Program Analysis and Compiler Transformations for Computational Accelerators

by

Taylor Lloyd

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© Taylor Lloyd, 2018

Abstract

Heterogeneous computing is becoming increasingly common in high-end computer systems, with vendors often including compute accelerators such as Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) for increased throughput and power efficiency. This thesis addresses the usability and performance of compute accelerators, with an emphasis on compiler-driven analyses and transformations.

First this thesis studies the challenge of programming for FPGAs. IBM and Intel both now produce systems with integrated FPGAs, but FPGA programming remains extremely challenging. To mitigate this difficulty, FPGA vendors now ship OpenCL-based High-Level Synthesis (HLS) tools, capable of generating Hardware Description Language (HDL) from Open Compute Language (OpenCL) source. Unfortunately, most OpenCL source today is written to be executed on GPUs, and runs poorly on FPGAs. This thesis explores traditional compiler analyses and transformations to automatically transform GPU-targeted OpenCL, achieving speedups up to 6.7x over unmodified OpenCL benchmarks written for GPUs.

This thesis next addresses the problem of automatically mapping OpenMP 4.X target regions to GPU hardware. In OpenMP, the compiler is responsible for determining the number and grouping of GPU threads, and the existing heuristic in LLVM/Clang performs poorly for a large subset of programs. We perform an exhaustive data collection over 23 OpenMP benchmarks from the SPEC ACCEL and Unibench suites. From our dataset, we propose a new

grid geometry heuristic resulting in a 25% geomean speedup over geompetries selected by the original LLVM/Clang heuristic.

The third contribution of this thesis is related to the performance of an application executing in Graphics Processing Units (GPUs). Such performance can be significantly degraded by irregular data accesses and by control-flow divergence. Both of these performance issues arise only in the presence of thread-divergent expressions—an expression that evaluates to different values for different threads. This thesis introduces GPUCheck: a static analysis tool that detects branch divergence and non-coalesceable memory accesses in GPU programs. GPUCheck relies on a static dataflow analysis to find thread-dependent expressions and on a novel symbolic analysis to determine when such expressions could lead to performance issues. Kernels taken from the Rodinia benchmark suite and repaired by GPUCheck execute up to 30% faster than the original kernels.

The fourth contribution of this thesis focuses on data transmission in a heterogeneous computing system. GPUs can be used as specialized accelerators to improve network connectivity. We present Run-Length Base-Delta (RLBD) encoding, a very high-speed compression format and algorithm capable of improving throughput of 40GbE up to 57% on datasets taken from the UCI Machine Learning Repository.

Preface

Chapter 4 has been published as T. Lloyd, A. Chikin, E. Ochoa, K. Ali, J.N. Amaral, "A Case for Better Integration of Host and Target Compilation When Using OpenCL for FPGAs," *Proceedings of the Fourth International Workshop on FPGAs for Software Programmers*, in Sept 2017. I was responsible for identifying transformation opportunities, implementing the device-side transformation of NDRange kernels to single workitem kernels, evaluating the experimental results, and drafting the manuscript. A. Chikin was responsible for implementing pointer restriction and dependence elimination, and drafting the manuscript. E. Ochoa was responsible for implementing the host-side transformation of NDRange to single workitem kernels, implementing the final toolchain, running the experiments, and drafting the manuscript. K. Ali and J.N. Amaral were supervisory authors and contributed to the focus of the final manuscript and to manuscript edits.

Chapter 6 has been submitted for publication as T. Lloyd, K. Ali, J.N. Amaral, "GPUCheck: Detecting CUDA Thread Divergence with Static Analysis." I was responsible for designing, implementing, and evaluating the framework and analysis. K. Ali and J.N. Amaral were supervisory authors and contributed to the experiment design, data analysis, and manuscript edits.

Chapter 5 has been submitted for publication as T. Lloyd, A. Chikin, S. Kedia, D. Jain, J.N. Amaral, "Automated GPU Grid Geometry Selection for OpenMP Kernels." I supervised and guided the data collection and machine learning model design. I analyzed the model created by the machine learning technique and proposed the final heuristic. A. Chikin also supervised data collection and machine learning model design, and drafted the manuscript. S. Kedia performed the machine learning model training and evaluation. D. Jain

performed the initial data collection. J.N. Amaral was a supervisory author, and contributed to manuscript edits.

Chapter 7 was completed in collaboration with IBM Canada and has been submitted for publication as T. Lloyd, K. Barton, E. Tiotto, J.N. Amaral, ”Run-Length Base-Delta Encoding for High-Speed Compression.” I was responsible for designing, implementing, and evaluating the framework and analysis, and drafting the manuscript. K. Barton and E. Tiotto were IBM collaborators, and participated in manuscript edits. J.N. Amaral was a supervisory author, and contributed to manuscript edits.

To Maddy

For supporting me, marrying me, and listening to my arcane problems.

Acknowledgements

I'd like to acknowledge my supervisor, Dr J Nelson Amaral, for supporting me throughout my university career, and always having the right advice. Thank-you for the many hours teaching and guiding me.

Thank-you also to Dr Karim Ali, for introducing me to static analysis, and opening me to new perspectives on old topics.

This research has been funded by IBM Center for Advanced Studies (CAS) Canada, the Intel FPGA University program, the government of Alberta, the Natural Science and Engineering Research Council (NSERC) and the University of Alberta. Thank-you all for the essential support.

Contents

1 Introduction **1**

2 Background **3**

2.1 Accelerator Devices 3
2.1.1 Graphics Processing Units (GPUS) 3
2.1.2 Field-Programmable Gate Arrays (FPGAs) 5
2.2 Accelerator Programming Languages 6
2.2.1 CUDA 6
2.2.2 OpenMP 6
2.2.3 OpenCL 8
2.3 GPU Divergence Analysis 9
2.4 Software Compression 10
2.4.1 Lossless Compression 11
2.4.2 Parallel Compression 12
2.4.3 Compression in Networking 12

3 Related Work **14**

3.1 FPGA High-Level Synthesis 14
3.1.1 Manually Optimized OpenCL 15
3.1.2 Combined Host/Device Compilation 16
3.2 GPU Parallelism Mapping 16
3.3 GPU Divergence Analysis 17
3.4 High-Speed GPU Compression 17

4 A Case for Better Integration of Host and Target Compilation When Using OpenCL for FPGAs **19**

4.1 Introduction 19
4.2 Optimizing OpenCL for FPGAs 21
4.2.1 `restrict` Pointers to Enable Simultaneous Memory Operations 21
4.2.2 Prefer Single-Work-Item kernels over NDRange kernels 22
4.2.3 Pipelining Reduction Operations with Shifting Arrays . 22
4.3 Compiling OpenCL for FPGAs 24
4.3.1 NDRange to Single Work-Item Loop (NDRangeToLoop) 25
4.3.2 Reduction-Dependence Elimination 28
4.3.3 Restrict Pointer Kernel Parameters 30
4.4 Prototype Performance Study 30
4.4.1 Benchmarks 31
4.4.2 Reduction-Dependence Elimination Efficacy 33
4.5 Concluding Remarks 34

5 Automated GPU Grid Geometry Selection for OpenMP Kernels 36
5.1 Introduction . . . 36
5.2 Mapping OpenMP to GPUs . . . 38
5.2.1 Nvidia P100 Geometry . . . 38
5.2.2 OpenMP 4 in LLVM/Clang . . . 39
5.3 Data Collection . . . 40
5.3.1 Best Discovered Grid Geometry Performance Relative to Compiler Default . . . 42
5.3.2 Threads Per Block . . . 42
5.3.3 Number of Blocks . . . 44
5.4 Modeling with Machine Learning . . . 45
5.4.1 Finding Additional Features . . . 46
5.4.2 Machine Learning Predictor Performance . . . 48
5.5 Production Heuristic . . . 48
5.5.1 Edge-Case: OpenMP SIMD . . . 54
5.5.2 Implications of Volta . . . 54
5.6 Concluding Remarks . . . 55

6 GPUCheck: Detecting CUDA Thread Divergence with Static Analysis 58
6.1 Introduction . . . 58
6.2 Static Analysis Engine . . . 61
6.2.1 Thread-Dependence Analysis . . . 61
6.2.2 Arithmetic Control Form (ACF) . . . 64
6.2.3 Inter-procedural Arithmetic Control Form (IACF) . . . 68
6.3 Detecting Divergent Behaviour . . . 69
6.3.1 Divergent-Branch Analysis . . . 70
6.3.2 Non-coalescable Memory Access Analysis . . . 71
6.4 An LLVM Prototype for GPUCheck . . . 73
6.5 Evaluation . . . 74
6.5.1 Does GPUCheck provide similar results to dynamic profiling? . . . 75
6.5.2 Do the problems identified by GPUCheck reflect real performance opportunities? . . . 78
6.5.3 Is GPUCheck performant enough to be used during active development? . . . 82
6.6 Concluding Remarks . . . 82

7 Run-Length Base-Delta Encoding for High-Speed Compression 84
7.1 A New Data Compression Algorithm . . . 84
7.2 RLBD Compression Format . . . 86
7.3 RLBD Compression and Decompression . . . 88
7.3.1 Serial Compression . . . 88
7.3.2 Serial Decompression . . . 90
7.4 GPU-Accelerated RLBD . . . 91
7.4.1 Parallel Compression . . . 91
7.5 Evaluation . . . 95
7.5.1 Is RLBD faster than traditional software compression schemes? . . . 96
7.5.2 What compression rates are required by modern supercomputing interconnects? . . . 101
7.5.3 Is RLBD compression effective on real-world data? . . . 103

7.5.4 What throughput improvements can be expected when implementing RLBD? . . . 103
7.6 Concluding Remarks . . . 104

8 Conclusion **106**

References **108**

List of Tables

4.1 Benchmark Execution Time and Applicable Transformations . 30

5.1 Grid Geometry (threads-per-block, blocks) selected for each benchmark by the LLVM selection, our ML model, our proposed heuristic, and exhaustive search. Speedup is shown relative to the LLVM selection. Slowdowns are shown as reciprocals for clarity. Thread-Per-Block values for the the Final Heuristic selected using leave-one-out strategy as described in 5.3.2. . . 56

6.1 Execution time for GPUCheck vs dynamic profiling. Branches and accesses show the number of instructions requiring ACF analysis over all instructions analyzed. 83

7.1 Identifiers for various possible schema. 87
7.2 Machines Used for performance testing 96
7.3 Datasets used for real-world testing 97
7.4 Common high-speed interconnects, their throughput, and compression/decompression throughput requirements by data savings ratio 101

List of Acronyms

API	Application Program Interface
BΔI	Base-Delta Intercept
CPU	Central Processing Unit
DMA	Direct Memory Access
DS	Data Savings
FPGA	Field-Programmable Gate Array
HLS	High-Level Synthesis
GPU	Graphics Processing Unit
LUT	Lookup Table
RLBD	Run-Length Base-Delta
SIMD	Same-Instruction Multiple-Data
SIMT	Same-Instruction Multiple-Thread
SM	Streaming Multiprocessor

List of Figures

2.1 OpenMP 3 example modified from `gemm` in the Unibench benchmark suite . . . 7
2.2 OpenMP 4 example simplified from `gemm` in the Unibench benchmark suite . . . 8

4.1 Altera OpenCL Compilation Flow. From [17]. . . . 20
4.2 Floating-point reduction sample preventing loop pipelining. From [31]. . . . 23
4.3 Floating-point reduction using a shift register to enable loop pipelining. From [31]. . . . 24
4.5 Host NDRange kernel invocation . . . 25
4.6 HelloWorld Kernel. From [2]. . . . 27
4.7 After NDRange Transformation [2]. . . . 27
4.8 After Constant Propagation [2]. . . . 27
4.4 Custom OpenCL Compilation Flow . . . 28

5.1 Execution time of kernel 21, as a function of the number and size of blocks. The best discovered point is 9.8x faster than the LLVM selection. . . . 37
5.2 Improvement available with the best discovered grid geometry versus the LLVM selection. . . . 43
5.3 Minimum Execution Time at set threads per block / Minimum Execution Time. Each line represents 22 of the 23 benchmarks in a leave-one-out manner. . . . 44
5.4 Minimum Execution Time given 96 threads per block / Minimum Execution Time. Average is weighted by the minimum execution time of each benchmark. . . . 45
5.5 Number of blocks minimizing execution time given 96 threads per block / Loop Tripcount for each benchmark. . . . 45
5.6 Speedup over the LLVM selection for the Random Forest Model Predictor grid configuration not including the prediction overhead. This performance cannot be realized in practice. Results are shown on a log scale to present equivalent speedups and slowdowns as equivalently-sized bars. . . . 49
5.7 Final Heuristic Algorithm . . . 52
5.8 Speedup for our modified heuristic over the LLVM selection. This performance can be realized in practice. Results are shown on a log scale. . . . 53

6.1 Original diffusion coefficient calculation in `srad`. . . . 59
6.2 Modified diffusion coefficient calculation in `srad`. . . . 59
6.3 Extract from `streamcluster`. . . . 60
6.4 `streamcluster` access pattern before (left) and after the code transformation (right). . . . 60

6.5 An example illustrating thread dependence. 62
6.6 An example illustrating if-conversion in ψ-SSA, which serves as inspiration for our ACF analysis. 64
6.7 An example illustrating how ACF handles loops. 68
6.8 An example illustrating the need for Inter-procedural Arithmetic Control Form (IACF). 68
6.9 GPUCheck Analysis Workflow 70
6.10 Coalescing algorithm in GPUCheck. 72
6.11 Divergency issues found in the Rodinia Benchmark Suite. Black indicates an issue found only by GPUCheck. White indicates an issue found only by `nvprof`. Grey indicates an issue found by both. The adjacent fractions are the number of issues found by GPUCheck, over the total issues found. 76
6.12 Original `gaussian` kernel functions (edited for clarity). 79
6.13 Extract from `lavaMD` demonstrating buffering in shared memory. 80
6.14 Original halo computation in `nw` kernels. 81

7.1 Configuration of Header for $h = 16$, $v = 8$, $s = 2$, and $c = 6$. Each small light grey box represents one byte. 86
7.2 RLBD Serial Compression Algorithm 89
7.3 RLBD 1-Lookahead Schema Selection 90
7.4 RLBD Serial Decompression Algorithm 91
7.5 2-Stage parallel compression flow 92
7.6 GPU Compression Pipeline 93
7.7 Visualization of thread-level cooperation within a GPU block during decompression 94
7.8 Compression Throughput by Algorithm on the Sandybridge machine 98
7.9 Throughput of synthetic data compression by schema 99
7.10 Throughput of synthetic data decompression by schema 99
7.11 Throughput vs Data Savings 102

Chapter 1

Introduction

Heterogeneous computing platforms are emerging as the dominant approach to supercomputing. GPUs have become an increasingly popular choice in achieving a superior performance per watt ratio with some of the highest absolute performance. In the most recent Top500 list [66], six out of the top ten machines use accelerator devices. The third fastest machine - the Swiss National Supercomputing Centre's Piz Daint, for example, delivers its performance with $5,320$ NVIDIA Tesla P100 GPUs. The upcoming Summit supercomputer, to be deployed at the Oak Ridge National Laboratory, is set to lead the TOP500 list in 2018 with slated peak performance of 200 petaflops from more than $25,000$ NVIDIA V100 GPUs coupled with $9,200$ IBM POWER 9 CPUs [50]. By coupling GPUs to each node, Summit aims to deliver over 200 petaflops in a power envelope of 10 MW, a five-fold increase over the performance of the previous Titan supercomputer while consuming only 10% more energy. Providing even more power efficiency than GPUs, Field-Programmable Gate Arrays (FPGAs) are beginning to ship in high-end server hardware. Intel is now shipping Xeon server CPUs with an integrated FPGA, and IBM ships System Z machines with a built-in FPGA. These devices are required to achieve necessary performance requirements while consuming less power than traditional CPU-centric systems.

Hardware accelerators such as GPUs and FPGAs can dramatically improve application performance, but require different programming paradigms for efficient execution. GPUs execute thousands of threads simultaneously,

interleaving thread execution to hide instruction latency. FPGAs execute spatially, using deep pipelining to efficiently process tasks. Both models map poorly to traditional programs optimized for a sequential CPU.

Further complicating accelerator programming, both GPUs and FPGAs maintain internal memory, and must manage copying of relevant data to and from the accelerator device. Accelerators are typically connected via a PCIE bus or equivalent, so for many computations data transfer can actually dominate execution time.

To simplify accelerator programming, compiler and language designers have taken two approaches, creating new programming languages like CUDA, OpenCL, OpenMP, and OpenACC. CUDA and OpenCL present programmers a low-level abstract machine model, allowing developers to take advantage of architectural characteristics by hiding few device-specific details. By contrast, OpenMP and OpenACC encourage programmers to express parallelism in their programs, but provide few machine details. In these languages, the compiler is responsible for mapping parallelism to the appropriate architectures, with the goal of hiding architectural details and achieving performance portability into the future.

This work leverages both styles of accelerator languages, applying compiler analyses and transformations to take better advantage of GPUs and FPGAs. This work is a compilation from a number of projects, each presented independently by chapter. Chapter 4 presents compiler transformations for OpenCL when targeting FPGAs, by combining host and device analysis. Chapter 6 presents a compiler analysis for CUDA programs to detect memory access patterns and conditional branches which may cause performance problems when executing on GPUs. Chapter 5 presents a compiler heuristic for mapping OpenMP-parallel programs to GPUs to maximize performance. Finally, Chapter 7 presents a novel compression algorithm, Run-Length Base-Delta (RLBD) originally intended to compress CPU-GPU data transfers. Unfortunately, performance was insufficient for the intended use, but RLBD can be used with GPU acceleration to improve supercomputing network performance.

Chapter 2

Background

2.1 Accelerator Devices

This body of work concentrates on GPUs and FPGAs, two specific accelerator devices.

2.1.1 Graphics Processing Units (GPUS)

GPUs are composed of tens to hundreds of streaming multiprocessors (SMs), each capable of independently executing thousands of threads in parallel. To efficiently load and execute code on such devices, a shared program *kernel* is executed by many threads at once, in a data-parallel fashion. When executing a kernel on a GPU, threads are grouped into thread blocks. All threads within a thread-block execute on a single streaming multiprocessor (SM), and are therefore able to perform cooperative tasks and share intermediate products. By contrast, threads in different thread-blocks cannot communicate directly. The number of thread-blocks and the number of threads per block are collectively referred to as a *grid geometry* and they both must be specified when initiating a kernel execution.

Each SM is a highly vectorized processing unit, capable of executing an instruction for a warp of 32 threads each cycle. All threads within the warp must execute the same instruction, as each SM has only a single instruction decoder. To hide instruction latency, the SM is fully pipelined and executes a different warp of threads each cycle, using zero-cost context switching. The GPU achieves zero-cost context switching by simultaneously holding registers

for each thread in a common register file, and using the thread ID as an offset. Each SM has hardware limits to the number of threads that can be simultaneously loaded and executed: In addition to the register file having finite capacity, each SM can hold only a finite number of threads (1024 on the Nvidia P100) and finite number of blocks (32).

Modern NVidia GPUs issue each instruction to a *warp* of 32 threads simultaneously: All threads in a warp must execute the same instruction each cycle. *Branch divergence* occurs when a conditional branch instruction is issued to a warp and the condition evaluates to a different value for some threads within the warp. In this case, the hardware must first execute the code in the taken path—leaving idle the threads that evaluate the condition to false—and subsequently it must execute the not taken path while leaving idle the threads that evaluate the condition to true. All threads may continue executing when the control flow reconverges at the join point in the flow graph. Such underutilization of processing resources reduces the performance of GPUs. Branch divergence is problematic because stalled threads are still assigned registers and execution slots, preventing other threads from being started to perform useful work.

A warp of threads issues instructions simultaneously, causing as many as 32 simultaneous memory access requests to be issued in one execution cycle. However, the bandwidth available to access a GPU global memory subsystem is limited. Thus, the hardware in a GPU is able to *coalesce* (merge) adjacent or overlapping requests that originated on the same cycle by threads within a warp into fewer requests, each accessing more data. Once a memory access request is issued, no threads in a warp can continue executing until all of the threads have been serviced. Therefore, coalescing multiple memory accesses into fewer requests dramatically improves throughput. However, the hardware is only able to coalesce requests if the threads in a warp are accessing adjacent or overlapping locations in memory, and if the range of accessed addresses is aligned to a cache line boundary. If the execution of a statement leads threads to access memory locations that do not fit within an aligned range of memory addresses, then multiple memory requests are necessary. Therefore,

the execution time is longer than in the case where all the accesses for the warp are coalesced into a single request. Each global memory request requires hundreds of cycles to be completed. Thus, GPU programs should be structured to avoid non-coalescable memory accesses.

All SMs share a common global memory, using local caching to reduce latency. The global memory is also attached to a Direct Memory Access (DMA) controller, which can transfer memory between global GPU memory and CPU memory without requiring either the CPU or GPU SMs for processing. The DMA controller allows memory transfers to and from the GPU to be overlapped with GPU processing.

Despite the increased reliance on GPUs, the structured style of parallel processing that GPUs require makes their performance sensitive to two problems: (1) *branch divergence* [22], in which adjacent threads exhibit different control-flow behaviour causing hardware stalls, and (2) *non-coalesced memory accesses* [8], in which adjacent threads access disparate memory addresses, overloading the underlying memory system with requests.

2.1.2 Field-Programmable Gate Arrays (FPGAs)

FPGAs consist of arrays of interconnected programmable logic blocks, which vary in complexity from simple lookup tables (LUTs) to complete functional units. LUTs can be wired up via programmable interconnects to form arbitrary digital circuits.

Typically, circuit configuration is specified via a Hardware Description Language (HDL) such as Verilog or VHDL. The HDL is then compiled into a 'bitstream' - a configuration file that sets the device's logic blocks and interconnect switches into a desired state. The compilation process consists of placing circuits specified by user HDL code to the chip, while considering chip area usage and interconnect length/congestion. The placed circuits then undergo routing, i.e., adding wires to correctly connect the placed components. Arriving at an optimal circuit configuration is a known NP-hard problem. Synthesis takes from hours to days.

2.2 Accelerator Programming Languages

2.2.1 CUDA

CUDA is a parallel programming API created by Nvidia to allow developers to tightly integrate with their GPUs. Functions written in C/C++/Fortran can be marked for GPU execution and called by CPU (host) code, while the CUDA compiler handles compiling and linking GPU executables.

Parallelism in the CUDA programming model is of a SIMT (Single Instruction Multiple Thread) form. In CUDA, a program is divided into host code and a series of *kernels*. The code for each kernel describes the execution of a single thread, but the programming model assumes that many threads will execute that same kernel code in parallel. Threads are grouped into blocks, and a number of blocks are executed simultaneously. The number of threads per block and the number of blocks are collectively referred to as a grid, and must be specified each time a kernel is executed.

CUDA is meant to enable tight integration with GPUs, and therefore exposes many GPU-specific features through the addition of new keywords and intrinsics. In particular, CUDA exposes the `threadIdx` and `blockIdx` variables, which expose the thread numbers during execution, and allow programmers to create thread-dependent behaviour. Because GPUs execute threads in warps of 32, Nvidia also exposes intrinsics to allow threads within a warp to cooperate at no cost through intrinsic functions like `ballot(c)` which takes a predicate from each thread, and returns a 32-bit binary value combining all of the predicates.

By enabling tight coupling of programs to GPUs, CUDA seeks to enable high-performance GPU computing through deep programmer knowledge and understanding.

2.2.2 OpenMP

OpenMP [18] allows programmers to mark sections of code as parallel without worrying about how such parallelism maps to hardware. However, since high-level programming models like OpenMP abstract the architecture-specific

details of code generation, accelerator offloading functionality exacerbates an age-old problem of such models: compilers must attempt to optimally map parallelism to each targeted architecture.

OpenMP is a prescriptive directive-based programming framework, and an API designed for both shared and distributed memory multiprocessing programming using C, C++, and FORTRAN. It is made up of a collection of compiler directives for controlling execution of a parallel application, library routines for interfacing with the runtime environment, and environment variables. Figure 2.1 shows an example taken from the `gemm` benchmark where the programmer annotated the loop on line 4 with the `parallel for` directive, instructing the compiler that iterations of that loop may be executed in parallel, though the inner loop on line 5 must still be executed serially. The compiler generates code to create worker threads that will execute in parallel and divides a work task among the worker threads. However, it is up to the runtime environment to allocate threads to different processors. This programming model abstracts details of work-sharing and thread coordination and allows programmers to focus on solving the task at hand. Moreover, the high level of abstraction makes OpenMP highly portable, promising, in principle, code scalability from standard workstations to supercomputers. However, because compiler heuristics can be sub-optimal, OpenMP designers included syntax for programmers to provide hints to the compiler to attempt to improve performance for a given target architecture. The OpenMP language specification has accumulated considerable bloat to accommodate such hints. Still, the goal of portability of OpenMP through

```
1  void gemm_OMP
2  (float *A, float *B, float *C) {
3    #pragma omp parallel for
4    for (i = 0; i < NI; i++) {
5      for (j = 0; j < NJ; j++) {
6        C[i*NJ + j] *= BETA;
7        for (k = 0; k < NK; ++k) {
8          C[i*NJ + j] +=
9            ALPHA * A[i*NK + k]
10                 * B[k*NJ + j];
11       }
12     }
13   }
14 }
```

Figure 2.1: OpenMP 3 example modified from `gemm` in the Unibench benchmark suite

more efficient mapping of parallelism to hardware has attracted considerable attention [51], [73].

