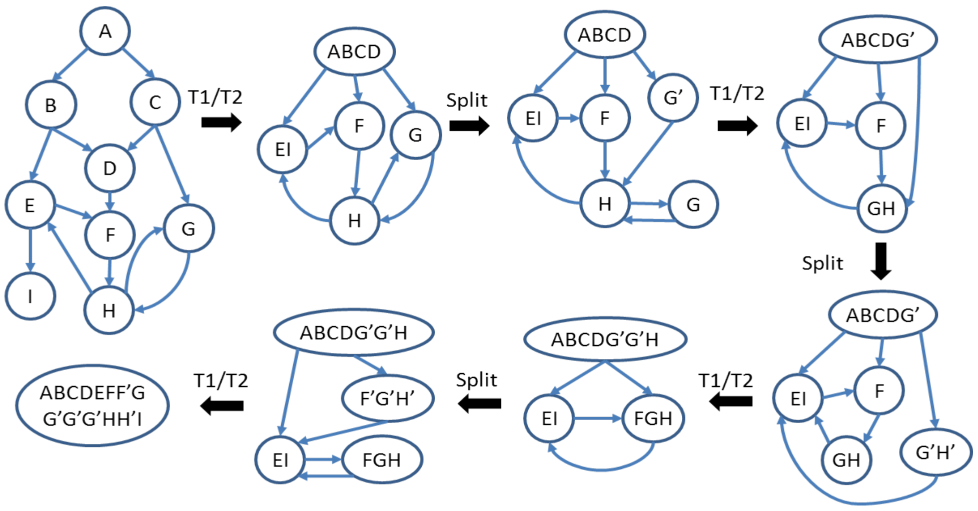


Figure 4.15.: Blowup caused by Node Splitting: Four copies of G, two copies each of F and H

each predecessor keeps a copy of this node. The out-edges of each copy are directed to the same nodes as in the original graph. In Fig. 4.14, splitting just one node results in a reducible CFG.

However, such node duplication can cause exponential code size blowup. Consider Fig. 4.15, which contains overlapping irreducible loops (Loop1: H→E→F →H and Loop2: H→G→H). In such a scenario, the CFG generated by Node Splitting would contain four copies of G, and two copies each of nodes F and H. Carter et. al. [107] proved that reducible CFGs generated by Node Splitting can get exponentially larger.

4.6.2 How Does the Structuring Algorithm Avoid Exponential Blowup?

The proposed CFG structuring mechanism transforms an unstructured CFG into a structured CFG. By Corollary 1, an irreducible CFG must be unstructured, whereas a structured CFG has to be reducible, by Theorem 4.3.1. Hence, the structuring mechanism also results in a conversion of an irreducible CFG into a reducible CFG. During this process, code size increase is limited to a polynomial function of the number of nodes in the CFG. This is possible since the mechanism requires no code duplication. The additionally inserted code comprises i) predicate variable assignments, ii) *OR* and *AND* instructions that determine condition variables for PR nodes and iii) additional jumps, such as during short-channelling.

Each iteration of the edge loop in the RICE transformation inserts at most eight instructions. It adds three instructions for predicate assignments/initialization, one for changing (logical OR/AND) the condition of the PR, and one for the jump in the short-channel node. Two more jump instructions are added at the end of the *predSetter* and *join* nodes. Furthermore, one instruction may be added during each call to the RICE transformation; this instruction jumps from *c_PR* to *npr*. As described in Section 4.5.3, the RICE transformation is executed at most n times, and the number of edge loop iterations is bounded by n. Therefore, the total number of RICE-added instructions is bounded by $8n^2$.

Let k_i be the number of nodes in the CS for the PR-PDC pair in the i^{th} iteration of the structuring loop. The number of edge loop iterations in the corresponding ROCE call(Algo. 7, Line 6) is bounded by k_i, as described in Section 4.5.3. In each edge loop iteration, at least six instructions are added. These include three initialization/assignments to the predicate variable and jumps at the end of *merge*, *predSetter* and the last *npdc* nodes. In the in-out loop, each iteration inserts two instructions, namely, the jumps at the end of *npdc* and *merge* nodes. Since the in-out loop executes at most k_i iterations, the number of instructions inserted during the inside out ROCE operation is $2k_i$. Summed up over all iterations of the structuring loop, the total ROCE-inserted instruction count is bounded by $\sum_{i=1}^{n}(6k_i + (2k_i)^2) \leq 6n^2 + 4n^3$. Hence, code expansion is bounded by a degree-3 polynomial of n in the worst case, which assumes each CFG node has an ICE and an OCE.

4.7 Related work

The term "structuredness" has often been used ambiguously, e.g., some [113–115] consider `switch` statements to be a part of the structured clauses, while others [87, 89] do not. Erosa et. al. [38] consider programs structured as long as there are no `goto`s. In contrast to the source-level definitions, the precise, CFG-level formalizations presented in this paper eliminate the ambiguity.

Structured programming originated with a focus on the ease of program understanding and maintenance. Much debate ensued; Dijkstra suggested that `goto` statements make this difficult [116], while Knuth [117] pointed scenarios where `goto` statements make programs easier to understand. Many "structuring" proposals aimed at removing them [38–40, 102, 103]. However, all of them operate on the program source, and not on the CFG, and hence can not be used for CFG analysis.

Decompilers convert binaries into high-level program source codes. Many decompilers therefore propose techniques to structure CFGs [115, 118, 119]. However, the term "structuring" is loosely interpreted; certain `goto`s may still remain in the program after decompilation, and removal of irreducibility is not guaranteed.

Generic structuring approaches convert an input program into one that contains only the base structured constructs/patterns. They can be categorized into two classes. The first class works on program source codes. Ammarguellat [36] presents a normalization-based technique. Zhang et. al. [37] present a structuring mechanism that employs both added predicates and code duplications to obtain structuring. However, these approaches are language-syntax dependent, and can not be extended to generic CFGs. The second class works directly on the CFGs. Williams [89] identifies unstructuredness using five base patterns and in follow-up work [104] proposes an algorithm to convert unstructured CFGs into structured ones. Similarly, Oulsnam [120] presented six base patterns that represent unstructuring, and proposed six transformations to replace these patterns with structured ones. A more generic mechanism was proposed in following work [105]. These mechanisms fail to describe how a given large CFG can be decomposed into the base structured patterns. Therefore, despite being CFG-based, they require sophisticated pattern matching to identify unstructuredness.

All the above approaches, except [38, 39] perform Node Splitting to remove irreducibility, and hence are susceptible to exponential blowup. These two approaches insert predicates in the code, however, no time complexity or code size expansion analysis for these methods has been presented. DJ graphs [101] is one approach that has proposed optimizations to reduce the amount of duplication incurred during Node Splitting.

4.8 Experiments

This sections compares the new structuring algorithm against classical Node Splitting in terms of the resulting code expansion. It also shows how obfuscation by irreducibility-insertion is rendered ineffective by the structuring algorithm. We show that irreducibility-inserted obfuscated CFGs are susceptible to automatic compiler analyses after the application of the structuring algorithm.

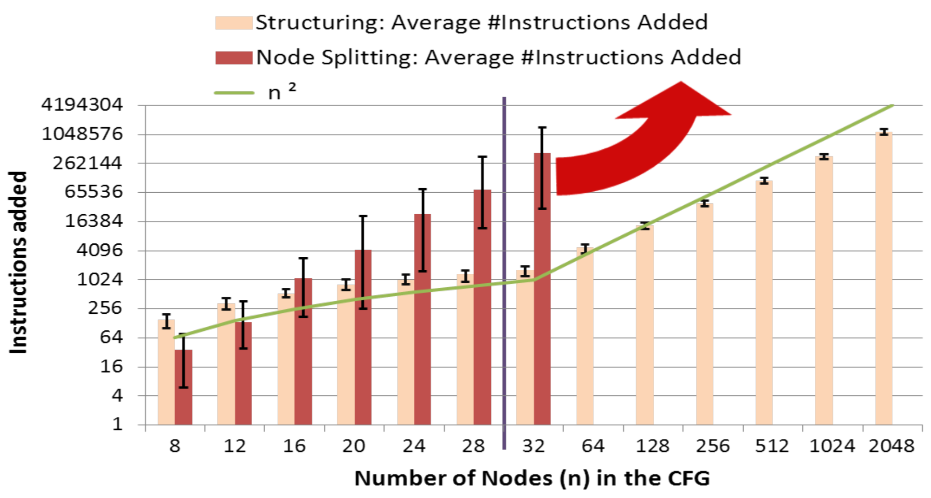


Figure 4.16.: Understanding code size expansion: The vertical line indicates switching of the X-axis to a logarithmic scale. Node Splitting generated CFGs are much larger. The size of 32-node CFGs after Node Splitting can be larger than the size of 1024-node structured CFGs. Node Splitting did not finish in 400 hours for CFGs of 64 nodes and beyond.

4.8.1 Setup

We performed our experiments with the LLVM [121] compiler. We implemented the structuring algorithm as an LLVM compiler (version 3.8) pass. Prior to performing structuring, this pass converts the CFG into an equivalent CFG where the in-degree and out-degree of each node is 2. All our experiments were conducted on a workstation with Intel i7-6700K 8-core 4GHz hyperthreaded processor. The machine hosted a 16GB RAM, running Ubuntu 14.04.

4.8.2 Code Size Expansion on Random CFGs

We present an experimental comparison of the code expansion resulting from the presented structuring algorithm and the Node Splitting approach. We use pseudo-randomly generated irreducible CFGs. These CFGs are generated as follows: all nodes except the exit node in these CFGs are condition nodes, and each node except the entry node has two in-edges. Each condition node is an unstructured condition. There is no code in these nodes, apart from a `load` instruction for fetching the condition variable, and a `jump` instruction. There are no self loops. It is fairly easy to show that all such CFGs with four or more nodes are irreducible (see Appendix for details). Although these CFGs have no nodes with single in-edge and single out-edge, they represent generic behavior. Single in/out-edges have no effect on code size expansion obtained by the structuring algorithm. On the other hand, Node Splitting technique would see larger code sizes owing to the duplication, if such nodes were to exist. Therefore, the comparison results shown here are conservative.

Fig. 4.16 shows the added instruction count for different CFG sizes. The bars show added instruction counts averaged over n different CFGs, each containing n nodes. It can be clearly seen that the code size growth in the structuring mechanism does not blow up, and is bounded by n^2 as the CFGs get larger. Error bars indicate that the variation in code growth for the different CFGs is small. For Node Splitting, the code size growth is rapid; the size of a 32-node CFG after Node Splitting can be larger than the size of a 1024-node structured CFG. The error bars show higher code size variation in Node Splitting. The rapid growth in CFG sizes during Node Splitting makes their generation and analysis extremely slow. Our experiments with CFGs of size 64 did not finish within 400 hours, at which point we terminated the experiments. Fig. 4.16 therefore lacks Node Splitting results for larger CFG sizes.

4.8.3 Unstructuredness in Compiler-generated CFGs

Table 4.1 shows unstructuredness in the C implementation [122] of the NAS parallel benchmarks [123]. Although few functions in these applications have source-level unstructuredness, i.e., the unstructuredness caused by constructs such as `break`,

Table 4.1.: Unstructuredness of CFGs in NAS Benchmarks: Even the functions with structured source code can have unstructuredness in the compiler front-end generated CFG. Optimized CFGs are more likely to possess unstructuredness.

Benchmark	**#Functions**	**#Functions with unstructured constructs**	**#Functions with unstructured CFGs generated from front-end**	**#Functions with unstructured CFGs after -O3**
CG	16	1	3	2
FT	25	2	3	5
EP	10	1	1	1
MG	26	0	2	7
LU	27	0	3	2
SP	31	0	4	4
IS	13	2	1	1
BT	34	0	1	4

`goto`, `continue` etc., the compiler front-end generated CFGs have higher occurrences of unstructuredness. Applications MG, LU, SP, and BP have no source-level unstructuredness; yet, the front-end generated CFGs for some functions in these applications are unstructured. Program source-level structuring techniques [36–40,102,103] cannot cater to such unstructuredness. Furthermore, optimized CFGs can be seen to contain more functions with unstructured CFGs. Benchmarks CG and LU are exceptions; the LLVM compiler inlined unstructured functions in the optimized versions, leading to a reduction in the total number of unstructured functions.