OpenMP 4.x Target Offloading

OpenMP 4.0, introduced in 2014, allows programmers to specify `target` regions - blocks of code to be executed on an accelerator device present in the system. Variables are mapped to the device data environment and the code generation for parallel constructs enclosed in the `target` region targets the accelerator architecture. Figure 2.2 shows the same parallel segment of the `gemm` benchmark as Figure 2.1, but targeting OpenMP 4.0. The target region is declared on line 17, causing the entire loop nest to be executed on the GPU. Lines 18-19 show the data mapping of the arrays, making them available within the target region. Finally, the outermost loop is made parallel as before on line 20. The additional `teams distribute` clause indicates that these loop iterations should first be divided amongst teams, and then amongst threads within each team.

```
15 void gemm_OMP
16 (float *A, float *B, float *C) {
17   #pragma omp target
18   \    map(to: A[:NI*NK], B[:NK*NJ])
19   \    map(tofrom: C[:NI*NJ])
20   #pragma omp teams distribute
        parallel for
21   for (i = 0; i < NI; i++) {
22     for (j = 0; j < NJ; j++) {
23       C[i*NJ + j] *= BETA;
24       for (k = 0; k < NK; ++k) {
25         C[i*NJ + j] += ALPHA * A[i*NK
             + k]
26                               * B[k*NJ +
                                    j];
27       }
28     }
29   }
30 }
```

Figure 2.2: OpenMP 4 example simplified from `gemm` in the Unibench benchmark suite

2.2.3 OpenCL

OpenCL is an open standard for parallel programming of heterogeneous systems and a programming language specification [26]. General OpenCL program architecture consists of a host device that controls one or multiple com-

pute devices by managing memory transfers and task distribution across devices. Compute devices are split into compute units, which, in turn, contain individual processing elements. OpenCL defines a Same-Instruction Multiple-Thread (SIMT) data-parallel model where many threads execute the same instruction on many data items. In OpenCL terminology this model is called *NDRange* (for N-dimensional range). OpenCL also provides task-level parallelism that exploits concurrency through stand-alone task distribution across different compute units.

The main purpose of OpenCL is to enable portable use of various hardware accelerators. While already popular for GPU accelerators, recent adoption of the framework as an High-Level Synthesis (HLS) input language has opened new opportunities to explore FPGA-specific compiler transformations.

2.3 GPU Divergence Analysis

Automated tools exist for most programming languages to detect common programming mistakes [7], [11], [36]. Such tools make use of static analysis techniques to verify correct program behaviour. We focus this section quite narrowly, highlighting prior work on detecting branch divergence and non-coalescable memory accesses for GPU kernels.

Sampaio et al. [58] propose a conservative divergent-branch analysis over affine expressions on thread identifiers, now implemented within LLVM. An extension to their work generalizes to divergent values throughout a GPU kernel. In contrast, GPUCheck does not require affine expressions, and can solve for non-linear conditions. Moreover, GPUCheck derives divergent expressions from arbitrary control-flow and data dependencies even across interprocedural boundaries.

Transforming non-coalescable memory accesses into coalesced accesses has been an active subject of research. Wu et al. [74] show that given perfect knowledge of memory access layouts, minimizing non-coalesced accesses is an NP-hard problem. However, the authors do not consider what analysis might be used to generate such layouts. GPUCheck is well-suited to perform such

a task. Affine polyhedral models have also been used to transform memory accesses [5], [71]. Venkat et al. [70] have recently extended the polyhedral model to include limited non-affine expressions. While GPUCheck cannot currently perform transformations, it identifies non-coalesced accesses, even when generated indirectly through multiple function invocations. Additionally, GPUCheck calculates memory access patterns simultaneously for data- and control-flow dependent values.

Fauzia et al. [19] present an approach that requires dynamic analysis to identify non-coalesced accesses using memory traces. Their work generates high-accuracy results, but requires any kernel to be executed, suffering from the same issues as any dynamic profiler. On the other hand, GPUCheck runs its analysis without even a GPU present, by calculating inter-thread behaviour.

2.4 Software Compression

Data compression schemes allow for data to be represented with fewer bytes, which is valuable both to reduce storage requirements and to transmit data at higher speeds. However, when considering the transmission of compressed data, one needs to also consider the compression and decompression operations that must occur at each end. More effective compression schemes tend to require more computation, reducing the maximum throughput at which they can operate. Compression schemes therefore exist along a spectrum, where incrementally more computation can result in incrementally smaller compressed data. Over recent years, network and interconnection link speeds have been increasing faster than compute and memory speeds. With faster networks, in order for a compression scheme to be advantageous, it must require less compute resources.

Software compression schemes such as brotli [1], gipfeli [40], lz4 [12] and Snappy [24] achieve impressive compression ratios across a wide sample of data at rates up to hundreds of Mbits. Meanwhile, simple compression schemes such as Base-Delta Intercept (BΔI) achieve limited compression, but are simple enough to be implemented in hardware at rates of up to hundreds of Gbits [54].

Data compression schemes are divided into two categories: *lossy* and *lossless*. Lossy compression schemes are allowed to discard information to reduce file sizes, and are used when exact replication of the compressed data is not necessary. Lossy compression is commonly used, for example, to compress JPEG images and MP3 audio. By contrast, lossless compression schemes must reproduce the exact input through decompression. Compression schemes represent a trade-off between compression ratio and computational requirements.

A *compression ratio* is a measure of the degree to which a compression scheme reduces the size of a piece of data, and is defined as $size_{original}/size_{compressed}$. The reciprocal of the compression ratio is known as *space savings*.

Throughput is a measure of the speed of compression and decompression, and is measured in bytes/sec. Typically compression and decompression rates are reported separately because decompression is usually much faster than compression.

2.4.1 Lossless Compression

Lossless compression is required in many cases, such as when transmitting files in ZIP format, where an inexact reproduction could result in data corruption. In particular, whenever compression is intended to be hidden from users, lossless compression is required. Two well-known lossless compressions are Run-Length Encoding (RLE) and Lempel-Ziv Compression (LZ).

RLE is a simple compression scheme that encodes repeated values using a sentinel value (typically a value repeated twice) followed by a count. These repeated values, known as runs, are thus compressed. Variations exist with regard to the size of each value, and the method of encoding runs. Run-length encoding is extremely simple to implement, and has even been implemented in hardware. Fax machines notably implement RLE for lines of color during transmission over telephone lines. While trivial to implement, RLE is ineffective when the incoming data does not contain long sequences of identical values.

Lempel-Ziv compression uses the intuition that a recently used value is likely to be reused. A sliding window of history is maintained during compres-

sion, and if an encountered value or sequence is present in the history window, it is replaced with a code word in the compressed output. During decompression, the same history window is constructed, such that the code words can be decoded by referencing the window. Many variations on LZ exist, varying the size of values and the size of the history window. Most commonly used today is the LZ4 algorithm [12].

2.4.2 Parallel Compression

Given the general availability of multicore processors, efforts have been made to parallelize software compression algorithms. Parallel compressors tend to either partition the input into segments for separate compression [45], or apply cooperative parallel algorithms. They often achieve sublinear scaling [35], [62].

For GPU parallelism, Sitaridi et al. introduced Gompresso [63], a GPU algorithm for decompression. Gompresso splits an input file into partitions and independently compresses each partition with LZ77, a Lempel-Ziv variant. Gompresso achieved decompression speeds of up to 16 GB/s on an Nvidia Tesla K40 by exploiting GPU parallelism. However, their algorithm sacrifices compression ratios, and does not investigate compression throughput.

Others have attempted to use GPUs to offload portions of a compression algorithm. Shastry et al. use a GPU to perform the Burrows-Wheeler transform in bzip2 compression [61], improving bzip2 throughput by 44% while leaving the CPU idle half of the time, and thus allowing other operations to continue.

2.4.3 Compression in Networking

There is a long history of using compression to improve network throughput. Craft used a variant of LZ compression implemented in hardware to achieve throughputs of up to 1 Gb/s in 1998 [14]. More recently, and more commonly, HTTP connections support compression to dramatically reduce webpage load time [42].

However, modern data centres and supercomputers are connected by fabrics operating at 10-40 Gb/s. Therefore, most software compression schemes

in use today cannot keep up with the volume of incoming or outgoing data.

Chapter 3

Related Work

As this work is a compilation of multiple research projects, related work is divided here by project.

3.1 FPGA High-Level Synthesis

Czajkowski *et al*. present an LLVM-based OpenCL compiler prototype for Altera FPGAs, with a proof of concept executing on the Stratix DE4 [17]. This compiler represents all basic blocks of the program as Control-Data Flow Graphs (CDFG) with their own inputs and outputs as determined through live-variable analysis. The CDFG allows for efficient implementation of a module as a pipelined circuit, as opposed to finite state machine with a datapath - an approach much better suited to the data-parallel OpenCL model. This compiler implements the NDRange execution model by issuing individual work items into a kernel pipeline, one after another. The task-parallelism execution model, where the kernel code is written in a serial fashion, is implemented in the compiler by attempting to pipeline every loop in the code. The compiler also creates a wrapper for the generated kernel circuits to handle the standard interfaces to the device-memory IO and does all necessary bookkeeping to track kernel execution and to issue new work-itemss into the pipeline. This work subsequently became the Intel FPGA OpenCL compiler, upon which our optimizations are based. Internally, the compiler consists of an LLVM-based HLS component that compiles OpenCL kernel code into Verilog, which is then synthesized using standard Intel FPGA Quartus software.

Intel FPGA also provides an implementation of the OpenCL API to allow host code to interface with devices: launch kernels, manage memory transfers, etc.

Figure 4.1 depicts the compilation flow as provided by Intel FPGA. A typical execution workflow consists of the host code programmatically loading a pre-compiled kernel binary file into the FPGA and initiating its execution. Intel FPGA also provides custom extensions to the OpenCL standard enabling certain architecture-specific user optimizations. While the prototype described here uses the Intel FPGA OpenCL toolchain targeting a Stratix V FPGA, the general concepts are applicable to reconfigurable architectures of other FPGA vendors.

As of 2015, Xilinx SDAccel development environment provides a HLS toolchain that is fully compliant with OpenCL 1.0. SDAccel is a closed-source application and little is known about its inner workings. We can infer from the user guide that the data-parallel NDRange execution model in this compiler is emulated through generation of a 3-dimensional loop-nest that iterates over the work-group and work-item dimensions [59]. Loop pipelining is one of the essential optimizations attempted by SDAccel [21]. SDAccel would also provide a good platform to prototype the analyses and transformations.

3.1.1 Manually Optimized OpenCL

Writing OpenCL code that delivers good performance on FPGAs is an open problem. Intel publishes a best-practices guide [31] detailing strategies and patterns that the Intel FPGA OpenCL compiler can efficiently execute. These patterns served as inspiration for the transformations presented in this paper. Zohouri *et al*. [77] performed manual optimizations on six Rodinia OpenCL benchmarks compiled with the Intel FPGA OpenCL compiler. After these optimizations, FPGAs could be competitive with GPUs on performance with dramatically better power efficiency. The effectiveness of these transformations inspired our compiler transformations.

3.1.2 Combined Host/Device Compilation

Lee *et al*. [38] developed an OpenACC-to-FPGA compiler framework, that converts OpenACC programs into OpenCL using the open-source OpenARC compiler, and then uses the Intel FPGA OpenCL compiler for HLS. The approach benefits from the fact that user-level source code contains device kernel code embedded into the host control code. This source code is annotated with pragmas that specify the code blocks that are offloaded to the device. Before the device code is outlined into a separate OpenCL kernel compilation unit, their compiler is able to take advantage of certain code-transformation opportunities that would not have been possible otherwise, such as bypassing global memory for inter-kernel communication using channels. To the best of our knowledge, our work is the first to implement combined compilation on OpenCL source code exposing similar transformation opportunities.

3.2 GPU Parallelism Mapping

Tuning of compile-time and launch-time kernel parameters of GPGPU code has attracted considerable research in light of GPUs' popularity. Vollmer et. al. presented an approach to construct auto-tunable GPU kernels by expressing them in an embedded DSL [72]. The abstractions enforced by the DSL both restrict the parameter search spaces, and allow the use of common search strategies. Lee et. al. developed OpenMPC - an extension to OpenMP for generation of parallel GPU code before OpenMP 4.0 was announced [39]. Their work allowed for auto-tuning of generated CUDA code through a search of the optimization space, but left the number of threads and number of blocks to be specified explicitly by the user through additional pragma directives. A similar work by Grauer-Gray, on GPU code generated from a high-level HMPP language, applied auto-tuning on a large optimization space that targeted GPU kernels [25]. Their approach hard-coded fixed GPU geometries that aimed at maximizing occupancy.

To the best of our knowledge, our work is the first to investigate automatic selection of Grid Geometry in the context of OpenMP GPU kernels.

Multiple applications of automated learning to parallel systems appear in the literature. Wang et. al. use neural networks to map parallelism of OpenMP 3 programs to NUMA computing clusters [73]. In the domain of heterogeneous computing systems; O'Boyle et. al. use a predictive modeling approach to select parallel OpenMP 3 loops for translation to OpenCL GPU kernels [51]. Tournavitis et. al., in the context of auto-parallelizing compilers, use a machine learning model to map parallelism to hardware while profiling, with the goal of maximizing the information collected [67].

Coons et. al. attempt to use instruction-placement heuristic algorithms powered by reinforcement learning technique to reduce communication overhead on Explicit Dataflow Graph Execution processors [13]. While their learning approach produces models that match expert hand-tuned heuristics, it lacks generality. To compensate for the highly-specialized nature of the learned models, they propose a hierarchical approach where units of code are classified into groups that perform well with similar heuristics, using the same learning model for classification. They observe that even when machine learning and data mining techniques are impractical, these methods can still deliver valuable insights to a compiler designer.

3.3 GPU Divergence Analysis

Previous attempts to analyze GPU workloads observe dynamic behaviour on either a simulator [4] or on physical GPUs [8]. These analyses produce precise results, but come at a cost. To benefit from dynamic analysis, the application developer must have access to GPUs with similar characteristics to the target system, or experience substantial overheads to simulate GPU execution.

3.4 High-Speed GPU Compression

Few attempts have been made to implement lossless compression on GPUs, mainly due to the negative effect on compression ratios caused by partitioning. Patel *et al*. implemented a bzip-like cooperative GPU compression algorithm, but found performance was actually slower than the CPU-based bzip2 algo-

rithm [53]. Ozsoy *et al*. implemented a partitioned LZ compression algorithm on GPUs, and achieved a performance throughput 2.2x higher than their parallel CPU algorithm [52]. These works both leave substantial room for a compression algorithm designed for efficiency on a GPU architecture.

Chapter 4

A Case for Better Integration of Host and Target Compilation When Using OpenCL for FPGAs

4.1 Introduction

This work seeks to take OpenCL programs meant for GPU execution, and apply compiler analysis and transformations to enable more efficient FPGA execution.

OpenCL [64] has emerged as a prominent programming model for FPGA. Intel and Xilinx release OpenCL compiler toolchains that support hardware synthesis directly from OpenCL source [17], [21]. GPU-targeted programs rarely achieve acceptable performance when run unmodified on FPGAs [77], so new FPGA-specific compiler techniques and insights are required.

In contrast to the data-parallel model favoured by GPUs, Intel FPGA HLS tools follow a different approach when implementing the NDRange model: synthesized kernels execute instructions in a pipelined fashion, similar to that of an assembly line. In an FPGA, this means that a data processing unit (e.g. a logic block) takes as input the output of a previous data processing unit. These units can perform concurrent computation because their work is independent from each other. The reconfigurable fabric on FPGAs makes SIMT parallelism a poor choice for applications. The pipeline parallelism

model improves utilization by requiring fewer copies of each operator, while maintaining overall throughput [28].

A major feature of OpenCL is separate host and device compilation, allowing OpenCL device vendors to specialize in device-code generation without concern for host implementations. This separation enforces strict Application Programming Interface (API) boundaries between host and device implementations and prevents otherwise trivial compiler optimizations and analyses. As a result, workarounds must be introduced to recover lost performance. For example, Intel FPGA extends the OpenCL specification with *channels* that allow kernel operations to be chained without needing to copy back to memory. If a compiler had access to the combined host and device code, chaining would be a trivial example of loop fusion. However, with kernel definitions compiled separately from invocations, programmers must implement additional APIs to realize the benefits of chaining.

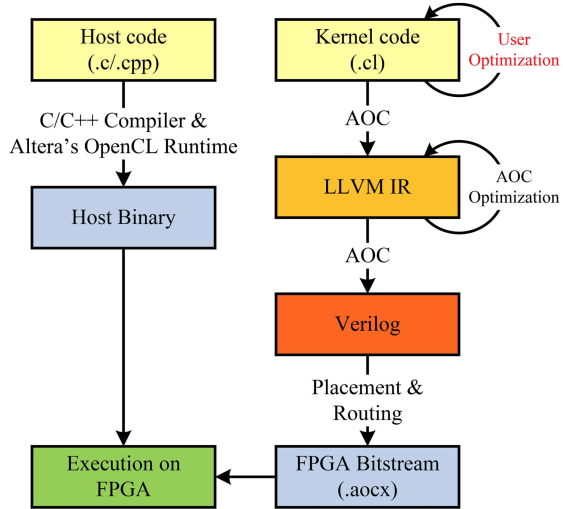


Figure 4.1: Altera OpenCL Compilation Flow. From [17].

This work builds on existing work by Zouhouri *et al*. [77] that analyzes and improves the performance of GPU-targeted OpenCL kernels on Intel FPGA devices using the Intel FPGA SDK for OpenCL. This paper expands compiler analyses to include both host and device compilation and introduces compiler transformations that benefit from sharing analysis information between the host and device compilation. In particular, we define three compiler transformations that transform OpenCL kernels to more closely match the best-practices published by Intel FPGA:

NDRange to Loop: Convert NDRange kernels, originally intended to be

repeatedly executed with a range of thread identifiers (IDs), to a single body of code that uses a loop induction variable to represent thread IDs.

Restrict Parameters: Improve device-side alias analysis by inspecting host code and by marking kernel parameters with the `restrict` attribute where applicable.

While the above two transformations take advantage of the host and device compiler information sharing; they also enable the compiler to optimize the kernel code further by applying single-work-items-specific transformations, such as:

Reduction-Dependency Elimination: Improve pipelining of reduction loops by storing partial sums in a shift register to reduce loop-carried dependencies.

These transformations integrate with the Intel FPGA SDK for OpenCL and are evaluated on the Rodinia benchmark suite [10]. Rodinia benchmark OpenCL implementations target GPU-like devices and, as such, are an appropriate baseline for FPGA-specific transformations.

4.2 Optimizing OpenCL for FPGAs

Intel FPGA maintains a best practices guide for writing OpenCL that will execute efficiently on FPGAs [31]. A subset of these optimizations motivate the remainder of the work, and are summarized here.

4.2.1 `restrict` Pointers to Enable Simultaneous Memory Operations

FPGAs improve performance by executing multiple operations simultaneously. However, memory operations are defined to behave as if performed in program order, and can have extremely long latencies. If two memory accesses never reference the same memory address, then the compiler can safely reorder or overlap the operations. By marking a pointer passed as kernel parameters with

`restrict`, programmers guarantee that any address accessible through that pointer is inaccessible through any other pointer. The Intel FPGA OpenCL compiler can then perform other memory operations simultaneously. As kernel parameters are passed opaquely from the host to the FPGA, it is otherwise extremely difficult for the compiler to prove that memory operations are safe to overlap. Memory operations require hundreds of FPGA cycles, so overlap is required for an efficient pipeline.

4.2.2 Prefer Single-Work-Item kernels over NDRange kernels

In an NDRange kernel, the same computation is executed by a large number of threads to support the data-parallel model. On FPGAs, chip area constraints prevent massively parallel processing units from being constructed. Instead, NDRange kernels are pipelined on FPGAs, allowing the stages to be executed concurrently such that subsequent threads can be started each cycle. All threads are executed in a single shared pipeline and thus values that do not differ between threads can be calculated once, and referenced from within the pipeline. However common intermediate products cannot be expressed in NDRange kernels, so single-work-items kernels are preferred.

Moreover, loops in NDRange kernels would have to be fully unrolled to support efficient pipelined execution because a pipeline is constructed across thread invocations. By converting an NDRange kernel to a single work item, loop exchange can generate pipelines where not otherwise possible. This conversion can also enable new transformations such as shift register reduction (described next). The effectiveness of the pipelined execution model depends on the target algorithm. Algorithms with little synchronization or control-flow may not benefit from single work-items execution at all and will have better performance with NDRange kernels.

4.2.3 Pipelining Reduction Operations with Shifting Arrays

The performance of the single work-item execution model depends on the ability to pipeline loops in the kernel code. Thus, removing loop-carried dependencies is especially important because such dependencies induce longer loop initiation intervals. Reduction operations, such as the double_add_1 method shown in Figure 4.2, cannot be pipelined well because the intermediate value temp_sum must be computed for each iteration before the next iteration can begin. Floating-point operations are relatively slow, causing the FPGA to stall for the majority of the computation. Addressing a similar problem in the context of software loop pipelining in 1992, Rau *et al.* [55] first introduced a technique called *modulo scheduling* that employs a rotating register file as a means to achieve a more compact loop schedule and thus reduces the loop initiation time. This technique was later implemented in hardware in the Intel IA-64 architecture [60].

```
1 __kernel
2 void double_add_1(__global double *arr, int
      N,
3                  __global double *result)
4 {
5   double temp_sum = 0;
6   for (int i = 0; i < N; ++i)
7     temp_sum += arr[i];
8   *result = temp_sum;
9 }
```

Figure 4.2: Floating-point reduction sample preventing loop pipelining. From [31].

As suggested in the best-practices guide [31] and implemented by hand by Zouhouri *et al.* [77], a variation of the rotating register technique can be employed by the programmer manually to minimize pipeline delays caused by the intermediate value in a reduction operation. Figure 4.3 shows the same reduction, but with the introduction of a local array to emulate a shift register. Instead of reducing elements of arr into a single variable, they are accumulated into a shift register. The shift register's depth is equal to the latency, in cycles, of the floating-point operations that form the dependency. Reduction input is read from the first element of the shift register, and written into the last. Effectively, this reduces the initiation interval of the loop to 1 cycle. After the loop completes, an extra reduction on the shift register contents produces the final reduction value. Because the final shift register summation loop has a smaller trip count, the improved initiation interval of the original loop yields an overall performance improvement. The Intel FPGA

OpenCL compiler looks for the pipelining idiom from Figure 4.3 in OpenCL code, and efficiently implements it using a shift register in hardware.

4.3 Compiling OpenCL for FPGAs

The transformations suggested in the Intel FPGA best-practices guide are meant to be performed by programmers. However, a sufficiently capable compiler should be able to automatically transform OpenCL device code to deliver more efficient execution on FPGAs. Thus, we integrated some of the transformations into the Intel FPGA SDK for OpenCL compiler. This compiler is a closed-source application based on the LLVM compiler infrastructure. Thus, arbitrary compiler passes targeting LLVM 3.0 can modify the Intermediate Representation (IR) .

```
1  __kernel
2  void double_add_2(__global double *arr, int
       N,
3                 __global double *result)
4  {
5    double shift_reg[II_CYCLES+1];
6    //Initialize all elements of shift_reg to
         0
7    for(int i = 0; i < N; ++i)
8    {
9      shift_reg[II_CYCLES] = shift_reg[0]+arr[
           i];
10     #pragma unroll
11     for(int j = 0; j < II_CYCLES; ++j)
12       shift_reg[j] = shift_reg[j+1];
13   }
14   double temp_sum = 0;
15
16   #pragma unroll
17   for(int i = 0; i < II_CYCLES; ++i)
18     temp_sum += shift_reg[i];
19   *result = temp_sum;
20 }
```

Figure 4.3: Floating-point reduction using a shift register to enable loop pipelining. From [31].

Our transformations are performed early in the compilation process because they attempt to automate best practices when writing source code. OpenCL is designed to allow for the separate compilation of host and device code. However, combined compilation allows for optimizations not previously possible. Moreover, the Intel FPGA compiler already requires some degree of such coordination by the user. For instance, the compiler may generate single-work-item code for a kernel that the host invoked in NDRange mode. Our compiler passes make use of coordination between the host and device code compilation processes, passing information between the two to enable certain transformations. A custom compiler driver facilitates combined compilation, accepting as input the host and device source-code files. The driver

```
1  size_t work_dim = 2;
2  size_t gbl_offset[2] = {0, 0};
3  size_t gbl_size[2] = {64, 8};
4  size_t lcl_size[2] = {32, 1};
5  clEnqueueNDRangeKernel(
6          cmd_queue, kernel, work_dim,
7          gbl_offset, gbl_size, lcl_size,
8          wait_list_size, wait_list, event);
```

Figure 4.5: Host NDRange kernel invocation

coordinates between the Intel FPGA compiler modified with our transformations and the host compiler, based on LLVM 4, with modifications to analyze and transform host-to-device communication. To integrate our three transformations with the Intel FPGA OpenCL Compiler, the host code analyses are choreographed with the device code transformations (Figure 4.4).