4.8.4 Obfuscation by Irreducibility Insertion

Introducing irreducibility in a CFG is a well-known obfuscation technique [107, 109, 110]. Fig. 4.17 shows an example of this technique. Prior to each loop entry node in the CFG, the technique inserts a node with an opaque predicate (condition variable), such as `a*(a+1) % 2 == 0`, and the condition false out-edge of the node is connected to some node in the loop body, creating an additional loop entry. The outcome of such opaque predicates is known at the obfuscation time, e.g., the above condition is known to return `true` for any integer 'a', meaning that the additional entry edge in the loop will never be executed at runtime. This assures functionally correct code execution. However, a static analyzer cannot understand that the dummy loop entry edge would never be executed at runtime, and hence fails to remove such loop entry edges. Irreducible CFGs are known to restrict compiler analyses. Node Splitting can deal with them but may lead to exponential code blowups. Compilers therefore do not perform Node Splitting. Secondly, for high-level languages such as Java, this technique disables decompilation. Although Java bytecode can represent irreducibility, due to the lack of `goto` statements, Java source code can not express irreducible control.

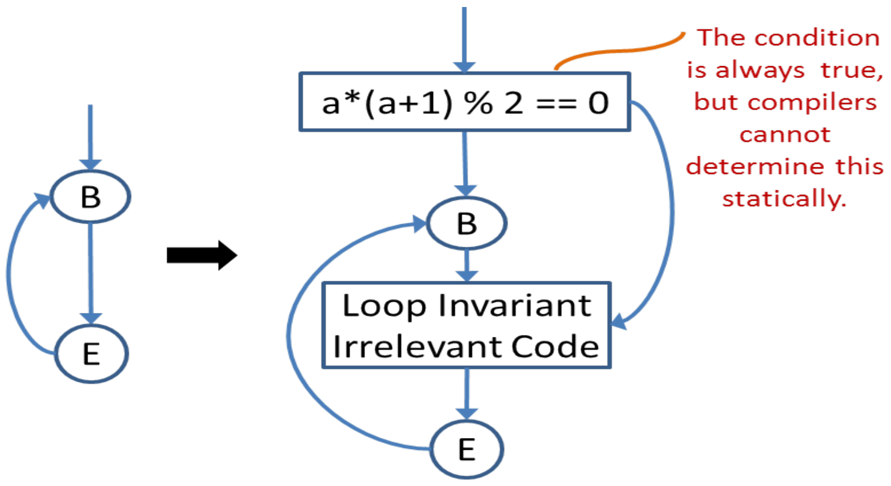

B
E
a*(a+1) % 2 == 0
The condition is always true, but compilers cannot determine this statically.
B
Loop Invariant Irrelevant Code
E

Figure 4.17.: Obfuscation via Irreducibility Insertion

To understand how such irreducibility insertion can impact compiler analyses, we wrote an obfuscation pass that operates on the LLVM IR. The pass inserts irreducibility using the above technique. In cases of nested loops, an irreducible edge is added from outside the outermost loop into the innermost loop body. Apart from irreducibility insertion, the pass inserts loop invariant irrelevant code into the loop body, which is a common obfuscation technique [109, 110]. In our implementation, this irrelevant code reads a dummy array and performs some computation on it. The generated result is used as an input to a `math` library call. Due to this call, compilers fail to understand this is indeed a dead code. Our obfuscator implementation operates on the front-end (Clang) emitted LLVM IR and generates CFGs with irreducible loops.

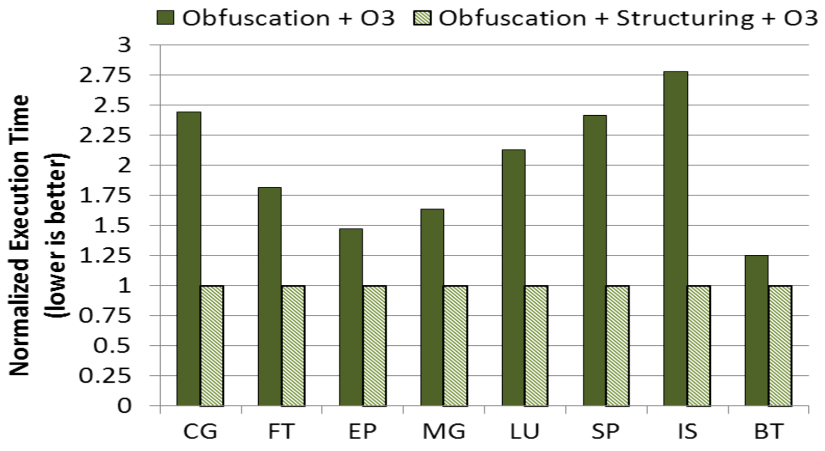


Figure 4.18.: Measuring analyzability of obfuscated CFGs achieved by structuring: The structured versions run significantly faster (1.92x geometric mean) compared to the unstructured (irreducible) counterparts.

Fig. 4.18 compares execution times of obfuscated programs from the entire NAS benchmark suite. The CFGs were obfuscated, and then were compiled with and without structuring, followed by LLVM -O3. The structured versions obtain better performance in all programs, with a geometric mean speedup of 1.92x. The key reason for this result is the inability of standard compilers, such as LLVM, to optimize irreducible CFGs. Many optimizations, especially the loop-based ones, fail to work on irreducible CFGs. After our new structuring pass is applied, all loops become reducible, and the traditional compiler passes become applicable. Apart from the