4.3.1 NDRange to Single Work-Item Loop (NDRangeToLoop)

Under the OpenCL NDRange model, a kernel function is invoked for a number of threads (work-itemss) that can cooperate and synchronize within a work-group. This model maps extremely well to GPUs, which have many Single Instruction, Multiple Data (SIMD) processors, but makes it difficult for kernel functions to express common work products. On FPGAs, where parallel kernels are implemented using pipelines, factoring out common work is key to improving performance. OpenCL allows thread and work-group sizes to be specified in three dimensions, denoted here as Z, Y, and X. The conversion of a NDRange kernel into a single work-items, transforms the kernel body into a series of loops over the respective n dimensions. Each loop executes the original kernel body for each thread. Thread ID references are remapped to the appropriate loop induction variable. Unfortunately, however, the number of dimensions, size, and count of work-groups are specified in host code and are inaccessible to device compilation. An example host invocation is shown in Figure 4.5.

Thus it is necessary to recover the number of dimensions, the starting indices of threads in each dimension, the number of threads in each dimension,

and the number of threads per work-group in each dimension. To do so we created a host transformation that injects dummy functions that take as argument each value of interest. After this transformation, standard LLVM passes for inter-procedural constant propagation are applied. Calls to the dummy function are inspected, and constant arguments are transmitted to the device compiler. This technique can be easily extended to share arbitrary constants and ranges for device kernel parameters, allowing additional device code specialization. For generality, the kernel function signature is first modified to take as argument the `work_dim`, `global_work_offset`, `global_work_size`s and `local_work_size` values. Then, the results of the host NDRange invocation analysis are read, and used in place of the kernel arguments when available.

As long as a kernel contains no synchronization points, which can be verified by checking the kernel for calls to the OpenCL `barrier()` function, the NDRange execution can be emulated through a single loop nest. To emulate work-groups, the kernel body is wrapped in loops corresponding to any dimensions for which `get_group_id()` is accessed. Calls to `get_group_id()` are then replaced with accesses to the appropriate loop induction variables. Next, loops are inserted to emulate work-itemss within each work-group, replacing accesses to `get_local_id()` and `get_global_id()` with the appropriate expressions on the loop induction variables. Finally, accesses to the invocation dimensions are replaced. As an example, a Hello World kernel is shown in Figure 4.6. After applying the transformations above, the kernel appears as depicted in Figure 4.7. Several optimizations can be performed at this point. Single-iteration loops can be elided entirely, and dead code elimination can remove the computation of unnecessary values. By applying these optimizations, the single-work-items example can be simplified to Figure 4.8.

In NDRange kernels, work-itemss must halt execution at barrier points until all other work-itemss in the work-group reach the same point. This behavior can be preserved after NDRangeToLoop transformation by splitting work-items loop nests at each barrier point. Our prototype NDRangeToLoop transformation can only convert NDRange kernels with barriers in top-level control flow, outside of any conditional statements or loops. When such bar-

```
1  __kernel void hello_world
2      (int tid) {
3
4
5
6
7
8
9
10
11
12
13   unsigned thread_id =
14     get_global_id(0);
15   if (thread_id == tid) {
16     printf("tid #%u\n", tid);
17   }
18
19
20 }
```

Figure 4.6: HelloWorld Kernel. From [2].

```
1  __kernel void hello_world
2      (int tid,
3      int offset_x, int offset_y,
4      int offset_z,
5      int global_x, int global_y,
6      int global_z,
7      int local_x, int local_y,
8      int local_z) {
9    int group_sz_x =
10     (global_x-1) / local_x+1;
11   for (int c=0; c<group_sz_x; c++) {
12     for (int f=0; f<local_x; f++) {
13       unsigned thread_id =
14         (c*local_x + offset_x)+f;
15       if (thread_id == tid) {
16         printf("tid #%u\n", tid);
17       }
18     }
19   }
20 }
```

Figure 4.7: After NDRange Transformation [2].

```
1  __kernel void hello_world
2      (int tid,
3      int offset_x, int offset_y,
4      int offset_z,
5      int global_x, int global_y,
6      int global_z,
7      int local_x, int local_y,
8      int local_z) {
9
10
11   for (int c = 0; c < 2 ; c++) {
12     for (int f = 0; f < 32; f++) {
13       unsigned thread_id =
14         (c * 32) + f;
15       if (thread_id == tid) {
16         printf("tid #%u\n", tid);
17       }
18     }
19   }
20 }
```

Figure 4.8: After Constant Propagation [2].

riers are encountered, the kernel is partitioned into unsynchronized sections, and each section is wrapped separately into work-items loop nests.

If values are calculated before and used after a barrier, they must be preserved across work-items loops. These values are identified by inspecting instruction operands, and then collecting operands calculated in a different partition than the user. Local arrays equal in size to the work-group are allocated, and each operand is saved into the array as it is calculated. Then, uses in other partitions are redirected to the allocated array.

After transforming the NDRange kernel to a single work-items kernel, it is necessary to transform the kernel invocation on the host. To invoke transformed device kernels appropriately, each `clEnqueueNDRangeKernel()` invocation is replaced with the following routine:

1. Ensure that the kernel about to be executed is one which was transformed. If not, invoke with the original call to `clEnqueueNDRangeKernel()`
2. Pass the original dimensions and work-group sizes and counts as arguments through `clSetKernelArg()`.
3. Invoke the kernel with `clEnqueueTaskKernel()` for single work-items execution.

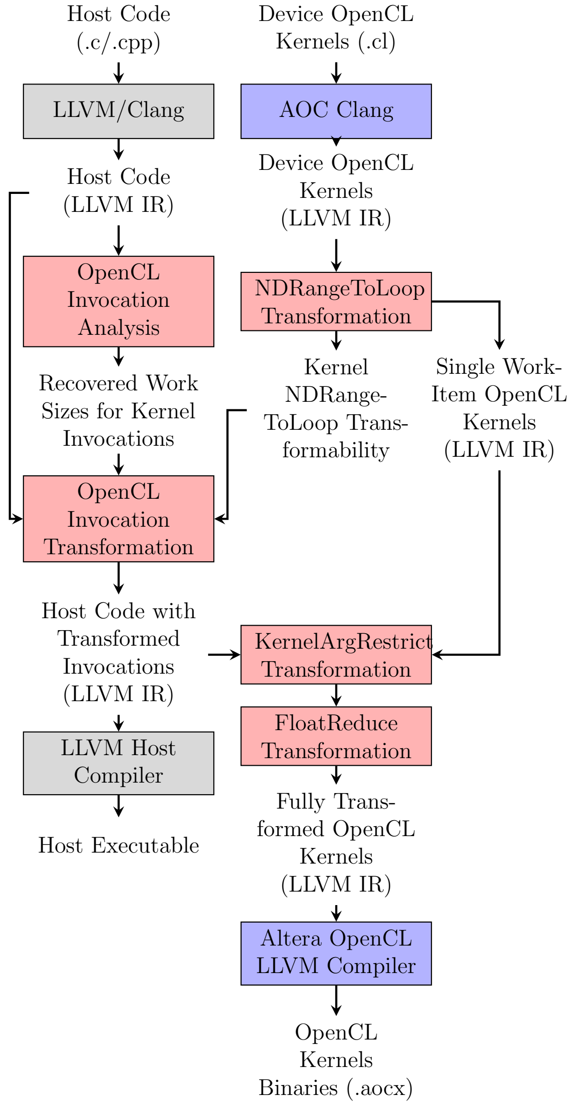


Figure 4.4: Custom OpenCL Compilation Flow

The prototype implementation of the NDRange transformation described and evaluated in this work has some limitations. Its NDRangeToLoop cannot handle barriers inside control-flow. Also, the work-group size must be known at compile-time to enable the allocation of the array to allow uses across partitions.

4.3.2 Reduction-Dependence Elimination

The floating-point reduction-dependence elimination transformation implements an idiom suggested by the Intel FPGA Best Practices Guide [31] as a technique to remove loop-carried dependencies by inferring shift registers for loops that carry out floating-point reductions, as demonstrated by going from Figure 4.2 to Figure 4.3.

First, an analysis detects all reduction idioms that are safe to transform. All loops that do not contain other loops are scanned for reduction expressions. A reduction expression a store to a value where an operand of the stored value is obtained from a load from the same address. The pattern-matcher handles two cases: when the reduction value is accessed through a pointer with no

offset, and when the reduction value is a memory location specified via a base address and an indexing expression. The latter case requires the use of exactly the same indexing expression for both the store and the corresponding load. Once a reduction expression is found, the analysis must verify if it is safe and beneficial to apply the transformation. To do so, the analysis performs the following checks:

- The type of the reduction value must be either 32-bit or 64-bit floating-point.
- The reduction value must not be used elsewhere in the loop body other than in the reduction operation.
- If reduction is done on an array element, the indexing expression and the array base pointer must be loop invariant.
- The binary operations that constitute the reduction must be associative and commutative.
- If the loop trip count is known at compilation time and is less than the shift register size, the reduction should not be transformed.

The final loop trip-count check warrants further explanation: calculation of the final reduction value requires the computation of the sum of the values in the shift register, which is a loop with exactly the same type of loop-carry dependency that was eliminated in the transformed reduction loop. This summation loop has a trip count equal to the number of elements of the shift register. Thus, the transformation is only beneficial when the number of original reduction iterations exceeds the size of the shift register, which is a compiler-specified constant suitable to the target FPGA. The shift register must have enough elements to cover the latency of floating-point operations that would prevent pipelining. In the prototype, targeting the Intel FPGA Stratix V, this constant is eight, which is the floating-point operation latency for the device.

Code generation consists of the following steps: a shift register array is created for a given reduction operation. All its values are initialized to zero in the loop pre-header. The original reduction statement is then rewritten to one that instead performs a store into the shift register's tail element. Immediately after the reduction expression, the values of the shift register are shifted down.

Benchmark	**Original Execution (s)**	**Transformed Execution (s)**	**Ratio**	**Restrict**	**NDtoL**	**FPReduce**
`gaussian`	0.28	1.85	6.69	✘	✔	✔
`hotspot3D`	9.65	25.12	2.60	✔	✔	✘
`kmeans`	37.52	13.43	0.36	✔	✔	✘*
`nn`	0.05	0.05	0.98	✘	✔	✘
`srad`	105.50	111.70	1.06	✔	✔	✘

Table 4.1: Benchmark Execution Time and Applicable Transformations

In the loop epilog, the final reduction value is computed by performing a sum over all shift register values and is stored into the original intended reduction value.

4.3.3 Restrict Pointer Kernel Parameters

When creating a buffer with the OpenCL API `clCreateBuffer()` function a programmer can use the `CL_MEM_USE_HOST_PTR` flag to indicate that the buffer should use memory referenced by the host [34]. Thus, the prototype assumes that in files that do not contain the `CL_MEM_USE_HOST_PTR` flag there are no overlapping buffers. Based on this assumption the prototype marks all global pointers as `restrict` for kernels in these files. In a refinement to this approach, the compiler would first mark all buffers that are not allocated in the host memory and then exclude only the kernels that use more than one such buffer from having their parameters marked `restrict`.

4.4 Prototype Performance Study

Several unmodified OpenCL kernels from the Rodinia benchmark suite form the baseline for a study of the prototype performance. To generate transformed kernels all the transformations described in this paper are enabled unless otherwise specified. In both the baseline and transformed benchmarks, the host code that loads and launches kernels had to be hand-modified to load kernels from FPGA-synthesized binaries, rather than compile them from source at runtime. We evaluate only benchmarks that NDRangeToLoop can trans-

form, limiting ourselves to `gaussian`, `hotspot3D`, `kmeans`, `nn`, and `SRAD`. The remaining benchmarks either fail to compile under the Intel FPGA OpenCL compiler, or are unaltered by our transformations. Out of the benchmarks tested, pointer `restriction` was applied to `hotspot3D`, `kmeans`, and `SRAD`. Reduction-dependence elimination applies to the `gaussian` and `kmeans` benchmarks.

Performance was evaluated on a Terasic DE5-Net board that contains an Altera Stratix V GX FPGA with 4GB of 1600 MHz DDR3 memory. The board is connected to a machine with an Intel Core i7-4770 CPU with 32GB of DDR3 memory, running CentOS 6.7 (Linux 2.6.32). We use the Intel FPGA SDK for OpenCL version 16.1.0.196. Our transformations on the device code are implemented against the SDK-compatible LLVM 3.0, while the host code transformations and analyses are implemented with the LLVM 4 compiler.

Each benchmark was run ten times for both the baseline and the transformed versions with the mean overall execution time reported. Minimal variance was observed between runs of a given program, never exceeding 0.5% of the mean. Thus the variance is not reported.

4.4.1 Benchmarks

Table 4.1 shows transformed kernel execution time and ratio over untransformed kernels. The five benchmarks handled by this prototype implementation can be divided into three groups according to the performance in relation to the baseline. For `gaussian` and `hotspot3d`, the transformed code is significantly slower than the baseline. `nn` and `srad` have roughly baseline performance; and `kmeans` is 2.6× faster than the baseline.

`gaussian`

This kernel contains a loop with a memory dependence on load operations. Before our transformations, the performance impact of the memory dependence is mitigated because multiple work-items can be simultaneously executed. After NDRangeToLoop however, the pipelining opportunities are obscured, because our analysis is unable to determine that the various kernel parameters are

independent. Though the introduced loops can be pipelined, the load and store operations must be executed in order to preserve semantics, and this effect cannot be mitigated by operator duplication as before our transformation. Performance could be improved either by not applying NDRangeToLoop kernel parameters are not marked restrict, or by exposing heuristics from the underlying compiler.

`hotspot3D`

This kernel contains a loop-carried dependence. Some loops are therefore not pipelined, which degrades the overall performance. Additional heuristics that measure overall kernel capacity for pipelining across all loops would be useful to decide when NDRangeToLoop transformation would be beneficial.

`nn` and `srad`:

Algorithms with little control-flow may not benefit from single work-items execution. In these benchmarks, the kernels transformed contained at most two conditional branches. As a result, both the baseline and transformed benchmarks can issue a new thread each cycle.

`kmeans`

Both the NDRangeToLoop and Restrict transformations are applied to this kernel. One of the kernels in `kmeans` contains a nested loop. Performance improves dramatically because this nested loop can be fully pipelined only with the NDRangeToLoop transformation.

Reduction-dependence elimination for `kmeans` was disabled because it degraded performance. The reduction loop nests in a loop that is already fully-pipelined, and the resulting improved reduction loop initiation interval means that a new loop iteration is dispatched every cycle for this and the outer loop. As a result, the loop induction variable increment and the comparison between the IV and the loop upper bound, together, form 87% of the kernel critical path (according to the Intel compiler optimization report). In turn, the FPGA is forced to reduce the operating frequency to accommodate the

number of integer operations that must be performed simultaneously on the induction variables of both loops in the loop nest. There is no way for this prototype implementation to perform this kind of analysis without access to the Intel compiler's internals that perform loop pipelining. However, with access to the pipelining code, an analysis could be added to the transformation that would detect these conditions and deem the transformation unprofitable.

4.4.2 Reduction-Dependence Elimination Efficacy

To measure the impact of the floating-point reduction dependence elimination as a stand-alone transformation, we evaluated several applications where the transformation finds opportunities to apply this transformation. For applications where the transformation analysis finds no opportunities, the code is left untouched so there is no impact. The following data points are not general because few single work-items example kernels are available for evaluation; rather, they serve as motivating examples of the kinds of gains that are possible. The data is not presented in an aggregated fashion because the transformed code varies in source: some are hand-written kernels taken from [77], some are Rodinia benchmark kernels transformed using the prototype toolchain.

The single work-items version of the `srad` benchmark, taken from [77] (the baseline single work-items implementation), which has a frequently executed reduction kernel, sees a 2.6× decrease in overall kernel execution time with the transformation applied.

A hand-written single work-items version of the `lud` benchmark, from [77], with the transformation applied only sees a 7% improvement in kernel execution time. Despite catching several opportunities in a hot region of the kernel code, the impact is small because the reduction loop has a varying trip count that depends on the iteration of the containing loop. As a result, some reductions are smaller than the size of the shift-register and some are larger. The effect is that these two cases cancel each other out, yielding a marginal overall improvement. A classical solution to this pattern would be to version the loop into a portion with a small trip-count that maintains the original reduction

pattern, and a portion with a sufficiently large trip-count that is transformed. However, versioning is costly for FPGAs because it results in higher resource utilization. In our experiments, loop versioning yielded no benefit.

The `gaussian` benchmark, after our NDRangeToLoop transformation, executes 3.3× faster with the reduction transformation applied. One of the two kernels in this benchmark consists of a single reduction operation on an array element. After NDRangeToLoop transformation this reduction becomes a single hot loop.

The only case where eliminating reduction dependence leads to performance degradation is in the `kmeans` benchmark, as described in section 4.4.1. That scenario appears to be anomalous. However a more robust implementation of the transformations should include a more thorough profitability analysis. Such analysis would be best implemented with full access to the source code for the entire software toolchain.

4.5 Concluding Remarks

Combining compilation of device kernel code and host code these compilation paths allows for inter-compiler communication which, in turn, enables new, previously impossible, compiler transformation opportunities. We have implemented three transformations for OpenCL execution on FPGAs using the combined compilation toolchain and studied their performance. The variable performance across benchmarks indicates that more analysis is required to determine when transformations should be applied. A sophisticated analysis could prevent transformations from occurring when it would be unprofitable; more specifically, having access to the loop pipelining code of the Intel FPGA compiler would allow for the application of the NDRangeToLoop transformation only when the kernel can be pipelined successfully. Access to such code and analyses would also help with the issue encountered when applying the reduction-dependency elimination transformation to the `kmeans` benchmark.

This work is a step forward in the automatic optimization of OpenCL applications for FPGA execution. While these transformations were only tested on

a small number of benchmarks, and experienced varying levels of success, the performance improvements seen show that these techniques, when applied judiciously, can dramatically improve program performance without programmer involvement. Such automatic transformations will play a key role in continued FPGA adoption, as specialized device knowledge can be further reduced and performance of generic OpenCL programs made truly portable.

Chapter 5

Automated GPU Grid Geometry Selection for OpenMP Kernels

5.1 Introduction

This work presents two approaches for automatically selecting a GPU grid geometry for an OpenMP target region, a machine learning model and a hand-tuned heuristic.

The selection of the grid geometry needed to map target regions to GPU hardware is a problem that is intuitively well-suited to be solved by a predictive-modelling approach where a collection of static and dynamic features of the GPU kernel is extracted from the `target` region, and a predictor can be constructed to output the grid geometry that results in good performance. Our approach starts with an extensive experimental characterization of the relationship between grid geometry and performance for the set of all compilable C/C++ OpenMP benchmarks in the SPEC ACCEL [33] and Unibench [57] benchmark suites. This search revealed that the performance of an individual benchmark can be improved by up to 9.8 times, as shown in Figure 5.1. The geometric mean across all benchmarks indicated that a 36% improvement over the LLVM grid geometry selection mechanism was possible for this set of benchmarks if a near-optimal grid geometry could be always selected.

The next step was to use various machine learning techniques to attempt to model the performance of GPU kernels extracted from OpenMP `target`

regions as a function of static and dynamic program features and of the grid geometry. Random decision forests, the most successful of our attempted techniques, produced significant speedups for some benchmarks. The grid geometry predictions using random-forest resulted in a geometric mean speedup of 5% relative to the LLVM selection.

Despite their relative success, a closer examination of the ML predictors revealed some serious limitations in their applicability to production compiler-runtime systems. The predictor must perform feature evaluation at program runtime, immediately before kernel launch, because many of the features are runtime characteristics of parallel loops inside `target` regions. Therefore, the time needed to evaluate features and run the predictor must be added to the overall kernel execution time. This prediction overhead negated all improvement achieved by the use of more efficient grid geometry [1].

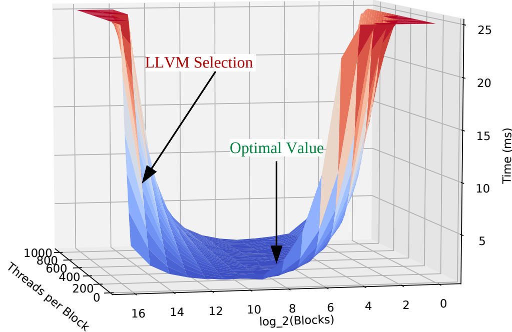


Figure 5.1: Execution time of kernel 21, as a function of the number and size of blocks. The best discovered point is 9.8x faster than the LLVM selection.

Nonetheless, the success of the random-forest predictor in generating more performant grid geometries indicates that the predictor was able to discover relationships between features, grid geometry and performance. The next step was to discover an approach that could capture the relationships discovered by the predictive model with a low-overhead heuristic. Careful examination of the grid geometry space, coupled with a detailed analysis of the predictions produced by the random-forest model, led us to discover a relatively simple, effective, and inexpensive heuristic that can be used in an industrial-strength compiler to select grid geometry for GPU-intended `target` regions. This new

[1] Preliminary results presented by other researchers at workshops indicate that this problem is more general: the run-time prediction overhead is likely to be a limiting issue that will have to be addressed in order to make the use of machine learning for code generation viable.

heuristic has the virtue of being simple, but its discovery had eluded several experienced compiler designers both in industry and in the open-source compiler community.

This new low-overhead heuristic led to speedups of up to 7x over the LLVM selection, with a geomean performance improvement of 25.9% for the entire set of benchmarks across SPEC ACCEL and Unibench.

This chapter makes the following contributions:

1. An exhaustive characterization across the space of possible grid geometries on a variety of benchmarks in order to understand grid geometry effects on performance.

2. An analysis of the efficacy of existing heuristics in the OpenMP 4.x implementation for LLVM/Clang, by comparing to an exhaustive search over possible grid geometries.

3. Description of a methodology to apply machine learning to the problem of grid geometry selection. The result is substantially improved kernel execution time, but impractical prediction overhead.

4. A low-overhead heuristic suitable for production compilers with superior performance compared to the best machine-learning prediction model investigated.

5.2 Mapping OpenMP to GPUs

This section details hardware limitations of a modern Nvidia GPU, and the original techniques used by LLVM/Clang to map OpenMP target regions to GPUs.

5.2.1 Nvidia P100 Geometry

The Nvidia P100 is a modern Nvidia GPU for high-performance computing composed of 56 streaming multiprocessors (SMs), each can issue an instruction for 64 threads in each cycle [48]. The P100 has enough registers to maintain

the state of 1024 threads simultaneously, and thus each SM can context switch between threads at no cost and does so each cycle to hide instruction latency.

When assigning blocks of threads to SMs, there are a number of resource limitations that must be obeyed by the GPU. An SM can only simultaneously execute up to 32 blocks, and only up to 1024 threads. Even then, all of the threads can use only up to a total of 64k registers, and the total usage of shared memory must be less than 64KB. If any of these limits is reached, then the SM cannot accept any more blocks. Blocks that are not accepted must be queued. There is, therefore, a trade-off for the selection of the number of threads per block. A block with more threads can perform more cooperative work, and is better able to hide latency because there are more threads available for execution. However, a block with fewer threads is much more likely to fit with other blocks to share a SM without leading to queuing.

There is also a second trade-off to consider because queuing blocks for later execution on an SM takes a non-zero amount of time. The trade-off is in the determination of how much work to schedule per thread after the number of threads per block is determined: scheduling more work per thread limits parallelism but also reduces the scheduling overhead.

5.2.2 OpenMP 4 in LLVM/Clang

As implemented in OpenMP 4 for LLVM/Clang, OpenMP `target` regions are first outlined into separate procedures, which are then cloned to create a procedure destined for code generation for the device. The device procedure then undergoes construct-specific code generation that transforms user code into a kernel suitable for execution on the GPU. The host code that previously contained the target region is modified to invoke the outlined device procedure as a GPU kernel, including queuing data-transfer, etc. The code-generation process takes the GPU architecture into account, generating data-parallel code in place of parallel loops, where applicable. The resulting kernel is translated into *Parallel Thread Execution*(PTX) - a pseudo-assembly language, designed by NVIDIA to abstract low-level code generated by compilers from the specifics of the hardware implementation. The translation from LLVM-IR to PTX is

done using Nvidia PTXAS - Nvidia's proprietary assembler. The GPU driver finally compiles and executes PTX at runtime.

Adding an extra dimension to the problem of mapping user OpenMP 4.x programs to GPU code is the GPU Execution model's heavy reliance on an efficient grid geometry - the choice of the size and number of blocks that should be spawned for the given `target` region. Unlike the lower-levels of abstraction provided by programming models like CUDA and OpenCL, OpenMP does not require the programmer to select the number of threads and blocks when defining a target region. The standard does includes a *teams* construct to specify parallel execution across leagues of threads that maps well onto GPU thread blocks. However, the number of teams selected, as well as the number of threads, is up to the compiler implementation. The programmer is only able to specify an upper bound for teams using the `num_teams` clause and an upper bound for threads using the `thread_limit` clause. The OpenMP 4 for LLVM/Clang implementation currently implements the runtime to fix 128 threads per block, and select the number of blocks to be equal to the number of parallel loop iterations divided by threads per block. This runtime is used in at least two production compilers and represents the current state-of-the-art.