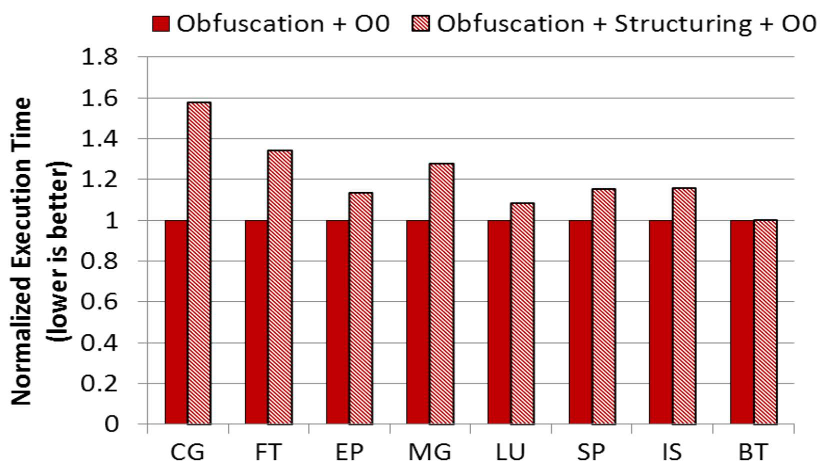


Figure 4.19.: Measuring overheads introduced by structuring: Structuring causes an overhead of 20% (geometric mean).

loop invariant code motion of the irrelevant code added by obfuscation, we attribute the better performance of structured versions to other optimizations such as loop vectorization, unrolling, and strength reduction. The variation in the amount of benefits achieved is also attributed to the factor of execution time each application spends in the obfuscated loops. With higher execution times of such loops, the benefits increase.

The structuring algorithm inserts additional variables and bran-ches into the program, which is bound to introduce overheads in the execution. To measure this overhead, we compare the execution times of obfuscated NAS applications with and without structuring, along with LLVM -O0. Using -O0 disables most of the compiler optimizations, and the difference between the structured and unstructured codes' execution times represents the overheads of the structuring mechanism, as shown in Fig. 4.19. The structured program versions execute 20% slower. We attribute this slowdown to the additional register pressure and extra branches. Note however that this overhead is offset by the compiler optimizations, as is evident from Fig. 4.18.

Listing 4.1: Synthetic GPU Code Example

```
void __device__ __noinline__ devFunc(float a,
        float* x, float* y, int c) {
  //this loop simply increases the computation
  for(int i=0; i<98; i++) {
    y[threadIdx.x] = y[threadIdx.x] + 4;
    if(y[threadIdx.x] < 9) {
      y[threadIdx.x] *= y[threadIdx.x];
    }
    y[threadIdx.x] /= 3;
  }
}

 __global__ void kernel(float a, float* x,
          float* y, int c) {
  //this loop simply increases the computation
  for(int i =0 ; i <10; i++) {
    if(threadIdx.x < 16) {
      devFunc(a,x,y,c);
      if(threadIdx.x  < 4) {
        goto b;
      }
    } else {
      b: devFunc(a,x,y,c+2);
    }
    y[threadIdx.x] = a * x[threadIdx.x];

  }


}
```

General-purpose compilers may reduce this overhead by choosing to execute the structuring pass on only those functions that contain irreducibility.

4.8.5 Case Study on a GPU Program

Listing 4.1 shows a synthetic example GPU code that benefits from the structuring pass. Threads 0 to 15 in a warp execute the `if` path, while threads 16 to 31 execute the `else` path. Both these paths execute *devFunc*, whose execution across these two paths gets serialized owing to the SIMD execution on GPUs. Also, threads 0-3 jump from the `if` path to the `else` path. In the state-of-the-art compilation of this kernel, each warp *devFunc* thrice – two times from the `if` path, and once times from the `else` path. After structuring, this number goes down to two. We observed a speedup of 1.47x on an NVIDIA Tesla K40 GPU with the structured version over the original, unstructured code version.

4.9 Chapter Takeaways

This chapter has formalized the notion of structured CFGs by presenting formal definitions for the three base structured patterns that compose a structured CFG. It introduced a concept of folding, which replaces base structured patterns with single nodes, and helps determine whether or not a given CFG is structured. A new structuring algorithm was presented that converts arbitrary unstructured CFGs into structured ones. The algorithm repeatedly applies two novel transformations that remove unstructuredness by redirecting the control flow and inserting additional predicate variables. The time complexity of this algorithm was studied along with the caused code expansion. The chapter proved that the algorithm limits code expansion to a polynomial function of the number of nodes in the CFG, and presented experimental evidence. This result is important since it overcomes the issue of exponential code blowup of irreducible CFGs in previous Node Splitting-based techniques, which resort to code duplication. The chapter showed that the irreducibility insertion technique employed by software obfuscators is rendered ineffective by the proposed structuring algorithm. Furthermore, this structuring algorithm is directly applicable to various compiler passes [91–97] that either give up and resort to Node Splitting when faced with irreducibility, or need special implementations [124–126] to address it.

5. CONCLUSIONS AND FUTURE WORK

This thesis has presented approaches towards various issues faced while programming accelerator-based systems. Chapter 1 looked at the automatic tuning mechanisms for OpenMP-to-CUDA translators, and proposed a new tuning system that obtains higher performing configurations faster than the previous systems. Chapter 2 presented a mechanism to automatically scale out-of-card computations to limited-sized GPU memories. It also presented a mechanism to overlap kernel computations with communications, and described automatic multi-GPU scalability. Chapter 3 presented a MapReduce programming system that can use both CPUs and accelerators in a cluster with a single input source code. The chapter also presented a scheduling mechanism that addressed the load imbalance issue across CPUs and accelerators. Finally, Chapter 4 formalized the notion of structured control flow graphs, and proposed an unstructured-to-structured CFG conversion mechanism. This mechanism can reduce the impact of divergent code execution on accelerators with SIMD-style execution units.

The techniques proposed in the thesis have catered to some of the fundamental issues in accelerator programming, and have also opened up avenues for future work, as we list below.

- **Extending COSP**: As was seen in chapter 2, while the COSP technique works fine for regular applications, the splitting can not occur if the data access information is not correctly available. Such scenarios are common in irregular applications. As the compiler can not automatically perform computation splitting, one way to deal with irregular applications might be to obtain more information from the programmer, with an additional directive. The other alternative would be to predict the access pattern at the runtime by recording the previous accesses.

- **Further work on HeteroDoop**: The current HeteroDoop system does not utilize GPUs for global reductions, primarily since the considered applications were map-heavy. The future work should investigate applications which are reduce-heavy and experiment with the GPU-offloading of the same. The current *tail scheduling* scheme caters to intra-node heterogeneity, where all nodes contain same CPUs and equal number of GPUs. To handle scenarios with inter-node heterogeneity, data locality needs to be considered, as the current tail scheduling scheme would result in the execution of excessive remote map tasks.