5.3 Data Collection

To understand how GPU grid geometry affects the performance of OpenMP 4.x programs, we gather performance data at various grid geometry configurations. We evaluate the compiler's current heuristic against this data, and analyze the data for trends that may lead to a better approach. Performance data can also be combined with features extracted statically and dynamically from kernels to produce a dataset for training machine learning models.

We examine 23 kernels taken from 11 OpenMP C/C++ benchmarks. Benchmark applications are taken from the SPEC ACCEL [33] and Unibench [57] benchmark suites. Data collection was conducted on a workstation machine with an Intel i7-4770 CPU with 32GB of RAM, running CentOS 6.7. To collect kernel execution times, we use the CUDA 8 drivers and runtime and a Nvidia

Titan X Pascal with a locked clock frequency. Reported times are averaged over 5 executions. On each execution, for each of the possible grid geometry configurations, all benchmarks are executed in an interleaved fashion, before switching to the next grid configuration. The Clang compiler used for this research currently supports portions of the 4.5 specification. The backwards-compatibility-breaking features of 4.5 have no impact on our experiments because the benchmarks did not make use of them.

The grid geometry space can be represented as a (t, b) tuple, where t is the number of threads per block and b is the number of blocks. Due to architectural constraints, t can vary between 1 and 1024 threads, while b can vary between 1 and 2^{16} blocks. These ranges yield $2^{16} \times 2^{10} = 2^{26}$ possible combinations, which are far too many to actually execute. However, because warps are executed simultaneously, it makes little sense to execute with a number of threads that does not perfectly fill a warp. Therefore, the search space can be limited to $2^{16} \times 2^{5} = 2^{21}$ combinations, which is still too large. Trading accuracy for execution time, instead of testing every block count, we test only powers of 2, yielding $17 \times 32 = 544$ total combinations per benchmark. This approach makes a trade-off: potentially missing the true optimal grid geometry in between the test points to complete the search in a reasonable amount of time.

Accurately collecting runtime data for these benchmarks is surprisingly challenging. For instance, the GPU aggressively scales clock speed to maintain performance without overheating, making it difficult to compare results. Nvidia provides utilities to lock the frequency, but these get automatically disabled after a period of time whenever the driver is idle. Compounding this problem, we noticed substantial uptime effects wherein kernels take longer to execute after the computer has been running for several days. We were unable to find the cause of this problem. The solution was to re-start the computer immediately before data collection began.

The last challenge for data collection was the sheer amount of time required. Locating near-optimal geometry involves spending many cycles executing benchmarks using extremely poor configurations, resulting in each data

collection taking more than a week. When combined with the frequency and uptime effects above, it took four attempts to collect accurate and actionable data.

5.3.1 Best Discovered Grid Geometry Performance Relative to Compiler Default

An analysis of the collected data allows for a comparison with the simple heuristic currently used in compilers to determine the limits in the potential speedup that a change to this heuristic could yield. Figure 5.2 shows the percentage improvement for each benchmark, over the LLVM selection, when the best discovered grid geometry is used. Speedups are presented on a log scale, to present equivalent speedups and slowdowns as equally-sized bars. Choosing a better grid geometry can yield up to 9.8x improvement, with a geomean of 36% potential improvement across all tested benchmarks. The negative improvement seen in a few of the benchmarks is an artifact of the coarse-grained search used to reduce the data collection time: the course-grain search simply did not test a grid geometry that was as performant as the one currently chosen by the compiler. In these cases the performance difference is marginal (within $\approx 5\%$).

5.3.2 Threads Per Block

A relevant question is: *Is there an optimal number of threads per block?* A way to answer this question is to plot the execution time for a range of threads-per-block values for the programs available for the study and then to select the best value. However, this approach has the drawback that it would be using *the same* set of programs both to set a parameter and then to measure performance. To avoid this methodological flaw, the exploration for the best thread-per-block parameter uses a leave-one-out strategy: each fold is formed by 22 of the 23 benchmarks. The execution time for each program for all combinations of number of blocks and threads-per-block for each benchmark is determined by exploration. Figure 5.3 plots for each fold (identified by the benchmark left out in the legend) the ratio between the sum of the

minimum execution time for the benchmarks in the fold for a given threads-per-block value and the sum of the minimum overall execution time. Thus Figure 5.3 plots the overhead of using a given threads-per-block value over the best execution time found by the search. For most benchmarks, the overhead is minimized at either 64 or 96 threads per block, with performance degrading in both directions. Kernel 18 is a notable outlier that requires explanation.

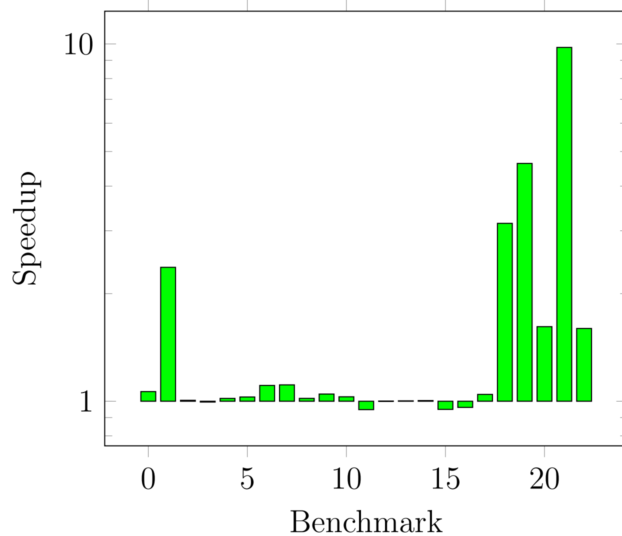


Figure 5.2: Improvement available with the best discovered grid geometry versus the LLVM selection.

Based on the results from this exploration, the number of threads per block to be used to evaluate benchmark 6 should be 64, for benchmark 18, it should be 32, and for all other benchmarks 96 threads per blocks should be used. These numbers ensure that information from the benchmark used for evaluation is not used to obtain the prediction. However, to obtain a model to find the number of blocks (Section5.3.3) a single value for the number of threads per block is used, and the most common one from Figure 5.3 is 96.

SIMD Operations in Kernel 18: Kernel 18 makes use of the `omp simd` directive, marking a loop to be parallelized using SIMD units within each thread. Current GPUs do not have SIMD units, so compilers currently attempt to emulate SIMD operations. If a target region contains a SIMD clause, the code generator used by Clang currently sets aside a warp of threads to be used as a large SIMD unit. When each GPU thread reaches the `omp simd` loop, it waits for the SIMD warp to execute the operations, then resumes execution. Because of the dedicated warp, Kernel 18 is penalized for running with 32 threads per warp. When Kernel 18 is left out, the remaining benchmarks

performance is minimized at 32 threads per block, because no block-level co-operation is required.

5.3.3 Number of Blocks

Next the model must predict the total number of blocks. Figure 5.4 shows the overhead of forcing 96 threads per block and varying the block count for each kernel. Each kernel has a distinct minima, showing that some factor that varies by kernel affects the ideal block count.

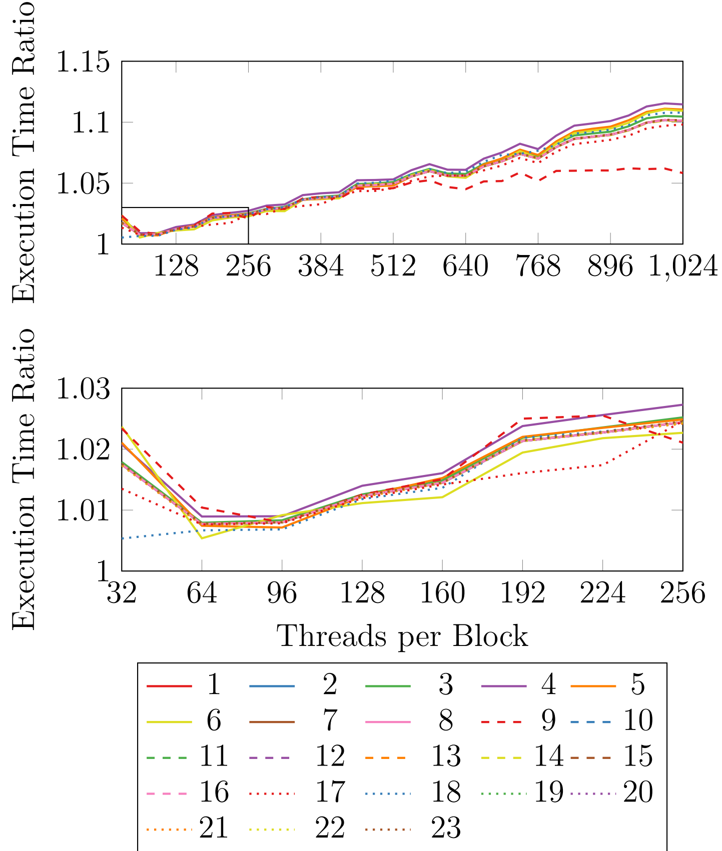


Figure 5.3: Minimum Execution Time at set threads per block / Minimum Execution Time. Each line represents 22 of the 23 benchmarks in a leave-one-out manner.

The heuristic used in LLVM calculates the number of blocks as a linear function of the *loop tripcount* – the number of parallel loop iterations to be executed. To investigate the validity of that assumption, Figure 5.5 plots, for each benchmark, the number of blocks required to minimize execution time when 96 threads per block are created, versus the loop tripcount. The plot also shows a line for the LLVM heuristic, assuming 96 threads per block instead of the original 128. If the LLVM heuristic's assumptions were optimal, we would expect a straight line from the bottom left to the top right of the plot. While there is indeed a weak linear relationship, there is clearly still some hidden variable, leaving room for a machine-learning model to discover

this relationship and improve performance.

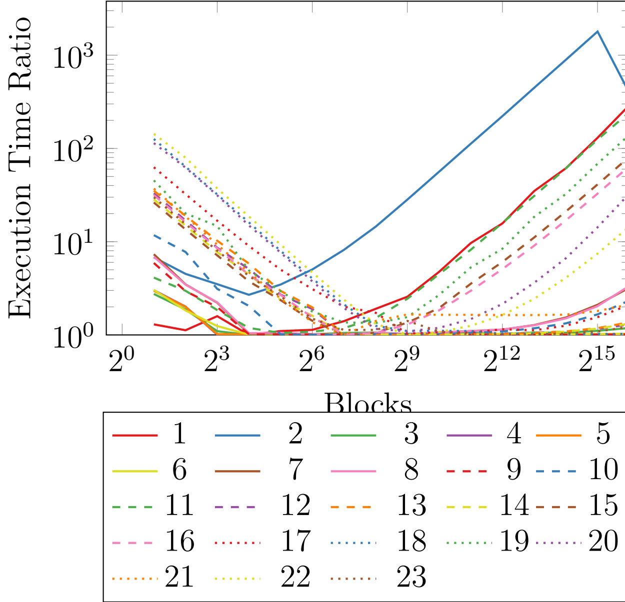


Figure 5.4: Minimum Execution Time given 96 threads per block / Minimum Execution Time. Average is weighted by the minimum execution time of each benchmark.

5.4 Modeling with Machine Learning

Our initial goal is to model the performance of a GPU kernel generated for a `target` region as a function of the selected grid geometry. The method is to use *offline supervised learning* to create a machine learning model that captures static and dynamic kernel characteristics in an attempt to generate a prediction of the optimal grid geometry. The model uses the dataset acquired through the exhaustive ex-

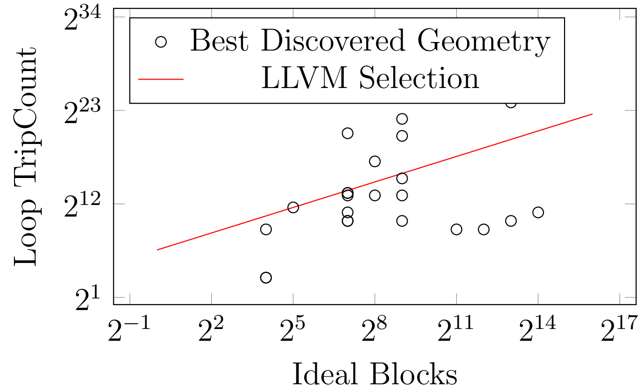


Figure 5.5: Number of blocks minimizing execution time given 96 threads per block / Loop Tripcount for each benchmark.

ploration of the grid geometry space for each kernel to train the predictor and is evaluated using leave-one-out cross-validation.

5.4.1 Finding Additional Features

Given only a weak linear relationship between the loop tripcount, used by the LLVM selection, and the ideal number of blocks (described in section 5.3.3), we investigate the use of additional features which may impact grid geometry. We introduce simple static analysis techniques to generate additional features, with the goal of further characterizing such kernels.

Stack Frame Size

The amount of local memory per thread, also known as the size of the stack frame, is one of the deciding factors in scheduling new thread blocks for execution on a given Streaming Multiprocessor (SM) . Because the SM has a fixed limited amount of memory dedicated to it, per-thread local memory usage is calculated at compile-time by the Nvidia PTX assembler. A sufficiently large stack frame might mean that only one thread block at a time may be scheduled for execution on an SM. Thus, a thread's stack frame size is a crucial parameter for deciding the number of blocks (teams) to use.

Register Count

Each SM has a fixed-size register file to be divided among running threads. For example, the Nvidia P100 has a 256KB register file per SM adding up to 65536 32-bit registers per thread-block. Thus, per-thread register count affects how many blocks can be scheduled to run on a single SM, and it is important that the register count is taken into account when choosing the number of threads and blocks. The per-thread register count is also a value computed at compile time by the Nvidia PTX assembler.

Directive and Clause Use

Different OpenMP directives and clauses impact the code generated by the compiler. Separating kernels based on which OpenMP constructs they employ

is a way to classify different behavior. Due to limited availability of OpenMP 4.x programs, there is little variety in the types of constructs used in our benchmark suite. Thus, we restricted this feature to count only the clauses that may affect inter-thread cooperation within thread-blocks.

Code Size Estimate

One possible reason for the variation in ideal number of blocks is the amount of work required per parallel loop iteration. If the work is extremely small, then it may be sensible to assign multiple iterations to a particular thread. Conversely, if the work required is large then it may make sense to assign each thread only a single iteration. To accommodate this intuition, we built a simple static analysis approximating the number of instructions executed per loop iteration. Each instruction is assumed to have equal cost, with conditional statements assumed to be evaluating true 50% of the time, and nested loops assumed to be executing 128 iterations. The resulting estimate of runtime instructions provides an estimate metric representing the total amount of work per-thread.

Total Work Estimate

Given that the size estimate feature is an approximate measure of work done per-thread, an intuitive extension to the feature is the product (size estimate $\times$ tripcount). This feature estimates the overall amount of work to be done by a given kernel. Ensemble techniques can theoretically capture this type of relationship as a meaningful feature on their own; however, this combined size estimate was devised as a means of providing meaningful information to simpler models with the aim of reducing prediction time.

Random Forest Model

Attempts to build a linear-regression model were not successful. Therefore, we turn to ensemble approaches. Random Forests are an ensemble learning method that can be used for both classification and regression. They are a combination of tree predictors such that each tree depends on the values of a

random vector sampled independently and with the same distribution for all trees in the forest. In this methodology, the model is designed to predict the execution time of a kernel based on a given grid geometry and the features already used for the linear regression model. For each kernel, execution times were obtained for all 544 thread-block combinations. Thus there is a total of $22 * 544 = 11968$ training data points and 544 test points (corresponding to each kernel not used for training). A forest with 2000 trees was empirically determined to yield the most accurate model. The implementation used is based on Breiman and Cutler's Random Forests for Classification and Regression [6].

At prediction time, the model is queried for the predicted execution time at every thread and block combination measured in the data exploration. The threads per block and block count that correspond to the shortest expected execution time are then selected for kernel launch.

5.4.2 Machine Learning Predictor Performance

Figure 5.6 shows the speedup over the LLVM selection for all the 23 kernels when using the best predicted grid geometry by this random forest model . The graph also displays the optimum speedup achievable through grid geometry, as shown in Figure 5.2. For data shown in this graph, the predictor is used to obtain both number of threads and number of threads per block that will be used for kernel launch. The performance obtained with these predictions ranges from 68% slower to 6.4 times faster, with a geomean speedup of 5% across all benchmarks. This result likely indicates that the model was able to discover relationships between program features that correlate with the kernel's performance.

5.5 Production Heuristic

The random forest model successfully predicted the grid geometry and outperformed the existing compiler heuristic, substantially for some benchmarks. However, the time taken to perform that prediction at runtime dwarfed the actual execution time of most benchmarks. With the small workloads used for

the grid-geometry space exploration, prediction time would exceed the runtime of many benchmarks.

Having constructed a model that could improve performance, but with impractical overhead, we first attempt to create simpler models that could replicate the prediction accuracy but with lower overhead. The accuracy of a new linear model was too low. Either the relations are truly non linear or the feature set, combined with a limited number of benchmark kernels, was insufficient to build a reasonably accurate linear relationship between the grid configuration parameters and execution time.

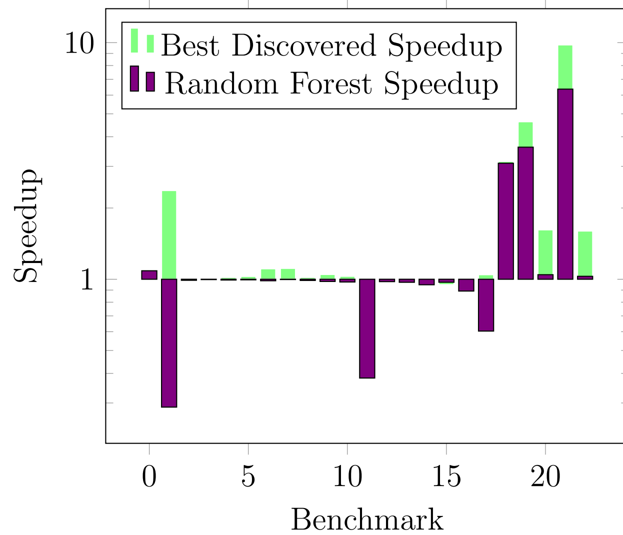


Figure 5.6: Speedup over the LLVM selection for the Random Forest Model Predictor grid configuration not including the prediction overhead. This performance cannot be realized in practice. Results are shown on a log scale to present equivalent speedups and slowdowns as equivalently-sized bars.

Next we examined the dataset, with the goal of figuring out the insights the random forest model had derived. Inspection revealed some intriguing relationships. A loop iteration is the smallest parallelizable unit of work. However, the product of threads per block and blocks that minimizes execution time is often *larger* than the loop trip-count. Thus, counterintuitively, some threads must be assigned *zero* work in this situation. For some benchmarks, the time is minimized at the first exploration point where all threads are assigned work, but for others, execution time is minimized when there are dramatically more threads. Finally, when the loop trip count becomes too large, this relationship breaks down, and fewer and fewer threads/loop iteration are required.

When execution time is minimized by using more threads than loop iterations, at least some warps of threads are partially empty, because OpenMP

loop iterations are first assigned to blocks, and then to threads within a block. These partially empty warps are then less affected by traditional GPU performance problems such as branch divergence and non-coalesced memory accesses [76]. However, using the GPU in this manner is extremely inefficient — the speedup over correct resource utilization is only ≈ 1%, which is negligible. The speedup gain would also be completely negated if the GPU is used by multiple-programs at the same time because the resources wasted through over-provisioning would increase the device occupancy, preventing other programs from being scheduled efficiently.

Based on these insights, the kernels studied can be divided into three classes, to be handled separately by a new compiler heuristic:

1. **Short-Loop Kernels** - Kernels that use a small amount of parallelism are likely not well suited for GPU execution. Such kernels tend to be naively translated from parallel code written for CPUs and do not consider the unique characteristics of the GPU architecture. The heuristic can improve the overhead of such benchmarks by creating blocks of 1 thread each, and distributing work across SMs to treat the GPU more similarly to a large multi-core system. Because blocks of size 1 use so few resources on each SM, they don't suffer as much from the resource-wasting problem described earlier.

   These kernels are detected when the loop trip count is less than or equal to the number of available SMs. For these kernels we use 1 thread per block and a number of blocks equal to the loop trip count.

2. **Ideal-Loop Kernels** - Kernels that can use an appreciable fraction of the GPU are already well-handled by the existing heuristic. No substantial performance gains can be found here because the grid selected is already relatively optimal.

   These kernels have a loop trip count that is larger than the number of SMs available on the GPU (28 for our Nvidia Titan X Pascal). For these kernels, the heuristic prescribes 96 threads per block, and the number of blocks is $\lceil \frac{\text{tripcount}}{96} \rceil$ blocks.

3. **Long-Loop Kernels** - Kernels with loop trip counts vastly higher than the GPU can execute simultaneously generate massive queues of blocks, preventing opportunistic work. By limiting the number of blocks executed to the maximum executable by the device, the queuing overhead can be reduced substantially.

   These kernels can be identified at runtime by inspecting both the maximum number of blocks the GPU can execute for this kernel, and inspecting the loop trip count provided by the kernel. If the loop trip count exceeds the product of threads per block and number of blocks, then a kernel falls into this class. For these kernels, the new heuristic prescribes the use of 96 threads per block, and the setting of the number of blocks to the maximum that can be simultaneously loaded on the device without queuing.

Formalizing the features considered by the heuristic, a GPU device descriptor should specify the following properties:

- **SMCount** The number of streaming multiprocessors available on the device.

- **ThreadLimit** The maximum number of threads an SM can hold simultaneously.

- **RegisterLimit** The maximum number of 32-bit registers an SM can hold.

- **SharedMemLimit** The maximum amount of shared memory available on an SM.

- **BlockLimit** The maximum number of blocks an SM can hold.

- **ThreadsPerBlock** An experimental value, the ideal threads per block for this device.

Section 5.3.2 determined that 96 threads per block is the prediction for most benchmarks. However to use a proper methodology, the evaluation of the

```
GetGeometry ( device, kernel ):
  if kernel.Parallelism ≤ device.SMCount then
    /* Short-Loop Kernels                                    */
    threads = 1;
    blocks = kernel.Parallelism;
  else
    threads = device.ThreadsPerBlock;
    threadLimit = device.ThreadLimit / device.ThreadsPerBlock;
    regLimit = device.RegisterLimit / (kernel.Registers *
     device.ThreadsPerBlock);
    sharedLimit = device.SharedMemLimit / kernel.SharedMem;
    blocksPerSM = min(threadLimit, regLimit, sharedLimit,
     device.BlockLimit);
    maxBlocks = blocksPerSM * device.SMCount;
    kernelBlocks = kernel.Parallelism / threads;
    if kernelBlocks ≥ maxBlocks then
      /* Long-Loop Kernels                                   */
      blocks = maxBlocks;
    else
      /* Ideal-Loop kernels                                  */
      blocks = kernelBlocks;
    end
  end
  return (threads, blocks)
```

Figure 5.7: Final Heuristic Algorithm

heuristic uses the threads-per-block value predicted by the fold that excluded the benchmark that is been evaluated.

The heuristic also requires a kernel descriptor that contains the following properties:

- **Registers** The number of registers required per thread by a kernel.
- **SharedMem** The amount of shared memory required per block by a kernel
- **Parallelism** The number of parallel work units (typically loop iterations) in a kernel

The heuristic pseudocode is shown in Figure 5.7. This heuristic uses distinct strategies to generate grid geometry for all three kernel classes. This

heuristic meaningfully captures all of the kernels that we studied. It accounts for the behaviours observed and makes efficient use of resources. A key insight is to avoid the drastic over provisioning required to truly minimize kernel execution time. An evaluation of this new heuristic against the LLVM selection is shown Figure 5.8. The kernels on the left have long loops, the kernels on the right have short loops and the ones on the middle have ideal loops. Results range from 39% slower to 7 times faster, with a geomean speedup of 25.9%. The methodology in the evaluation is identical to the one used for data collection presented in Section 5.3: five runs were performed for every configuration, with the mean runtime of each used for all calculations. Variance among results was statistically insignificant as a result of measures put in place to reduce it, as described at the end of Section 3.0.

The speedup comes exclusively from the kernel with long or short loops because the kernels with ideal loops are already well-optimized for GPUs by the LLVM selection. There were only two kernels with short loops in the sample, the performance for one improved drastically while the performance for the other was unaffected. Performance improvements are observed in this class of kernels when the kernel was poorly written for execution in GPUs and the heuristic causes threads to be separated across SMs. The long-loop kernels generally sees large performance improvements, with few slowdowns. The proposed heuristic goal is to avoid block queuing. Future heuristics may be able to separate and identify cases where block queuing is desirable. The proposed heuristic

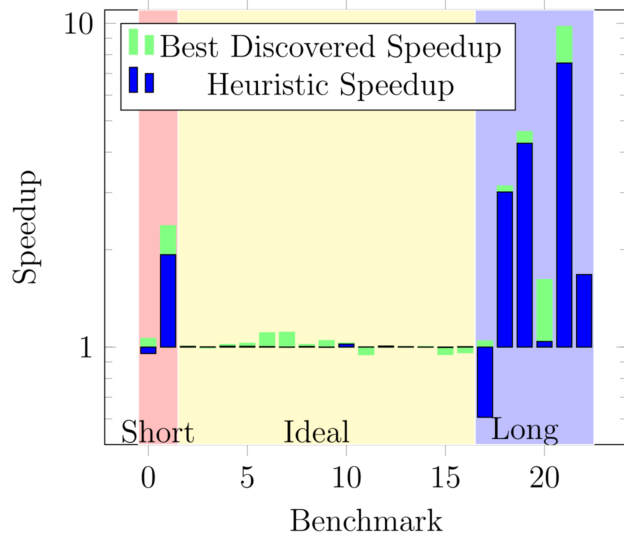


Figure 5.8: Speedup for our modified heuristic over the LLVM selection. This performance can be realized in practice. Results are shown on a log scale.

successfully capitalizes on cases where the LLVM selection performed poorly, while maintaining performance on kernels with ideal loops. The low computational complexity makes the runtime overhead of the heuristic negligible and its simplicity allows programmers predictable performance.