- **Control flow structuring**: We found that conventional compilers do not perform all their passes as dataflow algorithms. The most performance-critical passes are performed on loops, which do not necessarily traverse the entire flow graph. However, the interspersed nature of the loop-based and other passes makes it difficult to decide exactly when the structuring pass should be run. This is an avenue for future work. While this thesis has proved that compilers can generate unstructured CFGs, it is not clear if they do generate irreducible CFGs routinely. A study [99] evaluates the presence of irreducibility in compiler-generated llvm bytecodes, and concludes that it is very rare, and hence can be ignored. However, looking for irreducibility in the generated binary code is a more appropriate way to assess whether compilers generate irreducible code or not. We leave such a study to the future work.

REFERENCES

REFERENCES

[1] NVIDIA, “CUDA,” [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html, 2014, (accessed June 30, 2014).

[2] AMD, “Gcn architecture,” [Online]. Available: http://www.amd.com/en-us/innovations/software-technologies/gcn, 2014, (accessed September 02, 2014).

[3] Intel, “Intel xeon phi coprocessor - the architecture,” [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner, 2014, (accessed September 02, 2014).

[4] IBM, “Cell broadband engine,” [Online]. Available: https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine, 2014, (accessed September 02, 2014).

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, “Brook for GPUs: stream computing on graphics hardware,” in *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*. ACM, 2004, pp. 777–786.

[6] OpenCL, “OpenCL,” [Online]. Available: http://www.khronos.org/opencl/, 2014, (accessed July 28, 2014).

[7] 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. IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.36

[8] T. Han and T. Abdelrahman, ““hiCUDA: High-Level GPGPU Programming”,” *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 78 –90, jan. 2011.

[9] OpenACC, “OpenACC: Directives for Accelerators,” [Online]. Available: http://www.openacc-standard.org, 2014, (accessed June. 11, 2014).

[10] PGI, “PGI OpenACC Compiler http://http://www.pgroup.com/resources/accel.htm.”

[11] CAPS, “CAPS Compilers http://www.caps-entreprise.com/products/caps-compilers/.”

[12] J. Beyer, “Use of OpenACC and OpenMP directives in CRAY Compilation Environment (CCE) http://on-demand.gputechconf.com/gtc/2013/presentations/S3084-OpenACC-OpenMP-Directives-CCE.pdf.”

[13] OpenMP, "OpenMP: Version 4.0," [Online]. Available: http://openmp.org/wp/openmp-specifications/, 2013, (accessed July 31, 2014).

[14] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "Effects of compiler optimizations in openmp to CUDA translation," in *OpenMP in a Heterogeneous World - 8th International Workshop on OpenMP, IWOMP 2012, Rome, Italy, June 11-13, 2012. Proceedings*, 2012, pp. 169–181. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30961-8_13

[15] A. Sabne, P. Sakdhnagool, S. Lee, and J. Vetter, *Evaluating Performance Portability of OpenACC.* Springer International Publishing, 2015, pp. 51–66. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-17473-0_4

[16] A. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter, "Understanding portability of a high-level programming model on contemporary heterogeneous architectures," *IEEE Micro*, vol. 35, no. 4, pp. 48–58, 2015. [Online]. Available: http://dx.doi.org/10.1109/MM.2015.73

[17] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "Scaling large-data computations on multi-gpu accelerators," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 443–454. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465023

[18] F. Song, S. Tomov, and J. Dongarra, ""Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems"," in *Proceedings of the 26th ACM international conference on Supercomputing*, ser. ICS '12. ACM, 2012, pp. 365–376. [Online]. Available: http://doi.acm.org/10.1145/2304576.2304625

[19] F. Zhang, L. Hu, J. Wu, and X. Shen, ""A SPH-based method for interactive fluids simulation on the multi-GPU"," in *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, ser. VRCAI '11. ACM, 2011, pp. 423–426. [Online]. Available: http://doi.acm.org/10.1145/2087756.2087834

[20] J. Kim, H. Kim, J. H. Lee, and J. Lee, ""Achieving a single compute device image in OpenCL for multiple GPUs"," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP '11. ACM, 2011, pp. 277–288. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941591

[21] M. Bauer, H. Cook, and B. Khailany, ""CudaDMA: Optimizing GPU memory bandwidth via warp specialization"," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, nov. 2011, pp. 1 –11.

[22] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil, ""Software Pipelined Execution of Stream Programs on GPUs"," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. IEEE Computer Society, 2009, pp. 200–209. [Online]. Available: http://dx.doi.org/10.1109/CGO.2009.20

[23] A. Hagiescu, H. P. Huynh, W.-F. Wong, and R. Goh, ""Automated Architecture-Aware Mapping of Streaming Applications Onto GPUs"," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, may 2011, pp. 467 –478.

[24] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke, ""Sponge: portable stream programming on graphics engines"," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '11. ACM, 2011, pp. 381–392. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950409

[25] Apache, "Hadoop," [Online]. Available: http://hadoop.apache.org/, 2014, (accessed June 30, 2014).

[26] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[27] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. New York, NY, USA: ACM, 2007, pp. 59–72. [Online]. Available: http://doi.acm.org/10.1145/1272996.1273005

[28] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855742

[29] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, ""Mars: a MapReduce framework on graphics processors"," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. ACM, 2008, pp. 260–269. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454152

[30] J. Stuart and J. Owens, ""Multi-GPU MapReduce on GPU Clusters"," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, may 2011, pp. 1068 –1079.

[31] B. Catanzaro, N. Sundaram, and K. Keutzer, "A map reduce framework for programming graphics processors," in *In Workshop on Software Tools for MultiCore Systems*, 2008.

[32] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "Heterodoop: A mapreduce programming system for accelerator clusters," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: ACM, 2015, pp. 235–246. [Online]. Available: http://doi.acm.org/10.1145/2749246.2749261

[33] I. El-Helw, R. Hofman, and H. E. Bal, "Glasswing: Accelerating mapreduce on multi-core and many-core clusters," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: ACM, 2014, pp. 295–298. [Online]. Available: http://doi.acm.org/10.1145/2600212.2600706

[34] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "Mapcg: Writing parallel program portable between cpu and gpu," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 217–226. [Online]. Available: http://doi.acm.org/10.1145/1854273.1854303

[35] H. Wu, G. Diamos, J. Wang, S. Li, and S. Yalamanchili, "Characterization and transformation of unstructured control flow in bulk synchronous gpu applications," *Int. J. High Perform. Comput. Appl.*, vol. 26, no. 2, pp. 170–185, May 2012. [Online]. Available: http://dx.doi.org/10.1177/1094342011434814

[36] Z. Ammarguellat, "A control-flow normalization algorithm and its complexity," *IEEE Trans. Softw. Eng.*, vol. 18, no. 3, pp. 237–251, Mar. 1992. [Online]. Available: http://dx.doi.org/10.1109/32.126773

[37] F. Zhang and E. H. D'Hollander, "Using hammock graphs to structure programs," *IEEE Trans. Softw. Eng.*, vol. 30, no. 4, pp. 231–245, Apr. 2004. [Online]. Available: http://dx.doi.org/10.1109/TSE.2004.1274043

[38] A. Erosa and L. Hendren, "Taming control flow: a structured approach to eliminating goto statements," in *Computer Languages, 1994., Proceedings of the 1994 International Conference on*, May 1994, pp. 229–240.

[39] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of control dependence to data dependence," in *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '83. New York, NY, USA: ACM, 1983, pp. 177–189. [Online]. Available: http://doi.acm.org/10.1145/567067.567085

[40] B. S. Baker, "An algorithm for structuring flowgraphs," *J. ACM*, vol. 24, no. 1, pp. 98–120, 1977. [Online]. Available: http://doi.acm.org/10.1145/321992.321999

[41] 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

[42] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 319–332. [Online]. Available: http://dx.doi.org/10.1109/CGO.2006.38

[43] W. Blume and R. Eigenmann, "Performance analysis of parallelizing compilers on the perfect benchmarks programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 643–656, 1992.

[44] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, “Compiler optimization-space exploration,” in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO ’03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 204–215. [Online]. Available: http://dl.acm.org/citation.cfm?id=776261.776284

[45] R. P. J. Pinkers, P. M. W. Knijnenburg, M. Haneda, and H. A. G. Wijshoff, “Statistical selection of compiler options,” in *Proceedings of the The IEEE Computer Society’s 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, ser. MASCOTS ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 494–501. [Online]. Available: http://dl.acm.org/citation.cfm?id=1032659.1034235

[46] K. D. Cooper, D. Subramanian, and L. Torczon, “Adaptive optimizing compilers for the 21st century,” *J. Supercomput.*, vol. 23, pp. 7–22, August 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=603339.603341

[47] OpenACC, http://www.openacc-standard.org/, November 2011. [Online]. Available: http://www.openacc-standard.org/

[48] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, ““Automatic CPU-GPU communication management and optimization”,” *SIGPLAN Not.*, vol. 47, no. 6, pp. 142–151, Jun. 2011. [Online]. Available: http://doi.acm.org/10.1145/2345156.1993516

[49] M. Amini, F. Coelho, F. Irigoin, and R. Keryell, ““Static Compilation Analysis for Host-Accelerator Communication Optimization”,” in *24th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, sep 2011, also Technical Report MINES ParisTech A/476/CRI.

[50] L. Gu, J. Siegel, and X. Li, ““Using GPUs to compute large out-of-card FFTs”,” in *Proceedings of the international conference on Supercomputing*, ser. ICS ’11. ACM, 2011, pp. 255–264. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995937

[51] S. zee Ueng, M. Lathara, S. S. Baghsorkhi, and W. mei W. Hwu, ““CUDA-Lite: Reducing GPU programming complexity”,” in *In: LCPC’08. Volume 5335 of LNCS*. Springer, 2008, pp. 1–15.

[52] Y. Yan, M. Grossman, and V. Sarkar, ““JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA”,” *Euro-Par 2009 Parallel Processing*, p. 887–899, 2009.

[53] S. Hong and H. Kim, ““An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness”,” in *Proceedings of the 36th annual international symposium on Computer architecture*, ser. ISCA ’09. ACM, 2009, pp. 152–163. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555775

[54] 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. ACM, 2008, pp. 195–204. [Online]. Available: http://doi.acm.org/10.1145/1356058.1356084

[55] NVIDIA Corporation, *CUDA C Best Practices Guide*, May 2011.

[56] 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

[57] OpenMP, "Improvement to support Accelerators http://openmp.org/wp/2012/03/openmp-is-being-improved-for-accelerators-multicore-and-embedded-systems/," 2012.

[58] H. Bae, D. Mustafa, J.-W. Lee, Aurangzeb, H. Lin, C. Dave, R. Eigenmann, and S. Midkiff, ""The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation"," *International Journal of Parallel Programming*, pp. 1–15, 2012, 10.1007/s10766-012-0211-z. [Online]. Available: http://dx.doi.org/10.1007/s10766-012-0211-z

[59] H. Bae and R. Eigenmann, ""Interprocedural symbolic range propagation for optimizing compilers"," in *Proceedings of the 18th international conference on Languages and Compilers for Parallel Computing*, ser. LCPC'05. Springer-Verlag, 2006, pp. 413–424. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69330-7_28

[60] "StreamIt Benchmarks [Online]. Available: http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml."

[61] NVIDIA, "CUDA SDK," [Online]. Available: https://developer.nvidia.com/gpu-computing-sdk, 2014, (accessed July. 31, 2014).

[62] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, ""Rodinia: A benchmark suite for heterogeneous computing"," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. IEEE Computer Society, 2009, pp. 44–54. [Online]. Available: http://dx.doi.org/10.1109/IISWC.2009.5306797

[63] W. Thies, M. Karczmarek, and S. P. Amarasinghe, ""StreamIt: A Language for Streaming Applications"," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. Springer-Verlag, 2002, pp. 179–196. [Online]. Available: http://dl.acm.org/citation.cfm?id=647478.727935

[64] H. P. Huynh, A. Hagiescu, W.-F. Wong, and R. S. M. Goh, ""Scalable framework for mapping streaming applications onto multi-GPU systems"," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. ACM, 2012, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145818

[65] A. Aji, J. Dinan, D. Buntinas, P. Balaji, W. chun Feng, K. Bisset, and R. Thakur, ““MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-based Systems”,” in *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, june 2012, pp. 647 –654.