The grids chosen for each kernel, and the associated speedups are shown in Table 5.1.

5.5.1 Edge-Case: OpenMP SIMD

While our proposed heuristic matches, or exceeds, the performance of the existing heuristic on 22 of 23 kernels, and is within 10% of the best discovered performance on 19 of 23 kernels, performance was substantially degraded for one kernel. Benchmark 18 makes use of the `omp simd` construct, which, according to the specification, directs OpenMP implementations to implement the following loop using SIMD vector units. The selection of threads per block, using the leave-one-out strategy described in Section 5.3.2, indicates 32 threads to be the value most likely to maximize performance. SIMD execution on a GPU is emulated using additional dedicated warps of threads, because current GPUs do not have SIMD units. Thus, as an artifact of the code-generation scheme, the SIMD region is serialized, leading to poor performance. Intrinsically, until GPUs incorporate SIMD units, code-generation for the `simd` pragma in GPU code will remain a crutch that leads to inefficient code. Still, to maximize performance, the correct strategy is to allocate extra warps to accommodate the SIMD construct code generation. The near-optimal grid geometry configuration discovered for this kernel indicates that blocks of 256 threads lead to good performance. To generalize this insight, more code that utilizes SIMD constructs is needed.

5.5.2 Implications of Volta

The Volta architecture, recently launched by NVIDIA, introduces several changes that would require minor adjustments to the heuristic approach presented in this paper [49]. Pascal generation cards, which have been used in this work, have SMs that can issue an instruction for 64 threads per cycle. This number

gives insight to the discovery made during grid geometry search space exploration that experimentally deemed 96 threads per block to be a reasonable choice for most OpenMP kernels we have encountered. By only slightly over prescribing the number of threads per block to the number of threads that can be issued an instruction each cycle, a sufficient amount of latency-hiding can be achieved without suffering the excessive scheduling overhead. In Volta, individual SMs have higher core counts and can issue an instruction to double the number of threads per cycle. While we expect our insights to scale similarly to the new architecture, a new set of experiments, similar to the ones performed in section 3, is required to derive the **ThreadsPerBlock** value for Volta-based devices.

5.6 Concluding Remarks

Finding a heuristic that can perform well on a diverse set of programs can be a challenging task that requires extensive analysis. Compiler construction has a long history of optimizers that consist of such heuristics, with hand-tuned parameters evolving over time out of years of expertise and experiments by researchers and developers. Machine learning has been recently gaining traction as a tool in the compiler researcher's toolbox that can model characteristics of program behavior useful for compile-time decisions. Despite strengths in capturing unknown relationships to produce meaningful predictions, using machine learning to model program performance has drawbacks. Collecting sufficient programs to successfully predict performance can be more difficult than to invent a heuristic that achieves the same result. Moreover, even for successful and powerful models, usage of dynamic program features can make such predictor systems infeasible because of incurred prediction time costs. In this paper we have demonstrated the problem of tuning the GPU grid geometry for specific kernels generated from OpenMP 4.x programs that use accelerator offloading constructs. Based on a set of static and dynamic features, approximating the thread and block values to maximize efficient hardware resource utilization is a problem that intuitively lends itself to a modelling approach.

Benchmark	LLVM	ML Model		Final Heuristic		Best-Discovered Configuration	
	Grid Geometry	Grid Geometry	Speedup	Grid Geometry	Speedup	Grid Geometry	Speedup
1	(128,1)	(96,256)	1/1.380	(1,10)	1/1.047	(64,8)	1.064
2	(128,1)	(96,1024)	1/23.784	(1,10)	1.930	(32,2)	2.371
3	(128,4)	(96,512)	1.005	(64,8)	1.005	(64,4096)	1.006
4	(128,4)	(96,512)	1.002	(64,8)	1.002	(64,512)	1/1.005
5	(128,4)	(96,2048)	1.004	(64,8)	1.006	(32,256)	1.018
6	(128,8)	(96,256)	1.002	(64,16)	1.005	(32,4096)	1.028
7	(128,8)	(96,1024)	1/1.030	(64,16)	1.003	(96,128)	1.107
8	(128,8)	(96,1024)	1/1.028	(64,16)	1.000	(96,128)	1.111
9	(128,8)	(96,1024)	1/1.001	(64,16)	1.004	(32,2048)	1.019
10	(128,16)	(96,512)	1/1.009	(64,32)	1/1.000	(96,128)	1.047
11	(128,16)	(96,256)	1/1.015	(32,64)	1.022	(512,32)	1.029
12	(128,24)	(96,512)	1/2.386	(64,48)	1.001	(32,128)	1/1.055
13	(128,64)	(96,512)	1/1.002	(64,128)	1.008	(64,128)	1.001
14	(128,64)	(96,1024)	1.002	(64,128)	1.003	(160,128)	1.003
15	(128,64)	(96,2048)	1/1.010	(96,256)	1.001	(320,32)	1.004
16	(128,79)	(96,512)	1/1.316	(64,156)	1/1.002	(192,64)	1/1.054
17	(128,79)	(96,512)	1/1.259	(64,156)	1.003	(160,64)	1/1.041
18	(128,256)	(96,512)	1/1.001	(64,448)	1/1.648	(256,128)	1.045
19	(128,1024)	(96,256)	3.009	(64,448)	3.014	(128,128)	3.147
20	(128,8192)	(96,512)	3.877	(64,448)	4.261	(256,128)	4.630
21	(128,10157)	(96,512)	1/1.026	(64,448)	1.041	(96,128)	1.617
22	(128,32768)	(96,512)	8.554	(64,448)	7.541	(384,128)	9.779
23	(128,125986)	(96,1024)	1.502	(64,448)	1.675	(384,1024)	1.598

Table 5.1: Grid Geometry (threads-per-block, blocks) selected for each benchmark by the LLVM selection, our ML model, our proposed heuristic, and exhaustive search. Speedup is shown relative to the LLVM selection. Slowdowns are shown as reciprocals for clarity. Thread-Per-Block values for the the Final Heuristic selected using leave-one-out strategy as described in 5.3.2.

Yet, because prediction relies on the parallel loop tripcount being a key feature in estimating the amount of parallelism exhibited by the code, even a successfully tuned model proved unusable because the prediction time often exceeded kernel execution time. Despite the limited practicality of the machine learning model for this problem, its success in generating superior grid geometry led to useful insights that could be learned by examining the trends discovered in the trained model. These insights, in turn, enabled the creation of a heuristic. This hybrid approach of machine learning as a means to inform or guide researchers shows that predictive models can not only be used to directly make decisions, but also to aid the creation of heuristics through expert knowledge.

Chapter 6

GPUCheck: Detecting CUDA Thread Divergence with Static Analysis

6.1 Introduction

This chapter addresses the following research problem: *given a program containing portions that are designated for execution on GPUs, detect and report branch divergences and non-coalescable memory accesses.* To achieve that, we present GPUCheck, a static analysis tool built on top of a novel static analysis framework, that identifies and reports the locations in a given program source code that are likely to exhibit poor GPU performance. We also describe a prototype implementation of GPUCheck on top of the Clang/LLVM compiler [37] that works on an intermediate representation of the program, with mappings to the original source code. GPUCheck uses our novel inter-procedural, context-sensitive Arithmetic Control Form (ACF), a representation for static address range analysis and conditional expression analysis. Our prototype identifies performance issues in 17 well-known Rodinia benchmarks [10], including the example in Figure 6.1.

Static analysis can detect divergence performance issues in many common cases. For the majority of programs, branch divergence occurs in a GPU when the condition evaluated by the branch depends, either directly or indirectly, on the thread index. Therefore, branch-divergence detection can be defined as a taint-analysis problem [3] that identifies whether variables and memory

```
21 den = (qsqr-q0sqr) / (q0sqr * (1+q0sqr)) ;
22 c = 1.0 / (1.0+den) ;
23 if (c < 0){temp_result[ty][tx] = 0;}
24 else if (c > 1) {temp_result[ty][tx] = 1;}
25 else {temp_result[ty][tx] = c;}
```

Figure 6.1: Original diffusion coefficient calculation in `srad`.

```
26 den = (qsqr-q0sqr) / (q0sqr * (1+q0sqr)) ;
27 c = min(max(1.0f / (1.0f+den),0.0f),1.0f) ;
28 temp_result[ty][tx] = c;
```

Figure 6.2: Modified diffusion coefficient calculation in `srad`.

locations *may* be *tainted* by the thread index. A branch whose computation is tainted by the thread index may exhibit divergent behaviour. However, this check alone might lead to many false positives. Thus, such divergence analysis must be combined with a pruning algorithm to provide helpful feedback to programmers.

The example in Figure 6.1, taken from the `srad` benchmark in the Rodinia suite [10], illustrates branch divergence. To ensure that the coefficient `c` stays within the range 0 to 1, the code tests the value of `c` and makes the appropriate correction. However, the value `qsqr` is derived from data calculated from each thread's location in a 2D grid. Therefore, the value of `c` is different for each thread, causing potential control-flow divergence: a different group of threads may execute each of the statements in lines 23–24, leading to three execution cycles, each requires a memory access. Figure 6.2 shows an alternative implementation, where testing the value of `c` is a computation of `min` and `max` operations that are available as instructions in NVidia GPUs. In the transformed code, all threads execute the memory-access statement in Line 27 simultaneously. By modifying the example to avoid the divergence, performance can be substantially improved.

GPUCheck tends to be substantially faster than dynamic techniques, because it does not need to execute a program to perform its analysis. On the Rodinia benchmark suite, the median completion time for GPUCheck's anal-

```
29 kernel_compute_cost(int num,
       int dim, long x,
30           Point *p, float *
                 coord_d, ...) {
31
32    const int tid;
33    tid = blockDim.x * bid +
         threadIdx.x;
34    ...
35    float x_cost = d_dist(tid, x
         , num, dim, coord_d) * p
         [tid].weight;
```

(a) Noncoalescable memory accesses

```
36 kernel_compute_cost(int num,
       int dim, long x,
37           float *p_x, float *p_y,
                 float *p_z,
38           float *p_weight, float
                 *coord_d, ...) {
39    const int tid;
40    tid = blockDim.x * bid +
         threadIdx.x;
41    ...
42    float x_cost = d_dist(tid, x
         , num, dim, coord_d) *
         p_weight[tid];
```

(b) Perfectly coalesced accesses

Figure 6.3: Extract from `streamcluster`.

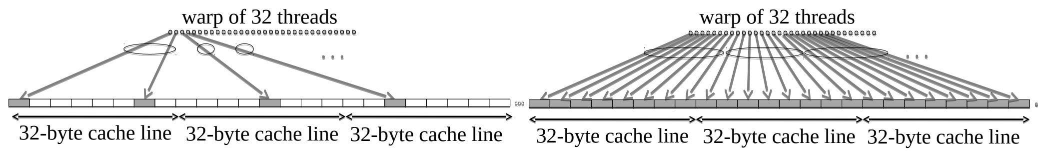


Figure 6.4: `streamcluster` access pattern before (left) and after the code transformation (right).

ysis is 210 ms, in comparison with 36.36 seconds required by Nvidia's own dynamic profiler `nvprof` [47].

In summary, this chapter makes the following contributions:

1. Arithmetic Control Form (ACF), a representation for statically computing differences between expressions computed by various threads.

2. GPUCheck, a static analysis tool that identifies common sources of performance degradation in GPU programs.

Figure 6.3a shows an extract from the `streamcluster` benchmark that illustrates a non-coalescable memory access, as detected by GPUCheck. The code computes a weighted Euclidean distance between points, using the array of structures `p[tid].weight` in the computation. The data structure `Point` occupies six 32-bit words as shown on the left of Figure 6.4, and there is a

gap between the `weight` field, shown in grey, of subsequent structures. As a consequence, the execution of the code in Line 35 of Figure 6.3a leads to the coalescing pattern shown on the left of Figure 6.4 where each memory transaction, represented by the ellipses, fetches few `weight` fields. A fix to this issue consists in using separate arrays for each member in the struct as shown in Figure 6.3b. The right of Figure 6.4 shows the placement in memory of the array `p_weight`. In this pattern, the accesses are coalesced into fewer memory transactions, each bringing the maximum number of `weight` fields allowed by the memory bandwidth.

6.2 Static Analysis Engine

The core of GPUCheck is a series of static analyses, which combine to determine how execution behaviour differs between threads, and the effects those differences can have on performance. In this section, we provide an overview of thread-dependence analysis and Arithmetic Control Form.

6.2.1 Thread-Dependence Analysis

An expression is *thread dependent* if it generates different values depending on the thread executing it at runtime. To detect potential thread-dependent expressions statically, GPUCheck identifies sources of thread dependence and propagates them through their uses in the program. As a motivating example, consider the pseudocode in Figure 6.5 where `a` is thread dependent, because it contains the value of the thread identifier `threadIdx`. Similarly, `c` is thread dependent, because it derives from `a`. However, is `b` thread dependent? The answer depends on the analyzed program point. For instance, at the point immediately after Line 46, the value of `b` is a constant 1, and at the point immediately after Line 48, `b` is a constant 0. However, `b` as used at Line 51 is clearly thread dependent, as the value depends on the if-condition. Given this intuition, we define thread dependence as follows:

Definition 1. An expression is *thread dependent* at a program point p if it may evaluate to different values by different threads at p.

If a given expression is not a source of thread dependence, then it can only be thread dependent through data-flow dependencies or control-flow dependencies. Detecting thread dependence through data-flow dependencies is trivial, as an operand of the expression must also be thread dependent. However, calculating thread dependence caused by control-flow dependencies requires additional analysis. Control-flow thread dependence occurs when a conditional branch evaluates a thread-dependent expression that may evaluate to different values depending on the execution path.

```
43 a = threadIdx;
44 c = a % 2;
45 if(c) {
46   b = 1;
47 } else {
48   b = 0;
49 }
50 Arr[a] = 0;
51 Arr[b] = 0;
```

Figure 6.5: An example illustrating thread dependence.

Divergence—both for branches and for memory access addresses—originates from source statements and is propagated to other expressions in the program through control-flow and data-flow dependencies. For each expression in the program, the thread-dependence analysis in GPUCheck computes a boolean value that determines if the expression is thread dependent. GPUCheck regards source statements as *tainting* statements and employs a taint analysis to determine which expressions are tainted by thread-dependent calculations. In a GPU program, a statement may be a source of thread dependence due to a variety of operations. For instance, a statement that randomly generates values, or performs an atomic operation such as compare-and-swap, may compute values that may be unique to each thread. However GPUCheck only considers statements retrieving the GPU thread identifiers (*threadIdx*) as sources of divergence.

GPUCheck uses an Interprocedural Finite Distributive Subset (IFDS) [44] taint analysis over a static single assignment (SSA) intermediate representation. Thread-dependence information is propagated through the IFDS supergraph representation of the program [56]. Given the set D of all SSA expressions in the program, the dataflow problem consists in determining which subset of D is thread dependent at each point in the program. The thread-

dependence property is propagated through distributive dataflow functions that can be decomposed into micro-functions. Each micro-function $f_x(y)$ expresses the propagation of the expression $x \in D$ through the statement that defines the expression $y \in D$. The special function *gen* creates and propagates the thread-dependence property regardless of the prior state, the function *prop* propagates the input thread-dependence state to output, and the function *kill* propagates no thread-dependence property, regardless of input. The use of an expression x in a program statement does not change the thread-dependence property of x. Therefore, for each statement that defines an expression $d \in D$, the thread-dependence property of each expression x where $x \neq d$ remains unchanged: $f_x(d) = prop$. If the statement that defines d reads the thread identifier, then d is a source of thread dependence and the micro-function $f_d(d) = gen$. Otherwise, $f_d(d)$ depends on the thread-dependence property of the operands used in the statement that defines d and on the control-flow dependencies of the statement that defines d. To compute $f_d(d)$, let $O(y)$ be the set of expressions that are used in the statement that defines expression y. Let $CDG(y)$ be the set of basic blocks that the statement that defines y is control dependent on, as determined by a control-dependence graph (CDG) [20]. Then $CDG(y)$ is the set of basic blocks that determines whether or not y executes. Let $cd(y)$ be the set of expressions used as conditions for branches exiting each basic block in $CDG(y)$. The value of $f_d(d)$ is then given by:

$$f_d(d) = \bigcup_{o \in O(d)} \big(f_o(d) \cup \bigcup_{c \in cd(o) \setminus cd(d)} f_c(d)\big) \tag{6.1}$$

Intuitively, $\bigcup_{o \in O} f_o(d)$ captures thread dependence over data dependencies by combining the thread dependence of all operands of d. The control dependences for an operand $o \in O(d)$ of the definition d (denoted $cd(o)$) are the decisions that lead to the execution of the expression that defines o. To capture relevant control dependences, the analysis computes the conditions that are required to reach an operand of d, but not d itself, producing the set difference $cd(o) \setminus cd(d)$. Such conditions are control dependences, and are combined to produce $f_d(d)$.

```
52 int readBounded(int* a) {
53     int tx = threadIdx.x;
54     if(tx > 256)
55         tx = 256;
56     int *addr = a + tx;
57     return *addr;
58 }
```

(a) A bounded array access.

```
59      tx0 = threadIdx.x
60      p1 = tx > 256
61 p1? tx1 = 256
62      tx = ψ(tx0, p1?tx1)
63      tmp = 4 * tx
64      addr = a + tmp
65      return addr
```

(b) If-converted ψ-SSA form for the code in **(a)**.

Figure 6.6: An example illustrating if-conversion in ψ-SSA, which serves as inspiration for our ACF analysis.

6.2.2 Arithmetic Control Form (ACF)

The thread-dependence analysis determines which expressions in the program are thread-dependent. The intuition is that a conditional expression that is thread dependent is a potential source of control-flow divergence, and a memory-access expression that is thread dependent is a potential source of non-coalescable memory accesses. However, this intuition alone would lead to numerous false positives, i.e., GPUCheck would be signalling potential divergences that are not actual divergences. For memory accesses, we are interested in determining if the range of addresses accessed by all threads in a warp falls within a single cache line. To achieve that, we have designed the Arithmetic Control Form (ACF) analysis. Given a thread-dependent expression, ACF determines the difference between the value of this expression as computed by each thread.

The value computed by an expression depends not only on the flow of values through expressions, but also on the conditional statements in the code. Existing work in support of if-conversion in SSA form [43], [65] serves as an inspiration for ACF. Ferriere and Stoutchinin [65] introduced ψ-nodes to represent the flow of SSA values through a segment of straight-line code in the presence of predicated execution. Intuitively, a ψ-node combines the results of multiple predicated instructions, unifying values in straight-line code in the same way ϕ-nodes unify values from differing basic blocks in traditional SSA.

Figure 6.6a shows a simple bounded array indexing operation. The ψ-SSA form for this code is shown in Figure 6.6b after if-conversion. The transformed code is in single-assignment form and the if-statement conditional expression is stored in predicate register `p1`. The ψ-node thus uses predicates to select between multiple possible values.

In the ψ-SSA form, ψ-nodes are equivalent to the sum of all incoming values multiplied by the associated incoming predicate. ACF extends this notion by computing complex predicates through as much of the program's control-flow as necessary to obtain an expression that precisely captures all possible execution traces. Intuitively, an ACF value for an expression is equal to a sum, where each summed element corresponds to a control-flow path and has a value equal to the expression as computed along that path, multiplied by the conditions required to execute that path.

For each execution of the code, a single predicate combination evaluates to 1 and all the other combinations evaluate to 0. Through this transformation, ACF produces a symbolic equation for each expression of interest. In essence, ACF is an alternative program representation, suited for analysis rather than actual execution. ACF represents the value generated by each expression as a tree of arithmetic operations, constants, and unknown values. For each expression of interest in a given CUDA kernel, ACF computes the differences between the expression as it is evaluated by each thread. Threads are computed by substituting constant thread identifiers, and simplification is performed by merging common predicates and cancelling non-thread-dependent subexpressions. In practice, most differences statically evaluate to a constant after simplification and thus can be used to determine if a tainted expression leads to either thread divergence or non-coalesced accesses. Consider the example code in Figure 6.6a, which implements a bounded array access. To compute the address accessed by each thread in Line 56, ACF analyzes all possible paths through the function. In this case, there are two paths, corresponding to the if case or the else case. Let `v` be a variable in the program. In ACF, the notation $[\boldsymbol{v}]$ indicates that `v` is represented symbolically, and $ACF(\boldsymbol{v})$ is the ACF value for `v`. A subscript indicates that a reference is thread-dependent,

and specifies the thread. The ACF value for the address on Line 56 is then defined as:

$$
\begin{aligned}
&ACF_t(\boldsymbol{addr}) = \\
&([\boldsymbol{threadIdx.x}_t] > 256) * ([\boldsymbol{a}] + 4 * 256) + \\
&([\boldsymbol{threadIdx.x}_t] \leq 256) * ([\boldsymbol{a}] + 4 * [\boldsymbol{threadIdx.x}_t])
\end{aligned}
$$

Whenever possible, ACF replaces variable references with their definitions. For example, $[\boldsymbol{threadIdx.x}_t]$ is used instead of `tx` in the ACF representation of the code in Figure 6.6a. An unknown value, such as $[\boldsymbol{a}]$ in Figure 6.6a, is represented symbolically. Given the ACF representation for the code in Figure 6.6a, a consuming analysis may query for the difference between the `addr` returned by threads 0 and 1: $ACF_1(\boldsymbol{addr}) - ACF_0(\boldsymbol{addr})$. The thread-dependence analysis determines which symbolic references, such as $[\boldsymbol{a}]$, are not thread dependent. Such references cancel out in the computation of differences.

$$
\begin{aligned}
&ACF_1(\boldsymbol{addr}) - ACF_0(\boldsymbol{addr}) \\
&= ((1 > 256) * ([\boldsymbol{a}] + 4 * 256) + (1 \leq 256) * ([\boldsymbol{a}] + 4 * 1)) \\
&- ((0 > 256) * ([\boldsymbol{a}] + 4 * 256) + (1 \leq 256) * ([\boldsymbol{a}] + 4 * 0)) \\
&= ([\boldsymbol{a}] + 4) - [\boldsymbol{a}] \\
&= 4
\end{aligned}
$$

For loops, ACF has to deal with loop induction variables. Similar to unknown variables, ACF handles a loop induction variable `iv` symbolically, using the following notation: $ACF(\boldsymbol{iv}) = [\boldsymbol{iv}]$. For instance, Figure 6.7 shows a simple parallel implementation for `memcpy`. The query $ACF(\boldsymbol{srcaddr})$ is used to determine the value of `srcaddr`. To solve this query, ACF captures the common behaviour across all loop iterations, under the assumption that if values common within the loop are not thread dependent, they will often cancel when a difference is calculated. $ACF(\boldsymbol{srcaddr})$ is therefore calculated as follows:

$ACF_t(\boldsymbol{srcaddr}) = [\boldsymbol{src}] + [\boldsymbol{i}] + [\boldsymbol{threadIdx.x}_t]$

ACF treats the index variable `[i]` symbolically. If both the initialization expression and the reinitialization expression for `[i]` are thread independent,

then the value of `[i]` is also thread independent. Thus, when computing the difference between expressions involving `[i]` for any loop iteration, the symbolic value `[i]` disappears, resulting in either a constant distance between threads or a distance that depends either on the thread identifier or on other symbolic variables.

To calculate the ACF representation for the function call `c = f(`$\langle args \rangle$`)` with the return expression *ret*, GPUCheck first calculates $ACF(ret)$, then replaces any arguments in $ACF(ret)$ with the actuals in $\langle args \rangle$. For the code example in Figure 6.8, $ACF(y)$ is calculated as follows:

$$\begin{aligned}
ACF(y) &= ACF(a(x)) \\
&= [i] - 16 \\
&= ACF(x) - 16 \\
ACF(x) &= ACF(b(4)) \\
&= [j] + 8 \\
&= 12 \\
ACF(y) &= 12 - 16 = -4
\end{aligned}$$

In this example, calls are resolved *down* the call stack. However, in many cases, branch divergence and memory coalescing analyses require upward call resolution, because the analysis is performed inside a nested function. For instance, in the example in Figure 6.8, what is $ACF(br)$? In ACF, the answer can only be $[j] + 8$. To provide more precise results, an inter-procedural ACF is needed.

```
66 void memcpy (char* tgt,
67             char* src,
68             size_t sz) {
69   int tx = threadIdx.x;
70   int dim = blockDim.x;
71   for(int i=0; i+tx<sz; i+=dim)
       {
72       char *tgtaddr = tgt + i +
           tx;
73       char *srcaddr = src + i +
           tx;
74       *tgtaddr = *srcaddr;
75   }
76 }
```

Figure 6.7: An example illustrating how ACF handles loops.

6.2.3 Inter-procedural Arithmetic Control Form (IACF)

To operate inter-procedurally, GPUCheck maps the actual arguments from a function call to the formal parameters in the function definition. This mapping may lead to multiple ACF representations for a given program expression—potentially one for each calling context.

```
77 int a(int i) {
78     int ar = i - 16;
79     return ar;
80 }
81 int b(int j) {
82     int br = j + 8;
83     return br;
84 }
85 int main() {
86     int x = b(4);
87     int y = a(x);
88     int z = b(y);
89     return y;
90 }
```

Figure 6.8: An example illustrating the need for Inter-procedural Arithmetic Control Form (IACF).

Producing IACF requires an inter-procedural control-flow graph (ICFG). GPUCheck constructs a set of IACF representations iteratively by first calculating the ACF representation in the function's context. GPUCheck then inspects this representation for references to the function's arguments. For each reference to a function argument, GPUCheck identifies all non-recursive call sites to the function, and substitutes the actual arguments for the formal parameters for each call

site. This process continues until all arguments corresponding to non-recursive function calls have been replaced. Similar to loop induction variables, arguments to recursive calls remain symbolic references. Therefore, IACF generalizes over recursive paths similarly to looping paths, sacrificing some precision for performance.

For the example in Figure 6.8, the best approximation that the intra-procedure analysis produces for the possible values of `br` in Line 82 is $ACF(\boldsymbol{br}) = [\boldsymbol{j}] + 8$. IACF produces a more precise result. IACF discovers that there is a set of two possible values for `br`, because there are two calls to the function `b()`, `b(4)` in Line 86 and `b(y)` in Line 88.

$$\begin{aligned} IACF(\boldsymbol{br}) &= \{[\boldsymbol{j}] + 8\} \\ &= \{ACF(4) + 8, ACF(\boldsymbol{y}) + 8\} \\ &= \{12, 4\} \end{aligned}$$

While IACF sets may grow arbitrarily large, GPU kernels tend to have small call graphs. In our experimental evaluation with the Rodinia benchmark suite, we found at most 6 IACF expressions for any given expression. The Rodinia benchmark suite claims to be representative of typical GPU applications, thus IACF performs adequately for GPU code. In applications where memory space or computational time is limited, ending IACF expansion when a set reaches a specified maximum size (similar to k-limiting [32]) allows for sacrificing precision to improve performance.

Figure 6.9 outlines the various analyses in GPUCheck and their dependences. Figure 6.9 also includes the dependent performance detection analyses detailed in Section 6.3.

6.3 Detecting Divergent Behaviour

We have implemented the detection algorithms in GPUCheck using IACF expression sets. These algorithms calculate differences between IACF expressions evaluated by different threads in a warp to detect thread-dependent behaviour. IACF expressions typically contain many run-time references that

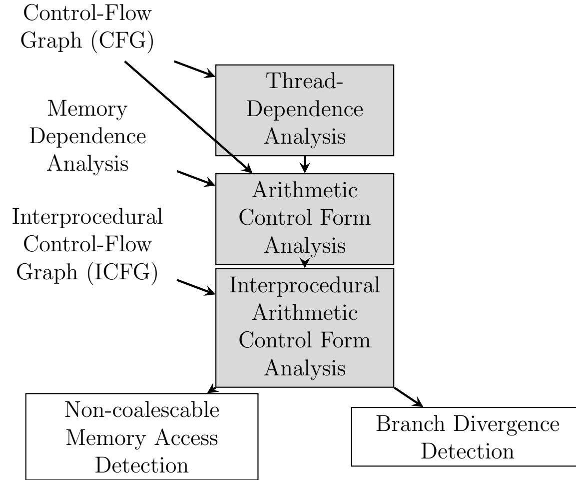


Figure 6.9: GPUCheck Analysis Workflow

the thread-divergence analysis have determined not to be thread dependent. In the calculation of the difference between two IACF expressions evaluated by different threads, thread-independent run-time references cancel out. In many cases, the result of the difference is a constant offset between the accesses in two threads that can be used for performance analysis.

6.3.1 Divergent-Branch Analysis

Divergent-branch analysis takes as input the set of thread-dependent conditional branches in the program under analysis, as discovered by the thread-dependence analysis. To improve the precision of GPUCheck, the analysis assumes a constant grid of 256 threads per block and 1 block per grid. It is better to use the actual grid that is created at runtime, but the grid is defined in host code and therefore not available during device-code analysis. The use of a constant grid can cause false positives or negatives: some CUDA code may assume the use of particular grid geometry. Typical CUDA code queries and adapts to arbitrary geometry, but if source code expects a particular grid then GPUCheck may produce unsound results. Because the results produced by GPUCheck are used to raise warnings, trading potential unsoundness for improved precision is a reasonable tradeoff.

GPUCheck constructs IACF expression sets for each thread-dependent con-

ditional branch. For each IACF expression $IACF_t(e)$, and for each thread t_x in a warp of threads $t_0 \dots t_{31}$, GPUCheck calculates the difference $IACF_{t_x}(e) - IACF_{t_0}(e)$. If any difference is non-zero, the warp of threads is branch divergent. GPUCheck calculates the divergence of each branch over all warps as $\frac{warps_{divergent}}{warps_{total}}$. If a branch exceeds a threshold n, then GPUCheck considers the branch sufficiently divergent, and reports it to the programmer as a warning.

6.3.2 Non-coalescable Memory Access Analysis

GPUCheck uses IACF to estimate the number of requests required to fulfill a memory operation that accesses a thread-dependent address. The analysis in this paper only models accesses to global memory. Shared-memory accesses require different access patterns for coalescing to occur. Moreover, penalties for non-coalescable accesses are substantially lower in shared memory. Therefore, GPUCheck does not attempt to analyze shared-memory accesses.

GPUCheck uses an address-space analysis to identify memory operations on pointers that may point to global memory. Non-coalescable memory access analysis takes the set of thread-dependent addresses from loads and stores that may point to global memory as input. Similar to divergent-branch detection, the analysis assumes a constant grid to improve precision at the expense of possible unsound results. For each IACF expression, and for each thread x, the analysis calculates the difference $IACF_{t_x}(addr) - IACF_{t_0}(addr)$.

The calculated differences are either constant, in which case they are collected in a set C, or non-constant, in which case they iterate a counter *non-const*. There are two reasons an IACF difference could be non-constant. Either the IACF analysis is imprecise, leaving constant but unknown values in the expression, or the expression is data dependent. Non-coalescable memory access detection assumes that the IACF analysis is precise, and therefore reports any non-constant expressions as non-coalescable addresses. For constant address differences, all accesses within a 256-byte range can be coalesced into a common request. Figure 6.10 presents the coalescing algorithm that GPUCheck uses to calculate the number of memory requests required for a given memory access, given a set of constant offsets C and a number of non-constant

```
Function coalescedRequests(C, nonconst)
    requests={ };
    for c ∈ C do
        fit = false;
        for r ∈ requests do
            if c ≥ r.low && c ≤ r.high then
                fit = true;
            else if c ≥ r.high - 256 && c ≤ r.high then
                r.low = c;
                fit = true;
            else if c ≤ r.low + 256 && c ≥ r.low then
                r.high = c + 8;
                fit = true;
        end
        if fit ≠ true then
            requests.append( (low: c, high: c+8) );
        end
    end
    return requests.size + nonconst;
```

Figure 6.10: Coalescing algorithm in GPUCheck.

accesses *nonconst*. The algorithm computes the number of required requests by greedily calculating the minimum number of 256-byte spans (from *r.low* to *r.high*) that can serve all of the constant offsets in C. The idea is that complete coalescing only occurs when all the accesses are to the same 256-byte cache line. However, our coalescing algorithm considers only the size of requests, and cannot identify when additional requests are required because of unaligned addresses. This imprecision results from canceling of symbolic values, including the base address, when computing differences between the ACF form of the addressing expression for different threads. A memory access that requires more than *maxRequests* per warp is deemed to be non-coalescable, where *maxRequests* is a configurable value. GPUCheck reports such accesses to the programmer.

6.4 An LLVM Prototype for GPUCheck

We have implemented a prototype of GPUCheck on top of the LLVM compiler infrastructure. Since the release of `gpucc` [75], Clang is able to compile CUDA programs to LLVM IR. Using `gpucc`, GPUCheck can operate on CUDA programs just like any other LLVM GPU language.

GPUCheck generates ACF on demand from LLVM's SSA representation, which requires a memory-dependence analysis and a control-dependence graph [16]. Given an expression e, the computation of $ACF(e)$ requires the computation of the ACF for each of the operands of e. GPUCheck memoizes the ACF for these operands and makes them available when needed for future ACF computations. Once the ACF for all operands of e is determined, arithmetic operations can be trivially converted to ACF to compute $ACF(e)$.

GPUCheck calculates predicates only for ϕ-nodes to select the correct operand because the input is already in SSA form. ϕ-nodes merge values over multiple incoming control-flow paths by specifying a mapping of each predecessor basic block to a value. To calculate the ACF expression for a ϕ-node, GPUCheck first calculates a predicate for each predecessor basic block b that evaluates true iff the ϕ-node is immediately preceded at runtime by b. The ACF value for the ϕ-node is then simply the sum of each predicate multiplied by the associated definition.

Let $CDG(e)$ be the set of control dependences for an expression e, as determined by the program control-dependence graph. Let $cond(e)$ refer to the set of conditional expressions corresponding to each control dependence in $CDG(e)$ that results in the execution of e. Finally, let $operand(x)$ denote the expression operands to a ϕ-node x. To determine the value of a ϕ-node, the ACF representation for each operand expression is multiplied against the ACF representation of each conditional expression $cond(e)$ required to reach that operand (Equation 6.2).

$$ACF(\phi) = \sum_{i \in operand(\phi)} \big(\prod_{c \in cond(i)} ACF(c) \big) * ACF(i) \tag{6.2}$$

If a ϕ-node contains a cyclic reference (e.g., loop induction variables), its ACF representation remains symbolic (i.e., $ACF(\phi_{loop}) = [\phi_{loop}]$). Otherwise, for each operand, all conditions required to select that value are converted to ACF representation, and multiplied against the value.

To resolve dependences between operations with memory, the generation of ACF requires a pointer-aware memory-dependence analysis as presented by Horwitz *et al*. [30], that provides a set of dominating stores for each memory load where possible. If a dominating store exists, the ACF representation uses the stored value to be used in place of the load instruction. Otherwise, the load becomes a symbolic reference.

The memory-dependence analysis propagates the effects of the statements of a procedure on memory—through an inter-procedural memory dependence analysis, on the local variables of the caller, and on global variables.

6.5 Evaluation

The main goal of GPUCheck is to assist developers by detecting potential performance-limiting issues in GPU programs at compile-time so that they can be eliminated to improve performance. Ideally, GPUCheck would run every time code is compiled. Therefore, the analysis must be sufficiently fast to avoid interrupting development.

This section reports on a performance evaluation of the LLVM GPUCheck prototype. For this evaluation, we define a divergent branch as one where more than 25% of warps diverge and noncoalescable memory accesses require at least 4 requests per warp. Our evaluation uses the CUDA implementation of the Rodinia heterogeneous computing benchmarks [10], a benchmark suite that attempts to capture a representative sample of GPU computing tasks.

Prior to GPUCheck, developers could only detect GPU performance issues through dynamic profiling: executing programs against testing data, and recording characteristics of the execution. To facilitate profiling, NVidia provides `nvprof`, a dynamic profiler. Both `nvprof` and GPUCheck identify non-coalesced memory accesses and divergent branches. Additionally, `nvprof`

detects a wide variety of other issues, including determining overall occupancy [47].

To provide a comparison, we profiled each Rodinia benchmark through `nvprof`, collecting all `--analysis-metrics`, which includes branch divergence counters for each branch in the source code, dynamic memory coalescing counters for each memory access in source code, as well as occupancy information and the dynamic instruction mix. Through debug information, assembly-level performance counters are associated with lines in the original source. If `nvprof` reports a line as divergent, or a memory operation requiring at least four requests, then that line is deemed either divergent or non-coalescable. We then run GPUCheck, converting the software source into LLVM IR and linking all device modules. Next, all branch instructions and memory operations are analyzed. GPUCheck uses debug source information to report offending source lines. The benchmarks are compiled using NVCC from CUDA 8 at optimization level `-O2` with `lineinfo` included. We then executed and profiled the benchmarks on an NVidia Pascal Titan X, using a host system running on an Intel i7-4770 with 32GB of RAM running CentOS 6. The CUDA implementations of `cfd`, `hybridsort`, `kmeans`, `mummergpu`, and `dwt2d` cannot be compiled with Clang/LLVM due to incomplete CUDA support, and so were not included in this evaluation. Therefore 17 benchmarks from the Rodinia suite are analyzed.

We evaluate GPUCheck through the following research questions: Does GPUCheck provide similar results to dynamic profiling? Do the problems identified by GPUCheck reflect real performance opportunities? Is GPUCheck performant enough to be used during active development?

6.5.1 Does GPUCheck provide similar results to dynamic profiling?

Figure 6.11 shows divergent branches and non-coalescable memory accesses as found by both the NVidia dynamic profiler and by GPUCheck. Differences in methodology cause GPUCheck and `nvprof` to report different but overlapping sets of divergency issues. As shown in Figure 6.11, GPUCheck discovers 170

Benchmark	Divergent Branches	Noncoalescable Accesses
backprop	3/3	7/8
bfs	4/4	4/4
b+tree	11/11	1/1
gaussian	3/3	3/3
heartwall	74/76	17/22
hotspot	4/4	0/2
hotspot3D	2/2	5/5
huffman	9/16	12/12
lavaMD	4/4	0/6
leukocyte	13/14	3/4
lud	5/5	0/0
myocyte	14/14	14/14
nn	1/1	0/1
nw	6/6	8/8
pathfinder	5/5	0/0
srad	10/12	8/8
streamcluster	2/2	2/4

Figure 6.11: Divergency issues found in the Rodinia Benchmark Suite. Black indicates an issue found only by GPUCheck. White indicates an issue found only by `nvprof`. Grey indicates an issue found by both. The adjacent fractions are the number of issues found by GPUCheck, over the total issues found.

divergent branches and 84 uncoalesced accesses, while `nvprof` detects only 52 and 37, respectively. GPUCheck also detects 76.9 % of branch divergences and 51.4 % of noncoalescable accesses detected by `nvprof`.

Ideally, both GPUCheck and `nvprof` would identify all possible issues in all benchmarks, however GPUCheck and `nvprof` have limitations. For a given instruction, `nvprof` aggregates across all executions. A memory operation that generates 32 requests per warp (i.e., fully non-coalescable) 10% of the time would therefore be reported by `nvprof` as requiring 3 requests per access, while GPUCheck would correctly identify the non-coalescable access. A similar strategy is used by `nvprof` for divergent branches. When inspecting branches, GPUCheck and `nvprof` use different thresholds to identify divergence. GPUCheck statically analyzes the branch condition per warp, reporting if more than 40% of warps are divergent, or data-dependent with unknown input. By contrast, `nvprof` uses an unknown threshold of all runtime executions of a warp, causing occasional disagreement on whether a particular branch is divergent.

Due to the ACF difference method of detection, GPUCheck is unaware of any memory requests per warp required due to alignment issues. This can cause partially coalescable accesses to fall below the threshold of 4 requests per warp when analyzed with GPUCheck, but above this threshold when profiled. Additional analysis to determine address alignment could be used to help resolve this issue.

As a static analysis tool, GPUCheck analyzes kernels and code paths that may never be executed at runtime. This feature allows GPUCheck to identify performance issues throughout the code, while `nvprof` is limited to code paths actually exercised by the provided workloads.

We view GPUCheck as complementary to `nvprof`, with each tool providing useful insights unavailable from the other, though there is substantial overlap. GPUCheck discovers new divergency issues, in addition to reporting existing divergency issues earlier in the development process.

6.5.2 Do the problems identified by GPUCheck reflect real performance opportunities?

Of the benchmarks where GPUCheck reported non-coalesced accesses or branch divergence, we select four benchmarks (`gaussian`, `lavaMD`, `nw`, and `srad`) to demonstrate the performance gains that may be obtained even on applications that are expected to be optimized. For each benchmark, we fix any issues detected by GPUCheck, and execute both our modified and the original code three times each on the same experimental machine used to collect the profiling information. We measure speedups over mean kernel execution time. By modifying the benchmarks to act on reports from GPUCheck, GPU kernel performance was improved by 5.5–25.6%.

Gaussian Elimination (`gaussian`)

The `gaussian` benchmark in the Rodinia benchmark suite uses two kernels, `Fan1` and `Fan2`, to solve for variables in a linear system of arbitrary size. Figure 6.12 shows simplified excerpts of both kernels. GPUCheck identifies non-coalesced memory accesses in `Fan1` at Line 97 and in `Fan2` at Line 106 and Line 110, both missed by `nvprof`. However, `nvprof` and GPUCheck identify divergent branches at the boundary checks in `Fan1` and `Fan2`, because the grid geometry of threads and blocks is not an exact match for the problem size. Better tuning of the grid to match the problem size may improve performance, but we did not make that change. Instead, we concentrate on the 2 non-coalesced accesses picked up only by GPUCheck.

In `Fan1`, the first element in each row of the matrix is initialized by a thread. The access stride by adjacent threads is the width of a row, because the matrix is stored in row-major format. Changing the indexing of the matrix to column-major format allows these accesses to be coalesced. With this change, `Fan1` initializes adjacent elements in each thread. When changing storage schemas, it is often necessary to consider how other accesses will be affected. `Fan1` and `Fan2` operate on the same matrix, thus the access pattern in `Fan2` is also changed by this transformation.

Fortunately, Fan2 is a two-dimensional CUDA kernel. Reversing the matrix storage schema in Fan2 is equivalent to exchanging the x and y thread dimensions in the kernel. We do not present the modified code here because the modifications are trivial changes to the array indexing operations replacing threadIdx.x with threadIdx.y and vice-versa.

```
91 __global__ void Fan1 ( ... ) {
92   int xidx =
93       blockIdx.x * blockDim.x +
           threadIdx.x;
94   if(xidx >= Size-1-t) return;
95   int off = Size*(t+1)+t;
96   m_cuda[Size*xidx+off] =
97       a_cuda[Size*xidx+off] /
           a_cuda[Size*t+t];
98
99 }
100 __global__ void Fan2( ... ) {
101   ...
102   if(yidx >= Size-1-t) return;
103   if(xidx >= Size-t) return;
104   ...
105   a_cuda[Size*xidx+yidx+off] -=
106       m_cuda[Size*xidx+off] *
107       a_cuda[Size*t+yidx+t];
108   if(yidx == 0) {
109       b_cuda[xidx+1+t] -=
110           m_cuda[Size*xidx+yidx+off]
                *
111           b_cuda[t];
112 }
```

Figure 6.12: Original gaussian kernel functions (edited for clarity).

After applying the modifications based on the output of GPUCheck, Fan1 runs 11.5% faster and Fan2 runs 5.9% faster than the original code. Overall, the gaussian kernels complete 8.8% faster. The non-coalescable memory accesses reported by GPUCheck are not detected by nvprof, and were previously undetectable by automated means.

LavaMD (lavaMD)

LavaMD is an N-body computation for simulating molecular dynamics interactions. It was originally produced by the Lawrence Livermore National Laboratory, and is derived from the ddcMD application, which performs the same computation sequentially. The lavaMD benchmark's kernel makes heavy use of shared memory buffers, copying memory in tiles for efficient memory access patterns as shown in Figure 6.13. However, the buffer size NUMBER_PAR_PER_BOX is carried over from previous CPU implementations, and set to 100 in the original benchmark. By contrast, this kernel is launched with 128 threads, leaving

nearly a quarter of threads idle through these loops.

Both GPUCheck and nvprof identify the loop at Line 115 in Figure 6.13 as a source of branch divergence. We fixed this issue by modifying the shared memory buffers and number of threads per block to 96 elements. By using a power of two for the shared memory sizes, the allocations divide evenly into the device's shared memory space, improving utilization. Additionally, reducing the number of idle threads allows more throughput per thread. With only these values changed, the lavaMD kernel executes 25.6% faster. There are 6 additional non-coalescable memory accesses detected by nvprof, caused by poor memory alignment while copying data to shared memory. GPUCheck missed these accesses, because it currently cannot identify alignment issues.

```
113 int wtx = threadIdx.x;
114 ...
115 while(wtx<NUMBER_PAR_PER_BOX){

116     rA_shared[wtx] = rA[wtx];
117     wtx = wtx + NUMBER_THREADS
            ;
118 }
```

Figure 6.13: Extract from lavaMD demonstrating buffering in shared memory.

Needleman-Wunsch (nw)

Needleman-Wunsch is an algorithm from the field of bioinformatics used to align proteins and nucleotides. The implementation in the Rodinia benchmark suite executes as a tiled matrix computation with a halo. Figure 6.14 shows the halo initialization from the original kernels. GPUCheck identifies the northwest corner setup as a point of divergence, and the west side setup as an non-coalesced access.

The divergence can be resolved by allowing all threads within the first warp to setup the corner, providing a small improvement by avoiding divergence. In this case, the non-coalesced access cannot be fixed because threads must read across both rows and columns of the matrix. Performance can still be improved, because this non-coalesced access is tightly synchronized. The synchronization points above and below this access hurt performance, because all threads in the block must wait while the access completes. A cursory inspec-

```
119 if (tx == 0)
120       temp[tx][0] = matrix_cuda[index_nw];
121 ...
122 __syncthreads();
123 temp[tx + 1][0] = matrix_cuda[index_w + cols * tx];
124 __syncthreads();
125 temp[0][tx + 1] = matrix_cuda[index_n];
126 __syncthreads();
```

Figure 6.14: Original halo computation in `nw` kernels.

tion shows that all threads write to different elements of `temp`, and thus all of the halo setup can be synchronized together, eliminating extra synchronization in Lines 122 and 124. By providing other warps with work to perform while resolving each non-coalesced access, performance can still be improved. When we apply both transformations, the `nw` kernels execute 5.5% faster.

Speckle Reducing Anisotropic Diffusion (`srad`)

The `srad` algorithm attempts to remove correlated noise, or *speckles* from imagery without destroying underlying information. This algorithm has applications in various imaging technologies such as ultrasound or radar.

One step of the `srad` computation calculates a coefficient of diffusion c, a value between 0 and 1, based on a stencil of nearby values. GPUCheck identifies branch divergence in this code, pointing to repeated conditional memory operations as shown in Figure 6.1 on Lines 23, 24, and 25. Figure 6.2 presents our modified code that uses arithmetic *min* and *max*, which have single-instruction implementations on the GPU, to remove the conditional behaviour entirely. Our modified `srad` kernel executes 30.8% faster. `srad` contains a second kernel, in which GPUCheck found no issues, thus the overall `srad` kernel execution time is improved by 15.7%. Dynamic profiling through `nvprof` fails at detecting this issue, which is representative of the performance gains from repairing branch divergence problems using GPUCheck.

6.5.3 Is GPUCheck performant enough to be used during active development?

GPUCheck is intended to be used actively during the development of GPU algorithms and applications. Therefore, the performance of the analysis is important. The analysis time reported for each benchmark consists of the time for the branch divergence, memory coalescing, and supporting analyses. These times are representative if the application is typically compiled using Clang/LLVM, allowing GPUCheck to actively raise warnings during compilation.

Table 6.1 shows the execution time required for each benchmark. We captured all timing results on a machine with an Intel i7-4770 processor, 32GB RAM, and NVidia Titan X Pascal GPU. GPUCheck completes its analysis in 90 ms to 5.5 s for each benchmark, with a mean analysis time of 596 ms, typically within Nielsen's recommended threshold for interactive user interfaces [46]. Therefore, GPUCheck can be integrated seamlessly into existing development environments, without adding much overhead to the normal workflow of developers. The notable outlier is `heartwall`, which executes for over 5 seconds. The `heartwall` benchmark program contains a large number of thread-dependent code paths, leading to ACF expressions being generated for 171 expressions. By comparison, only 46 expressions are inspected in `myocyte`, the next slowest analysis. Because GPUCheck scales with code size and not with workload size, GPUCheck can more efficiently handle benchmarks with long execution time.

6.6 Concluding Remarks

In this paper, we introduce GPUCheck, a static analysis tool that reasons about GPU program behaviour. Compared to profiling, GPUCheck has the following advantages:

1. GPUCheck's analysis time scales with the code size, while profiling time scales with actual program execution time. The difference in times can be very significant, because GPU programs tend to be highly parallel.

Benchmark	Branches	Accesses	GPUCheck Time (s)	Profiling Time (s)
backprop	4/13	18/145	0.14	2.38
bfs	6/12	14/67	0.12	24.38
b+tree	14/33	25/214	0.30	4.73
gaussian	4/9	7/69	0.09	5.02
heartwall	90/258	81/1364	5.53	281.23
hotspot	16/48	3/194	0.26	1.86
hotspot3D	4/16	21/195	0.43	105.55
huffman	13/41	28/277	0.28	41.26
lavaMD	6/29	9/162	0.15	21.16
leukocyte	15/66	3/332	0.21	57.79
lud	5/77	11/272	0.29	36.36
myocyte	27/4216	19/7499	1.25	1880.55
nn	1/2	5/32	0.09	1.37
nw	6/50	10/280	0.19	201.21
pathfinder	10/37	3/111	0.14	6.73
srad	18/58	25/540	0.53	8.04
streamcluster	2/9	9/82	0.13	2080.28

Table 6.1: Execution time for GPUCheck vs dynamic profiling. Branches and accesses show the number of instructions requiring ACF analysis over all instructions analyzed.