[66] Y. Zhang and F. Mueller, ““Auto-generation and auto-tuning of 3D stencil codes on GPU clusters”,” in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO ’12. ACM, 2012, pp. 155–164. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259037

[67] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. R. Quinn, ““Scaling Hierarchical N-body Simulations on GPU Clusters”,” in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’10. IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.49

[68] A. Nukada and S. Matsuoka, ““Auto-tuning 3-D FFT library for CUDA GPUs”,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC ’09. ACM, 2009, pp. 30:1–30:10. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654090

[69] J. W. Choi, A. Singh, and R. W. Vuduc, ““Model-driven autotuning of sparse matrix-vector multiply on GPUs”,” in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’10. ACM, 2010, pp. 115–126. [Online]. Available: http://doi.acm.org/10.1145/1693453.1693471

[70] R. Farivar, A. Verma, E. Chan, and R. Campbell, “Mithra: Multiple data independent tasks on a heterogeneous resource architecture,” in *Cluster Computing and Workshops, 2009. CLUSTER ’09. IEEE International Conference on*, Aug 2009, pp. 1–10.

[71] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, “Improving mapreduce performance in heterogeneous environments,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. Berkeley, CA, USA: USENIX Association, 2008, pp. 29–42. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855744

[72] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, “Tarazu: Optimizing mapreduce on heterogeneous clusters,” *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 61–74, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2189750.2150984

[73] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[74] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The hadoop distributed file system,” in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/MSST.2010.5496972

[75] "Hadoop streaming," [Online]. Available: http://hadoop.apache.org/docs/r1.2.1/streaming.html, (accessed June 02, 2014).

[76] libHDFS, "API http://wiki.apache.org/hadoop/LibHDFS," 2014, (accessed July. 30, 2014).

[77] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan primitives for gpu computing," in *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, ser. GH '07. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2007, pp. 97–106. [Online]. Available: http://dl.acm.org/citation.cfm?id=1280094.1280110

[78] N. Satish, M. Harris, and M. Garland, "Designing efficient sorting algorithms for manycore gpus," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–10.

[79] A. Davidson, D. Tarjan, M. Garland, and J. Owens, "Efficient parallel merge sort for fixed and variable length keys," in *Innovative Parallel Computing (InPar), 2012*, May 2012, pp. 1–9.

[80] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," School of Electrical and Computer Engineering, Purdue University, West Lafayette, Indiana, Tech. Rep. Paper 437, October 2012. [Online]. Available: http://docs.lib.purdue.edu/ecetr/437

[81] W. Jiang, V. T. Ravi, and G. Agrawal, "A map-reduce system with an alternate api for multi-core environments," 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. 84–93. [Online]. Available: http://dx.doi.org/10.1109/CCGRID.2010.10

[82] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 13–24. [Online]. Available: http://dx.doi.org/10.1109/HPCA.2007.346181

[83] M. Grossman, M. Breternitz, and V. Sarkar, "Hadoopcl: Mapreduce on distributed heterogeneous platforms through seamless integration of hadoop and opencl," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1918–1927. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2013.246

[84] K. Shirahata, H. Sato, and S. Matsuoka, "Hybrid map task scheduling for gpu-based heterogeneous clusters," *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 0, pp. 733–740, 2010.

[85] L. Chen, X. Huo, and G. Agrawal, "Accelerating mapreduce on a coupled cpu-gpu architecture," 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. 25:1–25:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389030

[86] L. Chen and G. Agrawal, "Optimizing mapreduce for gpus with effective shared memory usage," in *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '12. New York, NY, USA: ACM, 2012, pp. 199–210. [Online]. Available: http://doi.acm.org/10.1145/2287076.2287109

[87] C. Böhm and G. Jacopini, "Flow diagrams, turing machines and languages with only two formation rules," *Commun. ACM*, vol. 9, no. 5, pp. 366–371, May 1966. [Online]. Available: http://doi.acm.org/10.1145/355592.365646

[88] F. E. Allen, "Control flow analysis," in *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: ACM, 1970, pp. 1–19. [Online]. Available: http://doi.acm.org/10.1145/800028.808479

[89] M. H. Williams, "Generating structured flow diagrams: The nature of unstructuredness," *Comput. J.*, vol. 20, no. 1, pp. 45–50, 1977. [Online]. Available: http://dx.doi.org/10.1093/comjnl/20.1.45

[90] N. Chapin and S. P. Denniston, "Characteristics of a structured program," *SIGPLAN Not.*, vol. 13, no. 5, pp. 36–45, May 1978. [Online]. Available: http://doi.acm.org/10.1145/953395.953398

[91] B. Blackham and G. Heiser, "Sequoll: A framework for model checking binaries," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, April 2013, pp. 97–106.

[92] I. Matosevic and T. S. Abdelrahman, "Efficient bottom-up heap analysis for symbolic path-based data access summaries," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 252–263. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259049

[93] A. Shankar, S. S. Sastry, R. Bodík, and J. E. Smith, "Runtime specialization with optimistic heap analysis," *SIGPLAN Not.*, vol. 40, no. 10, pp. 327–343, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094837

[94] Y. Xie and A. Aiken, "Saturn: A scalable framework for error detection using boolean satisfiability," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 3, May 2007. [Online]. Available: http://doi.acm.org/10.1145/1232420.1232423

[95] R. Hundt, E. Raman, M. Thuresson, and N. Vachharajani, "Mao – an extensible micro-architectural optimizer," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2190025.2190077

[96] S. Kalvala, R. Warburton, and D. Lacey, "Program transformations using temporal logic side conditions," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 4, pp. 14:1–14:48, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1516507.1516509

[97] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy, "A global communication optimization technique based on data-flow analysis and linear algebra," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 6, pp. 1251–1297, Nov. 1999. [Online]. Available: http://doi.acm.org/10.1145/330643.330647

[98] M. Woodward, M. Hennell, and D. Hedley, “A measure of control flow complexity in program text,” *Software Engineering, IEEE Transactions on*, vol. SE-5, no. 1, pp. 45–50, Jan 1979.

[99] J. Stanier and D. Watson, “A study of irreducibility in c programs,” *Softw. Pract. Exper.*, vol. 42, no. 1, pp. 117–130, Jan. 2012. [Online]. Available: http://dx.doi.org/10.1002/spe.1059

[100] F. E. Allen and J. Cocke, “A program data flow analysis procedure,” *Commun. ACM*, vol. 19, no. 3, pp. 137–, Mar. 1976. [Online]. Available: http://doi.acm.org/10.1145/360018.360025

[101] S. Unger and F. Mueller, “Handling irreducible loops: Optimized node splitting versus dj-graphs,” *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 4, pp. 299–333, Jul. 2002. [Online]. Available: http://doi.acm.org/10.1145/567097.567098

[102] E. A. Ashcroft and Z. Manna, “The translation of ‘go to’ programs to ‘while’ programs,” in *IFIP Congress (1)*, 1971, pp. 250–255.

[103] L. Ramshaw, “Eliminating go to’s while preserving program structure,” *J. ACM*, vol. 35, no. 4, pp. 893–920, Oct. 1988. [Online]. Available: http://doi.acm.org/10.1145/48014.48021

[104] M. H. Williams and H. L. Ossher, “Conversion of unstructured flow diagrams to structured form,” *Comput. J.*, vol. 21, no. 2, pp. 161–167, 1978. [Online]. Available: http://dx.doi.org/10.1093/comjnl/21.2.161

[105] G. Oulsnam, “The algorithmic transformation of schemas to structured form,” *Comput. J.*, vol. 30, no. 1, pp. 43–51, 1987. [Online]. Available: http://dx.doi.org/10.1093/comjnl/30.1.43

[106] M. S. Hecht, *Flow Analysis of Computer Programs.* New York, NY, USA: Elsevier Science Inc., 1977.

[107] L. Carter, J. Ferrante, and C. D. Thomborson, “Folklore confirmed: reducible flow graphs are exponentially larger,” in *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, 2003, pp. 106–114. [Online]. Available: http://doi.acm.org/10.1145/640128.604141

[108] P. Havlak, “Nesting of reducible and irreducible loops,” *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 4, pp. 557–567, Jul. 1997. [Online]. Available: http://doi.acm.org/10.1145/262004.262005

[109] C. Collberg, C. Thomborson, and D. Low, “Manufacturing cheap, resilient, and stealthy opaque constructs,” in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’98. New York, NY, USA: ACM, 1998, pp. 184–196. [Online]. Available: http://doi.acm.org/10.1145/268946.268962

[110] D. L. Christian Collberg, Clark Thomborson, “A taxonomy of obfuscating transformations,” 1997.

[111] M. S. Hecht and J. D. Ullman, “Characterizations of reducible flow graphs,” *J. ACM*, vol. 21, no. 3, pp. 367–375, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/321832.321835

[112] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[113] C. T. Zahn, "Structured control in programming languages," *SIGPLAN Not.*, vol. 10, no. 7, pp. 13–15, Jul. 1975. [Online]. Available: http://doi.acm.org/10.1145/987305.987308

[114] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Eds., *Structured Programming*. London, UK, UK: Academic Press Ltd., 1972.

[115] E. Moretti, G. Chanteperdrix, and A. Osorio, "New algorithms for control-flow graph structuring," in *Fifth Conference on Software Maintenance and Reengineering, CSMR 2001, Lisbon, Portugal, March 14-16, 2001*, 2001, pp. 184–187. [Online]. Available: http://dx.doi.org/10.1109/.2001.914984

[116] E. W. Dijkstra, "Letters to the editor: Go to statement considered harmful," *Commun. ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968. [Online]. Available: http://doi.acm.org/10.1145/362929.362947

[117] D. E. Knuth, "Structured programming with go to statements," *ACM Comput. Surv.*, vol. 6, no. 4, pp. 261–301, Dec. 1974. [Online]. Available: http://doi.acm.org/10.1145/356635.356640

[118] C. Cifuentes, "Structuring decompiled graphs," Australia, Tech. Rep., 1994.

[119] T. Wei, J. Mao, W. Zou, and Y. Chen, "Structuring 2-way branches in binary executables," in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 1, July 2007, pp. 115–118.

[120] G. Oulsnam, "Unravelling unstructured programs," *Comput. J.*, vol. 25, no. 3, pp. 379–387, 1982. [Online]. Available: http://dx.doi.org/10.1093/comjnl/25.3.379

[121] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 75–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977395.977673

[122] "Omni OpenMP benchmarks," [Online]. Available: http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/download/download-benchmarks.html, 2016, (accessed March 11, 2016).

[123] N. A. S. Division, "NAS Parallel Benchmarks," [Online]. Available: https://www.nas.nasa.gov/publications/npb.html/, 2016, (accessed March 11, 2016).

[124] J. C. Kleinsorge, H. Falk, and P. Marwedel, "Simple analysis of partial worst-case execution paths on general control flow graphs," in *Proceedings of the Eleventh ACM International Conference on Embedded Software*, ser. EMSOFT '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 16:1–16:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2555754.2555770

[125] S. Hepp and F. Brandner, "Splitting functions into single-entry regions," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '14. New York, NY, USA: ACM, 2014, pp. 17:1–17:10. [Online]. Available: http://doi.acm.org/10.1145/2656106.2656128

[126] C. Wimmer and M. Franz, "Linear scan register allocation on ssa form," in *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 170–179. [Online]. Available: http://doi.acm.org/10.1145/1772954.1772979

VITA

VITA

Amit Sabne received his PhD under the guidance of Prof. Rudolf Eigenmann in August 2016. Before that, he completed his bachelors (B.E. Hons.) from BITS Pilani, India in 2009. His research interests are broadly in all aspects of computer systems, including but not limited to: programming languages, compiler transformations, system architectures, distributed and parallel computing, and theory of computation.