2. Since GPUCheck uses static analysis, it needs no test data and takes into consideration all possible executions through the code. In contrast, profiling can detect only issues that actually occur in the test data.

3. GPUCheck does not require a physical GPU to identify problems, because it does not execute GPU code. When there is competition for the use of GPU computation, GPUCheck frees up more GPU time for useful work.

GPUCheck detects branch divergences and non-coalescable memory accesses on 17 programs from the Rodinia benchmark suite. Fixing those issues improves performance, in terms of running time, by 5.5–25.6%. A prototype demonstrates that GPUCheck is complementary to dynamic profiling, and represents a strong foundation on which future analysis of parallel systems can be built.

Chapter 7

Run-Length Base-Delta Encoding for High-Speed Compression

7.1 A New Data Compression Algorithm

This paper introduces Run-Length Base-Delta (RLBD) encoding, a lightweight software compression scheme suitable for very high-speed transfers. RLBD builds upon ideas developed for cache compression, to produce a generally-applicable compression algorithm.

Base-Delta Intercept (BΔI) [54] exploits the key idea that values stored close to each other in memory tend to be distributed within a small value range. Sequences of values meeting this criteria are said to exhibit *value locality*. When this property holds, the values stored within a cache line are likely to be similar. Therefore, the differences between the values in a cache line can be stored in less space than the values themselves. RLBD generalizes this idea by assuming that arbitrary data that needs to be transferred over a network likely exhibits value locality. This value-locality property indeed holds for many real-world integer arrays and matrices, but interestingly can also be applied to floating-point types. Excluding the sign bit, binary floating-point representations, when interpreted as integers, share ordering (i.e. an integer comparison of positive floating-point values returns the correct result) [23]. This property of the floating-point representation leads to the value-locality property to be true for many floating-point arrays and matrices.

Most software compression schemes relies on storing patterns that have been observed into a history buffer. Compression is achieved by storing/-transmitting each pattern once and then storing/transmitting a pointer to the buffer for each pattern occurrence. This scheme allows repeated sequences and subsequences to be efficiently compressed, but also means that the overall compression ratio is dependent on the content of the history buffer. For such compressing schemes, dividing data to be compressed into segments, as done in the BΔI solution, reduces the possible compression. RLBD, just like BΔI, uses value locality to provide compression, but it does not limit the compressed segment to a fix size such as the length of a cache line or of a memory page. Therefore, as opposed to traditional pattern recognition and history buffers, RLBD allows segmentation and parallelization without sacrificing compression ratios. Value locality should be equally applicable regardless of starting point. Thus the data to be compressed can be partitioned to suit highly parallel architectures such as modern GPUs. Similarly, decompressors do not need to construct a history buffer, and can start at any valid RLBD frame.

This paper makes the following contributions:

- a new high-speed compression algorithm, Run-Length Base-Delta (RLBD), which exploits low dynamic range in high-speed data transfers to perform compression with minimal memory and computation overhead.
- a directory structure suitable for enabling parallel compression and decompression on GPUs, such that RLBD compression and decompression can be offloaded to a GPU compute accelerator.
- an evaluation of RLBD on datasets representative of real-world data transfers.
- an evaluation of the proposed algorithm on synthetic datasets to determine the expected compression and decompression throughput for both CPUs and GPUs.

7.2 RLBD Compression Format

RLBD uses the intuition that data that is stored nearby in memory is likely to contain similar values. This is often the case, for instance, with arrays or matrices of numbers use for graphics and for scientific computation. Therefore, compression may be achieved by replacing the full binary representation for all the similar values with a selected base value, and the differences between the base and subsequent values. This difference is the delta between the current value and the base value. This computed delta can be stored in fewer bytes than the original values because the values are similar, resulting in successful compression.

A Run-Length Base-Delta encoded stream is formatted as at least one h-byte header, followed by n deltas of size b, where n and b are specified in the header, collectively forming a frame. The header is formatted as an v-byte base value followed by an s-byte scheme identifier and a c-byte counter, where $h = v + s + c$.

For instance, a given RLBD encoding implemented with $h = 16$, $v = 8$, $s = 2$, and $c = 6$ has a header configuration as shown in Figure 7.1. The header is of fixed size, and specifies the encoding schema to the next header, as well as the position of the next header.

The base field in the header defines the point of reference for all deltas in this header, as well as the first bytes of the decoded stream. The base value is either 4 or 8 bytes, as determined by the schema. Valid options for the Schema Identifier for this example are shown in Table 7.1. A hexadecimal number is preceded by 0x.

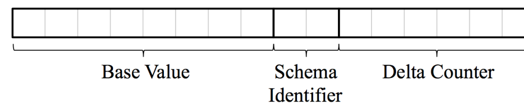


Figure 7.1: Configuration of Header for $h = 16$, $v = 8$, $s = 2$, and $c = 6$. Each small light grey box represents one byte.

Appended to the header is an array of n d-byte deltas, where d is determined by the Schema in the header, and n is the value written to the Delta

Schema	Identifier
8-byte Base, 0-byte Deltas	0x0800
8-byte Base, 1-byte Deltas	0x0801
8-byte Base, 2-byte Deltas	0x0802
8-byte Base, 4-byte Deltas	0x0804
8-byte Base, 8-byte Deltas	0x0808
4-byte Base, 1-byte Deltas	0x0401
4-byte Base, 2-byte Deltas	0x0402

Table 7.1: Identifiers for various possible schema.

Counter. The header and array of deltas describe a sequence of $n+1$ v-byte values, where v is the base size specified by the Schema. The header and array of deltas are together referred to as a frame. The decoded sequence corresponding to a frame is determined by the following function:

$$value[i] \leftarrow \begin{cases} base, & \text{if } i = 0 \\ base + delta[i-1], & \text{otherwise} \end{cases}$$

As an example to illustrate the RLBD encoding, consider the transmission of the following sequence of data using an eight-byte base with a one-byte delta:

```
0x0000000080008000 0x0000000080008008
0x0000000080008010 0x0000000080008018
0x0000000080008020 0x0000000080008028
0x0000000080008030 0x0000000080008038
0x0000000080008040 0x0000000080008048
0x0000000080008050 0x0000000080008058
```

With the RLBD encoding illustrated in Figure 7.1 and Table 7.1, this data is represented in compressed form as follows:

```
0x0000000080008000 0x080100000000000B
0x0810182028303840 0x485058
```

resulting in a compression ratio of $96/27 = 3.55$.

7.3 RLBD Compression and Decompression

Lossless software compression schemes typically detect repeating patterns to perform compression through the replacement of such patterns by a reference to their storage. By contrast, RLBD exploits only the low variation in value between data items that appear close to each other in the data stream. Therefore RLBD does not need to store or inspect past values.

7.3.1 Serial Compression

Compression can be performed reading each incoming value from the uncompressed stream exactly once, using the algorithm shown in Figure 7.2. An RLBD compressor composes a series of frames by first selecting a schema then computing and storing deltas from the first value until either a maximum is reached, or a delta cannot be encoded using the specified schema.

Schema selection is critical to the performance of the RLBD compressor to maximize both the compression ratio and the compression throughput. We propose a 1-Lookahead (1LA) schema selection algorithm, shown in Figure 7.3. The 1LA algorithm reads both a 4- and an 8-byte bases, as well as the immediate following value, selecting the schema with the highest compression ratio that is valid for the following value. The 1LA algorithm is unique in that it allows for an implementation where every value in the uncompressed stream is read exactly once, requiring no caching for performance. However, it may suffer from overhead because once a schema is selected a full header will be written by the compressor. If too few deltas can be encoded, the compressed output may be marginally larger than the input.

Once a schema is selected, the compression algorithm could use it until a delta can no longer be encoded. However, different schemas produce different compression ratios, ranging from ∞ to 1.0 excluding headers, but are more generally applicable as compression ratios are reduced. It therefore makes sense to occasionally stop to inspect whether a more efficient schema can be selected. However, the better the current schema, the less likely that an improvement is possible. Therefore, in the prototype implementation of RLBD

```
Function Compress(byte *input, byte *output, size)
    while size > 0 do
        (schema, baseSz, deltaSz, base) = Schema(input);
        maxBytes = min(128*baseSz/deltaSz, size);
        byte* header = output;
        output += 16;
        deltas = 0;
        while maxBytes > 0 do
            delta = input[0..baseSz]-base;
            if minbytes(delta) > deltaSz then
                break;
            end
            output[0..deltaSz-1] = delta;
            deltas += 1;
            output += deltaSz;
            input += baseSz;
            maxBytes -= baseSz;
            size -= baseSz;
        end
        header[0..7] = base;
        header[8..9] = schema;
        header[10..15] = deltas;
    end
```

Figure 7.2: RLBD Serial Compression Algorithm

a compression-ratio dependent maximum number of bytes to encode is set:

$$maxBytes = min(128 * compression\ ratio, 2^{48} - 1)$$

Using this methodology, the worst-case compression ratio (random noise, encoded using 8-byte deltas) is limited to 0.941 because a header will be introduced at most every 128 bytes. By comparison, the compression algorithm will encode as many 0-byte deltas as can fit into a header. Scaling the frequency at which the input stream is inspected for schema selection may lead the compression algorithm to discover more opportunities for better compression.

```
Function Schema(byte *input)
    base8 = input[0..7];
    next8 = input[8..15];
    diff8 = next8-base8;
    deltaSize8 = minbytes(diff8);
    base4 = base8[0..3];
    next4 = base8[4..7];
    diff4 = next4-base4;
    deltaSize4 = minbytes(diff4);
    if deltaSize8 == 0 then
        return (0x0800, 8, 0, base8);
    else if deltaSize8 == 1 then
        return (0x0801, 8, 1, base8);
    else if deltaSize8 == 2 then
        return (0x0802, 8, 2, base8);
    else if deltaSize4 == 1 then
        return (0x0401, 4, 1, base4);
    else if deltaSize8 == 4 then
        return (0x0804, 8, 4, base8);
    else if deltaSize4 == 2 then
        return (0x0402, 4, 2, base4);
    end
```

Figure 7.3: RLBD 1-Lookahead Schema Selection

7.3.2 Serial Decompression

The serial decompression algorithm is very straightforward as shown in Algorithm 7.4. This algorithm simply parses the compressed stream frame-by-frame. There is no need for caching history, the decompression algorithm reads data from the compressed stream exactly once.

```
Function Decompress(byte *input, byte *output, size)
    while size > 0 do
        base = input[0..7];
        schema = input[8..9];
        baseSz = schema >> 8;
        deltaSz = schema && 0x00FF;
        deltas = input[10..15];
        input += 16;
        size -= 16;
        while deltas > 0 do
            output[0..baseSz] = input[0..deltaSz] + base;
            input += deltaSz;
            output += baseSz;
            size -= deltaSz;
            deltas -= 1;
        end
    end
```

Figure 7.4: RLBD Serial Decompression Algorithm

7.4 GPU-Accelerated RLBD

RLBD compression may benefit from CPU parallelization, but it is being considered for use in supercomputing environments, where CPU utilization is at a premium. However, compute accelerator devices like GPUs require parallelism for efficient usage. We therefore present parallel compression and decompression algorithms suitable for such devices.

7.4.1 Parallel Compression

A simple approach to perform compression in parallel consists in dividing the data to be compressed into n partitions, thus allowing n compressors to independently compress their assigned section. This strategy is suitable to RLBD because it requires no caching of the data stream frequent patterns to produce efficient compression, and therefore the compression can start at an arbitrary point in the uncompressed stream without any significant effect on compression ratios.

To similarly enable parallel decompression, the decompression algorithm

must know the uncompressed offset associated with the header of the first frame in the partition, to decompress to the appropriate location. To enable parallel decompression, we introduce a Data-Segment Directory (DSD). To perform decompression of an RLBD stream, a decompressor requires a target address for the uncompressed data, a source address for the compressed data, and the length of the compressed data. To begin decompression at an arbitrary target address, a warm-up interval is also required to allow a header to be partially processed before decompression actually begins writing uncompressed data. Together, the source address, target address, compressed size, and warm-up interval form an initialization vector (IV). These IVs can be generated during compression, and appended to the end of an RLBD stream at the cost of a small overhead. The set of IVs forms a DSD.

Parallel RLBD compression therefore involves first separating the uncompressed data into partitions and compressing them in parallel, then combining each compressed partition into a single compressed body and assembling a DSD. To enable similar levels of parallelism for both compression and decompression, we construct a DSD Directory with IVs pointing to the start of each partition. Figure 7.5 shows the 2-step process, and resulting compressed data and DSD assuming four partitions.

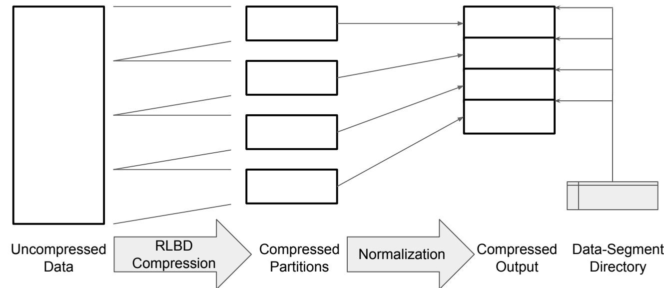


Figure 7.5: 2-Stage parallel compression flow

GPU Compression

The overhead required for both the post-processing step and for the spawning of threads makes CPU-parallel RLBD infeasible in practice for current CPUs. However, the GPU architecture with its highly-parallel design and extremely fast GDDR5/HBM2 memory [68] lends itself well to parallel processing. In

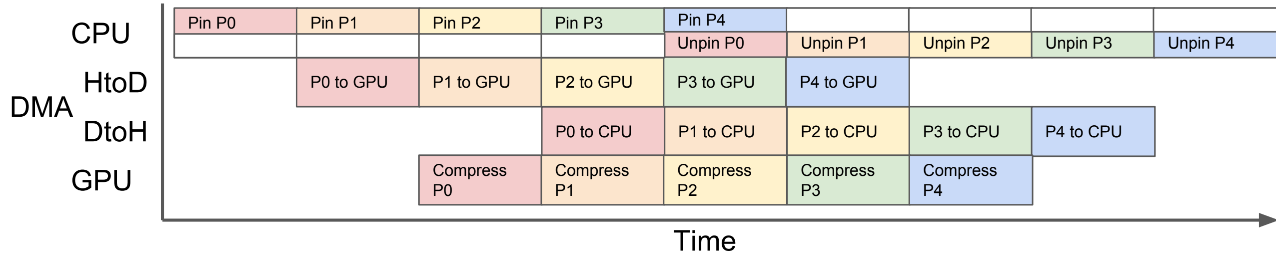


Figure 7.6: GPU Compression Pipeline

addition, GPUs have two levels of parallelism: block-level independent parallelism and thread-level cooperative parallelism.

When compressing a frame, threads cooperatively compute deltas from the shared base. GPUs execute instructions for a full warp of 32 threads simultaneously, so if fewer than 32 threads have useful work, computation is wasted. To reduce instances of extremely short RLBD frames, compression on the GPU increases the lookahead to 8LA. 8LA ensures for each frame that at least 8 of 32 threads can perform work, increasing GPU utilization. While increasing lookahead could reduce the compression ratio by ignoring short compression opportunities, it also prevents overly short frames that would be larger than the uncompressed data. These effects tend to cancel each other in practice, as shown in the evaluation.

GPUs maintain a separate memory from the CPU, and manage data transfers through Direct Memory Access (DMA) hardware. DMA hardware allows the memory transfer to occur without involving either the CPU or GPU, and enables pipelining to hide transfer costs. However, to allow the CPU to perform other tasks while transferring, the operating system must be informed about the memory pages involved to prevent relocation or paging out. Such memory is *pinned* to allow for asynchronous memory copies.

Pinning overly large amounts of memory can hurt system performance, as it prevents the operating system from reclaiming memory. In addition, we wish to hide as much of the GPU transfer costs as possible. Therefore, data to be compressed on the GPU is first decomposed into fixed-size segments, which are copied into a pinned buffer by the CPU, transmitted to the GPU, compressed on the GPU, transmitted back to the CPU, and copied out of the

pinned buffer in a pipelined manner, as shown in Figure 7.6. By pipelining, we can maintain 4 pinned buffers: 1 each for the DMA transfers, and 1 each for the CPU copying data in and out. The compression run on the GPU is the longest step in the pipeline, leaving the CPU enough time between copies to choreograph subsequent pipeline stages.

Because GPU threads cannot synchronize across blocks, GPU compression requires two independent kernels. The first kernel assigns 1 partition to each block, and generates compressed partitions, while the second kernel copies each partition into contiguous compressed data, and constructs and appends the DSD for parallel decompression.

GPU Decompression

As above, independent parallel decompressors are mapped to GPU block-level parallelism. However, thread-level cooperative parallelism allows each decompressor to operate frame-by-frame, with threads cooperatively expanding the deltas within each frame.

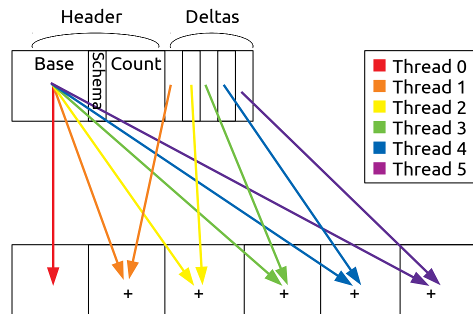


Figure 7.7: Visualization of thread-level cooperation within a GPU block during decompression

Each block is assigned a single partition, read from the DSD to allow for independent parallelism. All threads within a block independently read the frame header, then threads are sequentially assigned deltas until the entire delta count has been processed. Thread 0 first copies the base verbatim, before cooperatively expanding deltas with the other threads in the block. Figure 7.7 shows thread-cooperative decompression.

Decompression suffers from the same warp usage problem as compression, in which threads may be idle. However, the DSD contains final target information so no post-processing step is required for decompression, allowing decompression to achieve much higher throughput than compression. In addition,

it would be typical that the GPU RLBD implementation would decompress RLBD data compressed on a GPU, allowing the decompression algorithm to take advantage of the longer RLBD frames generated by GPU compression. Pipelining is also implemented in the same manner as compression, to hide the costs associated with memory transfer to and from the GPU.

7.5 Evaluation

RLBD compression is designed to achieve extremely high-speed software compression, to improve throughput on interconnects that operate at throughputs too large for traditional software compression. Therefore an evaluation of RLBD must determine if it is indeed faster than traditional schemes. To be relevant a software compression must be able to provide data at a speed that is consistent with the interconnection network that is used for data transmission. Therefore, it is also important to determine if RLBD is fast enough to work with the high-speed interconnects typically available in modern supercomputers. Finally, the evaluation must determine if RLBD compression is effective on real-world data, and what level of increased throughput can be expected from its adoption.

To integrate RLBD with modern high-speed interconnects, a Compression-Transmission-Decompression
pipeline (CTDP) must be constructed to improve overall throughput. Transmitted data should be discretized into segments, and segments should be sent through the pipeline so that Compression and Decompression can be overlapped with transmission. For large data sets, such a pipeline will improve the data transmission throughput a the cost of increased overall latency. For short data transfer, the increased latency may be unacceptable, and in that case a threshold can be used to determine when the CTDP should be enabled.

Name	CPU	RAM	GPU
sandybridge	Intel Xeon E5-2530 v2 CPU @ 2.60GHz	64GB @ DDR3 1333	Nvidia Tesla K20m
power8	IBM Power8 8286-42A 24-Core CPU @ 3.52GHz	512GB @ DDR4 1600	Nvidia Tesla P100

Table 7.2: Machines Used for performance testing

7.5.1 Is RLBD faster than traditional software compression schemes?

This evaluation compares RLBD against LZ4 [12] in "fast 8" and "fast 32" modes, as well as agaist Brotli [1], Gipfeli [40], and Snappy [24]. For all experiments, data is first loaded in-memory, then compression and decompression are independently timed. Compression and decompression are each performed five times per input, and the mean throughput is reported. This prototype of RLBD also has a GPU implementation. For the GPU algorithm data transfers to and from the device are not included because those times would be hidden by the pipelining.

The experiment is run on two machines with the configurations shown in Table 7.2. For performance tests, Intel CPU implementations were compiled with gcc 4.9.1 with `-O3 -march=native`, while Power8 CPU implementations were compiled with gcc 5.4.0 with `-O3 -march=native`. GPU implementations were compiled with CUDA 8, with `-O3 -arch=sm_35` or `-arch=sm_60`, as appropriate.

The goal of this performance evaluation is to provide insights into the performance that could be expected in an MPI-like [29] supercomputing context, using RLBD to compress transfers between nodes. Finding datasets that replicate this use case is surprisingly difficult because supercomputing benchmarks tend to transmit either zeroes or random data. Neither the very high compression rate for zeros nor the very low compression rate for random numbers would reflect the typical use case for RLBD. Thus, this evaluation uses five machine-learning datasets obtained from the UCI Machine Learning Repository [41], shown in Table 7.3. These datasets are available in CSV form, which is not representative of the sort of structured data that would be transmitted

Dataset	Entries	Attributes
commviol [9]	2215	2 strs, 4 i8s, 43 i32s, 98 f32s
gisette [27]	6000	46 i8s, 4955 i32s
secom [41]	1567	590 floats
weightlift [69]	4024	2 strs, 22 i8s, 26 i32s, 109 f32s

Table 7.3: Datasets used for real-world testing

between the memories of supercomputing nodes.

To more accurately represent the structured data transfers that would occur in a supercomputing application, we wrote a packing program that scans the attributes, and determines if they should be represented as a byte, 32-bit int, 64-bit int, float, or string. Then, this program output the file in two different formats:

1. **Array of Structs:** The data is packed row-by-row. This is most similar to the original CSV, but has poor memory locality for computation. This format also has poor value locality because each subsequent attribute represents different data.

2. **Struct of Arrays:** The data is packed in an array-of-attributes format. This method of data packing allows SIMD-style parallelism to perform with superior memory characteristics because attributes of sequential entries are side-by-side. This method of storage also exhibits excellent value locality because all instantiations of an attribute are placed together in memory. This data layout is quite desirable for enabling efficient parallelism, to the point that modern compilers attempt to restructure data to fit this model [15].

Figure 7.8 shows the compression throughput for RLBD on the Sandybridge. This evaluation uses two versions for each of the datasets listed in Table 7.3: an array-of-structures (aos) and an struct-of-arrays (soa). RLBD CPU outperforms all other CPU compression algorithms on seven of the dataset versions evaluated. For the secom(soa) version the lz4-fast 32 results in a

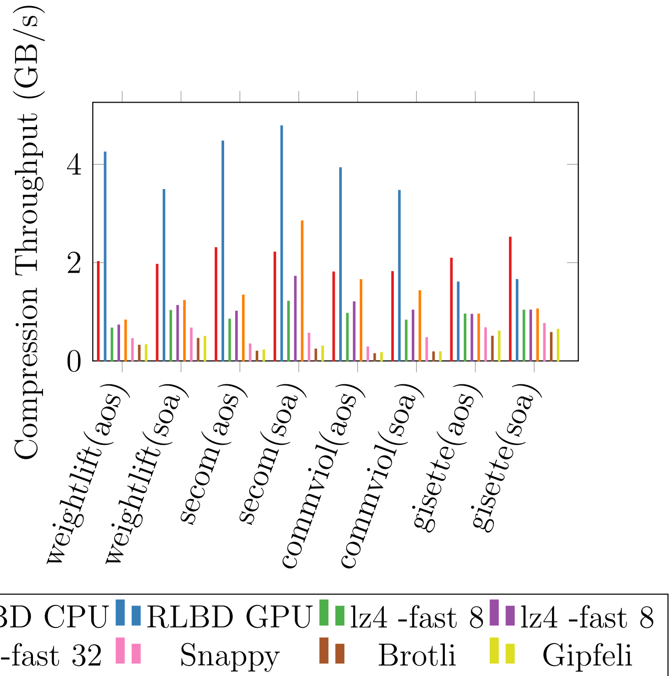


Figure 7.8: Compression Throughput by Algorithm on the Sandybridge machine

throughput that is 28% higher than the RLBD CPU version. While the performance of lz4-fast 32 on this dataset is interesting, its variable throughput performance on the other datasets prevents it from being applied to improve network throughput. In many cases lz4-fast 32 would be the limiting factor, actually harming overall throughput. For all dataset versions RLBD CPU is able to maintain a throughput of more than 1.81 GB/s. RLBD on GPU is much faster than the CPU algorithm on most inputs, with the exception of gisette. Gisette represents pixel data, which exhibits many relatively short (8 delta) compressible segments, which reduces the cooperative parallelism usable by the GPU during compression.

RLBD, in both the CPU and GPU implementations, maintains a substantially higher minimum throughput than competitors on the inputs presented.

Is RLBD throughput schema- or data- dependent?

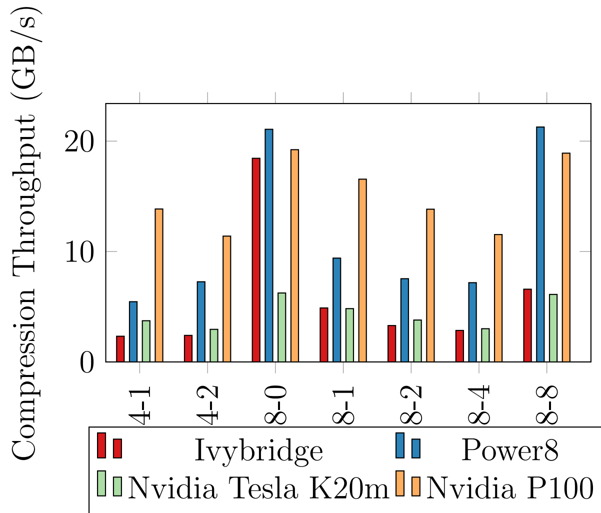


Figure 7.9: Throughput of synthetic data compression by schema

RLBD throughput varies heavily on the inputs presented, and RLBD GPU suffers when there are short runs, but RLBD internally relies on schemas to store deltas in a compressed form. To establish whether different schemas have different performance, we created a synthetic data generator. We generate a 16MB file consisting of 256B sequences that can be compressed using a particular RLBD schema, by selecting a random 8-byte base, then generating 32 random deltas and adding them to the base. The range of the deltas is determined by the scheme being generated. The throughput for each schema is shown in Figures 7.9 and 7.10.

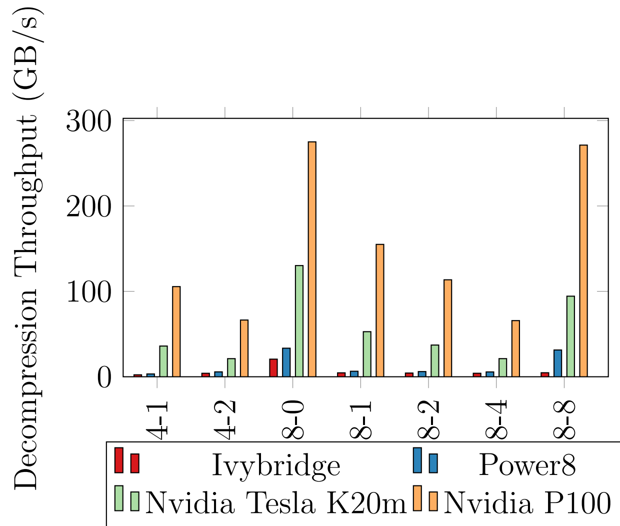


Figure 7.10: Throughput of synthetic data decompression by schema

The compression throughput shown in Figure 7.9 reveals that the Power8

CPU yields a throughput that is nearly the double of the throughput for the Sandybridge core in all cases. This difference is in part due to the different frequency of operation of the two processors — the Power8 core operates at 2061MHz, compared to 1200MHz in the intel core — but this difference alone does not explain all the performance differences.

To better understand the lower throughput by Sandybridge in comparison with the Power8, it is necessary to consider the main loop executed by the compression algorithm and how the statements are scheduled in thes processors. Consider Schema 8-8, where the Power8 CPU processes 10B/cycle: The loop has been simplified to 3 vector operations, an increment, and a branch. The CPU has 4-wide issue, 2 vector pipelines, and out-of-order execution, enabling it to complete a loop iteration every cycle and a half. The Sandybridge core also has 3 vector operations, but 3 increments and a branch. However, there is only a single vector pipeline per core, so a loop iteration can be completed only every 3 cycles. Most interestingly, memory bandwidth and latency are irrelevant to single-thread performance, as prefetching takes care of cache effects, and the limiting factor is the CPU's ability to issue instructions.

The Power8, Tesla K20, and P100 results show that compression throughput is maximized when compressing runs of the same value (8-0) or incompressible data (8-8), with performance tapering off for other schemas. The Sandybridge processor is however memory-bound, and the additional writes required by the incompressible data dramatically hurt performance.

However, the peak performance numbers shown in Figure 7.9 are not reflected in any of the real-world inputs. The synthetic results allow for near-perfect branch prediction and memory prefetching, providing long sequences of data for each RLBD frame. As a result, the only limiting factor is the instruction throughput of the CPU. In the previous real-world data, however, the selected scheme changes frequently in a data-dependent manner, hurting throughput. Similarly, the main compression loop has an unknown, data-dependent, bound. These effects slow throughput dramatically, and result in the performance differences observed between Figures 7.8 and 7.9. The decompression performance shown in Figure 7.10 is similarly reduced.

Interconnect	Transmission Throughput	Throughput Reqd (DS=0.92)	Throughput Reqd(DS=0.43)
10GbE	1280 MB/s	1391 MB/s	2976 MB/s
40GbE	5120 MB/s	5565 MB/s	11906 MB/s
PCIE 2.0 x16	16384 MB/s	17808 MB/s	38102 MB/s
PCIE 3.0 x16	32768 MB/s	35616 MB/s	76204 MB/s
NVLink 1.0	81920 MB/s	89043 MB/s	190511 MB/s

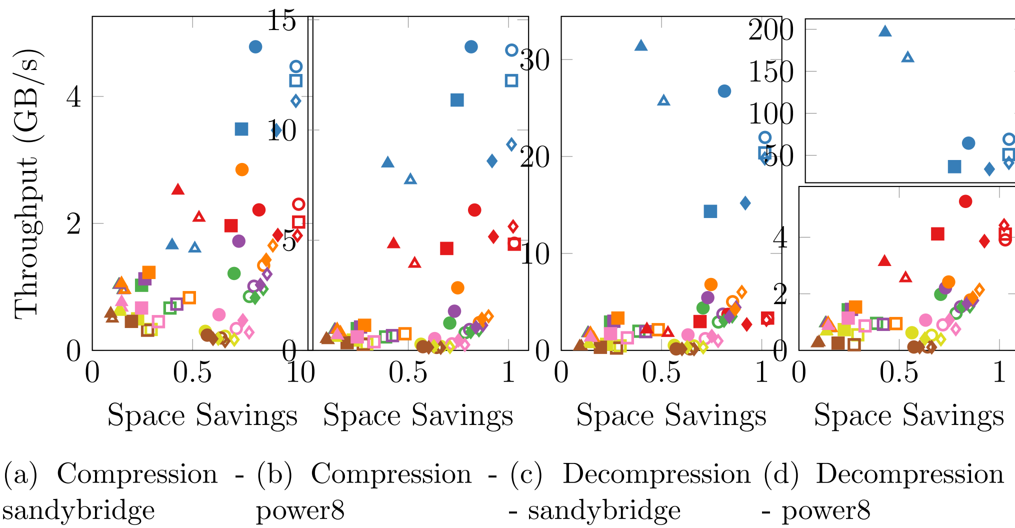
Table 7.4: Common high-speed interconnects, their throughput, and compression/decompression throughput requirements by data savings ratio

RLBD performance is affected by both the schema used and the length of the RLBD frame. However, the effect appears to mainly reduce peak performance.

7.5.2 What compression rates are required by modern supercomputing interconnects?

The throughput requirements for compression and decompression is determined such that each stage in the compression-transmission-decompression pipeline can be performed independently. Table 7.4 shows throughput rates for a variety of interconnects available in recent supercomputers: 10GbE and 40GbE are networking fabrics, and would be representative of inter-node communication, while PCIE and NVLink are used for intra-node communication with attached compute-accelerator devices. DS stands for data savings, and is calculated as $size_{compressed}/size_{original}$. The values DS=0.92 and DS=0.43 were selected because they represent boundary cases of compression ratios achievable by RLBD, as determined below in 7.5.3.

In addition to the native throughputs of these interconnects, Table 7.4 includes required compression and decompression throughputs for various levels of compression. As the data savings increases, the greater the effective throughput of the link.



(a) Compression - sandybridge (b) Compression - power8 (c) Decompression - sandybridge (d) Decompression - power8

Data Format	Compression Algorithms		Dataset
■ Struct of Arrays	● RLBD GPU	● RLBD CPU	♦ commviol
□ Array of Structs	● Brotli	● Snappy	■ weightlift
	● lz4 -fast 8	● lz4 -fast 16	● secom
	● lz4 -fast 32	● Gipfeli	▲ gisette

Figure 7.11: Throughput vs Data Savings

7.5.3 Is RLBD compression effective on real-world data?

The goal of RLBD is to enable compression on high-speed interconnects such as 10GbE or 40GbE, therefore increasing network throughput. RLBD achieves compression by exploiting value locality, in which adjacent locations tend to hold similar values. The datasets used were packaged in two ways:

When the data is organized as an array of structs (aos), all the attribute of each struct are packed together, each possibly being of a different type with very different values. Thus, as would be expected data formatted in this way typically exhibit low value locality, and as expected, RLBD is unable to compress 3 of the 4 datasets in this format, instead using up to an additional 2.3% of space. On the remaining dataset, gisette, RLBD reduced the filesize by 47%. Compression was possible because attributes in gisette represent pixel data, which exhibit value locality between adjacent pixels.

When the data is stored in the struct of arrays (soa) form, all attributes of the same type are stored contiguously, thus leading to higher value locality. This format more closely represents possible gains from transmitting arrays of values. As expected, RLBD achieves compression on all the data sets transmitted in this form. The RLBD file-size reduction varied within the 8-57% range. Even on data in this form, RLBD does not achieve dramatic compression ratios in most cases: Brotli compresses data in both formats from 33-91%.

When input data matches the value-locality assumptions of RLBD, data savings can vary between 8-57%. When it does not, the overhead of using RLBD is usually negligible, at worst 2.3%.

7.5.4 What throughput improvements can be expected when implementing RLBD?

At a data savings of 0.43 (57% file size reduction), a compression and decompression throughput of 2976 MB/s (see Table 7.4) would be required to saturate a 10GbE connection. Unfortunately, RLBD on the Sandybridge CPU can maintain a geomean throughput of only 2.08 GB/s. Therefore, when the

data is well-compressed the throughput of data compression becomes a limiting factor. However, RLBD compression and decompression on Power8 are more than fast enough for a 10GbE link, at geomean 4.98 GB/s and 3.84 GB/s respectively, though the decompression throughput prevents RLBD from being useful over 40GbE. Even on Power8, the geomean decompression throughput means that RLBD would actually degrade network throughput over 40GbE.

However, as shown in Table 7.4, GPUs are connected to CPUs over a far faster link than even a 40GbE network connection. If the Compression-Transfer-Decompression pipeline is extended to use a GPU accelerator, forming a 7-stage pipeline including GPU transfers on each side, then we can exploit RLBD's dramatic GPU performance. The Nvidia P100 included in the Power8 system is connected by PCIE 2.0, meaning compression throughput would be the pipeline limiting factor, at geomean 4.98 GB/s. This performance allows compression even over 40GbE connections, achieving up to 7903 MB/s, a 35% improvement over uncompressed 40GbE, which transmits only up to 5120 MB/s.

When RLBD is used to compress transfers over 10GbE, effective transfer speeds up to 2976 MB/s can be obtained on Power8, and up to 2.08 GB/s on Sandybridge processors. Newer supercomputers with 40GbE and P100 GPUs or newer can use GPU acceleration for compression, achieving effective transfer speeds up to 7903 MB/s.

7.6 Concluding Remarks

RLBD compression presents a new option for software compression over high-speed networking connections, and can enable substantial througput improvements. Using a single CPU core, RLBD can be used over 10GbE to improve throughput put to 57%. With the inclusion of modern GPU accelerators typical of current and upcoming supercomputers, RLBD can also be used to improve throughput on 40GbE fabrics.

RLBD fills a niche in the compression ecosystem, providing reasonable compression ratios at extremely high speed, where better compression has

insufficient performance.

Chapter 8

Conclusion

This compilation represents a series of methods to improve the performance and usability of compute accelerator devices, and utilize GPUs to improve supercomputing network performance.

Chapter 4 argues for combined compilation using OpenCL for FPGA targets, using traditional compiler analyses and transformations to adapt OpenCL code originally written for GPUs. By inspecting and modifying host code, speedups of up to 6.7x were achieved.

Chapter 5 explores the problem of mapping parallel OpenMP code to GPU architectures, using an in-depth data analysis to improve an industry-standard heuristic. When our proposed solution is used, benchmarks experience a geomean speedup of 25%.

Chapter 6 introduced GPUCheck, a static analysis tool for identifying performance problems due to GPU thread divergence. GPUCheck found performance problems in every benchmark in the Rodinia benchmark suite, and further investigation found that benchmark kernels experienced speedups of 5-30% when these problems were fixed.

Finally, Chapter 7 introduced Run-Length Base-Delta encoding, a high-speed compression algorithm capable of compression ratios up to 57% on representative data, at throughputs of geomean 10.40 GB/s on an Nvidia P100. While compression throughput was insufficient for the originally intended purpose of compressing CPU-GPU transfers, RLBD is sufficiently performant to be employed on 40GbE, improving inter-node transfers in modern supercom-

puters.

References

[1] J. Alakuijala and Z. Szabadka, "Brotli compressed data format," Tech. Rep., 2016. 10, 96

[2] Altera. (). Hello world design example, [Online]. Available: `https://www.altera.com/support/support-resources/design-examples/design-software/opencl/hello-world.html` (visited on 04/04/2017). 27

[3] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 259–269. 58

[4] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 163–174. 17

[5] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs," in *International conference on Supercomputing (ICSC)*, 2008, pp. 225–234. 10

[6] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001. 48

[7] O. Burn, *Checkstyle*, `https://checkstyle.sourceforge.net`, Accessed: 2017-04-27, 2003. 9

[8] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2012, pp. 141–151. 5, 17

[9] U. S. D. o. C. B. o. t. Census, *Census of population and housing, 1990 [united states]: Summary tape file 3a, record sequence example file*, 2006. DOI: `10.3886/ICPSR09592.v1`. [Online]. Available: `http://doi.org/10.3886/ICPSR09592.v1`. 97

[10] 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 *IEEE International Symposium on Workload Characterization (IISWC)*, 2009, pp. 44–54. 21, 58, 59, 74

[11] B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens, "Improving your software using static analysis to find bugs," in *Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM, 2006, pp. 673–674. 9

[12] Y. Collet, *Lz4 lossless compression algorithm*, 2013. 10, 12, 96

[13] K. E. Coons, B. Robatmili, M. E. Taylor, B. A. Maher, D. Burger, and K. S. McKinley, "Feature selection and policy optimization for distributed instruction placement using reinforcement learning," ser. PACT '08, Toronto, Ontario, Canada: ACM, 2008. 17

[14] D. J. Craft, "A fast hardware data compression algorithm and some algorithmic extensions," *IBM Journal of Research and Development*, vol. 42, no. 6, pp. 733–746, 1998. 12

[15] S. Curial, P. Zhao, J. N. Amaral, Y. Gao, S. Cui, R. Silvera, and R. Archambault, "Mpads: Memory-pooling-assisted data splitting," in *Proceedings of the 7th international symposium on Memory management*, ACM, 2008, pp. 101–110. 97

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991. 73

[17] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh, "From OpenCL to high-performance hardware on FPGAs," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2012, pp. 531–534. 14, 19, 20

[18] L. Dagum and R. Menon, "OpenMP: An industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, Jan. 1998. 6

[19] N. Fauzia, L.-N. Pouchet, and P. Sadayappan, "Characterizing and enhancing global memory data coalescing on GPUs," in *Code Generation and Optimization (CGO)*, IEEE Computer Society, 2015, pp. 12–22. 10

[20] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987. 63

[21] J. Fifield, R. Keryell, H. Ratigner, H. Styles, and J. Wu, "Optimizing OpenCL applications on Xilinx FPGA," in *International Workshop on OpenCL*, Vienna, Austria: ACM, 2016, 5:1–5:2. 15, 19

[22] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007, pp. 407–420. 5

[23] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys (CSUR)*, vol. 23, no. 1, pp. 5–48, 1991. 84

[24] Google. (2011). Snappy: A fast compressor/decompressor, [Online]. Available: `https://github.com/google/snappy` (visited on 01/19/2018). 10, 96

[25] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *2012 Innovative Parallel Computing (InPar)*, May 2012. 16

[26] K. O. W. Group. (2011). ''the opencl specification: Version 1.0", october 2011., [Online]. Available: `https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf` (visited on 04/07/2017). 8

[27] I. Guyon, S. Gunn, A. Ben-Hur, and G. Dror, "Result analysis of the nips 2003 feature selection challenge," in *Advances in neural information processing systems*, 2005, pp. 545–552. 97

[28] S. Hauck and A. DeHon, *Reconfigurable computing: the theory and practice of FPGA-based computation*. Morgan Kaufmann, 2010, vol. 1. 20

[29] R. Hempel, "The mpi standard for message passing," in *International Conference on High-Performance Computing and Networking*, Springer, 1994, pp. 247–252. 96

[30] S. Horwitz, P. Pfeiffer, and T. Reps, "Dependence analysis for pointer variables," 7, ACM, vol. 24, 1989, pp. 28–40. 74

[31] (). Intel FPGA OpenCL Best Practices Guide, [Online]. Available: `http://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf` (visited on 04/04/2017). 15, 21, 23, 24, 28

[32] N. D. Jones, "Program flow analysis: Theory and applications," 1981. 69

[33] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. W. Hwu, H. Li, M. S. Müller, W. E. Nagel, M. Perminov, P. Shelepugin, K. Skadron, J. Stratton, A. Titov, K. Wang, M. van Waveren, B. Whitney, S. Wienke, R. Xu, and K. Kumaran, "Spec accel: A standard application suite for measuring hardware accelerator performance," in *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers*, S. A. Jarvis, S. A. Wright, and S. D. Hammond, Eds. Springer International Publishing, 2015, pp. 46–67. 36, 40

[34] Khronos. (). Clcreatebuffer, [Online]. Available: `https://www.khronos.org/registry/OpenCL/sdk/1.0/docs/man/xhtml/clCreateBuffer.html` (visited on 04/04/2017). 30

[35] S. T. Klein and Y. Wiseman, "Parallel lempel ziv coding," in *Annual Symposium on Combinatorial Pattern Matching*, Springer, 2001, pp. 18–30. 12

[36] T. Kremenek, "Finding software bugs with the Clang static analyzer," *California: Apple Inc*, 2008. 9

[37] C. A. Lattner, "Llvm: An infrastructure for multi-stage optimization," PhD thesis, University of Illinois at Urbana-Champaign, 2002. 58

[38] S. Lee, J. Kim, and J. S. Vetter, "OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 544–554. 16

[39] 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. 16

[40] R. Lenhardt and J. Alakuijala, "Gipfeli-high speed compression algorithm," in *Data Compression Conference (DCC), 2012*, IEEE, 2012, pp. 109–118. 10, 96

[41] M. Lichman, *UCI machine learning repository*, 2013. [Online]. Available: `http://archive.ics.uci.edu/ml`. 96, 97

[42] Z. Liu, Y. Saifullah, M. Greis, and S. Sreemanthula, "Http compression techniques," in *Wireless Communications and Networking Conference, 2005 IEEE*, IEEE, vol. 4, 2005, pp. 2495–2500. 12

[43] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *ACM SIGMICRO*, vol. 23, 1992, pp. 45–54. 64

[44] N. A. Naeem, O. Lhoták, and J. Rodriguez, "Practical extensions to the IFDS algorithm," in *International Conference on Compiler Construction (CC)*, 2010, pp. 124–144. 62

[45] B. Nicolae, "High throughput data-compression for cloud storage," in *International Conference on Data Management in Grid and P2P Systems*, Springer, 2010, pp. 1–12. 12

[46] J. Nielsen, *Usability Engineering*. Elsevier, 1994. 82

[47] NVidia, *Profiler user's guide*, `https://docs.nvidia.com/cuda/profiler-users-guide/`, Accessed: 2016-11-29. 60, 75

[48] Nvidia, *NVIDIA Tesla P100 – The Most Advanced Data Center Accelerator Ever Built.* `http://www.nvidia.ca/object/pascal-architecture-whitepaper.html`, Accessed: 2017-10-30. 38

[49] ——, *NVIDIA TESLA V100 GPU ARCHITECTURE – The World's Most Advanced Data Center GPU.* http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, Accessed: 2018-01-01. 54

[50] (). Oak ridge readies summit supercomputer for 2018 debut, [Online]. Available: https://www.top500.org/news/oak-ridge-readies-summit-supercomputer-for-2018-debut/ (visited on 01/01/2018). 1

[51] M. F. P. O'Boyle, Z. Wang, and D. Grewe, "Portable mapping of data parallel programs to opencl for heterogeneous systems," in *Code Generation and Optimization (CGO)*, IEEE, 2013. 8, 17

[52] A. Ozsoy, M. Swany, and A. Chauhan, "Pipelined parallel lzss for streaming data compression on gpgpus," in *Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, IEEE, 2012, pp. 37–44. 18

[53] R. A. Patel, Y. Zhang, J. Mak, A. Davidson, and J. D. Owens, *Parallel lossless data compression on the GPU.* IEEE, 2012. 18

[54] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ACM, 2012, pp. 377–388. 10, 84

[55] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker, "Register allocation for software pipelined loops," in *Programming Language Design and Implementation (PLDI)*, 1992, pp. 283–299. 23

[56] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Principles of programming languages (PoPL)*, 1995, pp. 49–61. 62

[57] D. Rolls, C. Joslin, and S.-B. Scholz, "Unibench: A tool for automated and collaborative benchmarking," in *International Conference on Program Comprehension (ICPC)*, IEEE, 2010, pp. 50–51. 36, 40

[58] D. Sampaio, R. M. d. Souza, S. Collange, and F. M. Q. Pereira, "Divergence analysis," in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 35, 2013, p. 13. 9

[59] (). SDAccel Environment User Guide, [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_3/ug1023-sdaccel-user-guide.pdf (visited on 04/27/2017). 15

[60] H. Sharangpani and H. Arora, "Itanium processor microarchitecture," *IEEE Micro*, no. 5, pp. 24–43, Sep. 2000. 23

[61] K. Shastry, A. Pandey, A. Agrawal, and R. Sarveswara, “Compression acceleration using gpgpu,” in *High Performance Computing Workshops (HiPCW), 2016 IEEE 23rd International Conference on*, IEEE, 2016, pp. 70–78. 12

[62] J. Shun and F. Zhao, “Practical parallel lempel-ziv factorization,” in *Data Compression Conference (DCC), 2013*, IEEE, 2013, pp. 123–132. 12

[63] E. Sitaridi, R. Mueller, T. Kaldewey, G. Lohman, and K. A. Ross, “Massively-parallel lossless data decompression,” in *Parallel Processing (ICPP), 2016 45th International Conference on*, IEEE, 2016, pp. 242–247. 12

[64] J. E. Stone, D. Gohara, and G. Shi, “OpenCL: A parallel programming standard for heterogeneous computing systems,” *Computing in Science & Engineering (CSE)*, vol. 12, no. 3, pp. 66–73, 2010. 19

[65] A. Stoutchinin and F. de Ferriere, “Efficient static single assignment form for predication,” in *ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2001, pp. 172–181. 64

[66] *Top500 supercomputers*, `https://www.top500.org/`, Accessed: 2018-01-01, 2017. 1

[67] G. Tournavitis, Z. Wang, B. Franke, and M. F. O’Boyle, “Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping,” in *Programming Language Design and Implementation (PLDI)*, Dublin, Ireland: ACM, 2009. 17

[68] M. Ujaldón, “Cuda achievements and gpu challenges ahead,” in *International Conference on Articulated Motion and Deformable Objects*, Springer, 2016, pp. 207–217. 92

[69] E. Velloso, A. Bulling, H. Gellersen, W. Ugulino, and H. Fuks, “Qualitative activity recognition of weight lifting exercises,” in *Proceedings of the 4th Augmented Human International Conference*, ACM, 2013, pp. 116–123. 97

[70] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout, “Non-affine extensions to polyhedral code generation,” in *International Symposium on Code Generation and Optimization (CGO)*, 2014, p. 185. 10

[71] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, “Polyhedral parallel code generation for CUDA,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, no. 4, p. 54, 2013. 10

[72] M. Vollmer, B. J. Svensson, E. Holk, and R. R. Newton, “Meta-programming and auto-tuning in the search for high performance gpu code,” in *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*, ser. FHPC 2015, Vancouver, BC, Canada: ACM, 2015. 16

[73] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: A machine learning based approach," in *Principles and Practice of Parallel Programming (PPoPP)*, Raleigh, NC, USA: ACM, 2009. 8, 17

[74] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, and X. Shen, "Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on GPU," in *Principles and practice of parallel programming (PPoPP)*, vol. 48, 2013, pp. 57–68. 9

[75] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "gpucc: An open-source GPGPU compiler," in *International Symposium on Code Generation and Optimization (CGO)*, 2016, pp. 105–116. 73

[76] P. Xiang, Y. Yang, and H. Zhou, "Warp-level divergence in GPUs: Characterization, impact, and mitigation," in *High Performance Computer Architecture (HPCA)*, IEEE, 2014, pp. 284–295. 50

[77] H. R. Zohouri, N. Maruyama, A. Smith, M. Matsuda, and S. Matsuoka, "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016, 35:1–35:12. 15, 19, 20, 23, 33