DIVERGÊNCIA EM GPU: ANÁLISES E ALOCAÇÃO DE REGISTRADORES

DIOGO N. SAMPAIO

DIVERGÊNCIA EM GPU: ANÁLISES E ALOCAÇÃO DE REGISTRADORES

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Fernando Magno Quintão Pereira

Belo Horizonte

Março de 2013

DIOGO N. SAMPAIO

GPU DIVERGENCE: ANALYSIS AND REGISTER ALLOCATTION

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão Pereira

Belo Horizonte

March 2013

Sampaio, Diogo N.
S192d GPU divergence: analysis and register allocattion / Diogo N. Sampaio. — Belo Horizonte, 2013
xviii, 73 f. : il. ; 29cm

Dissertação (mestrado) — Federal University of Minas Gerais
Orientador: Fernando Magno Quintão Pereira

1. Compilers. 2. GPGPU. 3. Static Analysis. 4. Divergence. 5. Register Allocation. 6. Rematerialization. I. Título.

CDU 519.6*33 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Divergência em GPU: análises e alocação de registradores (GPU divergence: analysis and register allocattion)

DIOGO NUNES SAMPAIO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador
Departamento de Ciência da Computação - UFMG

PROF. LUIZ FILIPE MENEZES VIEIRA
Departamento de Ciência da Computação - UFMG

PROF. RENATO ANTÔNIO CELSO FERREIRA
Departamento de Ciência da Computação - UFMG

PROF. RODOLFO JARDIM DE AZEVEDO
Instituto de Computação - UNICAMP

Belo Horizonte, 08 de março de 2013.

Resumo

Uma nova tendência no mercado de computadores é usar unidades de processamento gráfico (GPUs) [1] para acelerar tarefas paralelas por dados. Esse crescente interesse renovou a atenção dada ao modelo de execução Single Instruction Multiple Data (SIMD). Máquinas SIMD fornecem uma tremenda capacidade computacional aos desenvolvedores, mas programá-las de forma eficiente ainda é um desafio, particularmente devido a perdas de performance causadas por divergências de memória e de fluxo. Esses fenômenos são consequências de *dados divergentes*. *Dados divergentes* são variáveis com mesmo nome mas valores diferentes entre as unidades de processamento. A fim de lidar com os fenômenos de divergências, esta dissertação introduz uma nova ferramenta de análise de código, a qual chamamos *Análise de Divergência com Restrições Afins*. Desenvolvedores de programas e compiladores podem servir-se das informações de divergência com dois propósitos diferentes. Primeiro, podem melhorar a qualidade de programas gerados para máquinas que possuem instruções vetoriais, mesmo que essas sejam incapazes de lidar com divergências de fluxo. Segundo, podem otimizar programas criados para placas gráficas. Para exemplificar esse último, apresentamos uma otimização para alocadores de registradores que, usando das informações geradas pelas análises de divergências, melhora a utilização da hierarquia de memória das placas gráficas. Testados sobre conhecidos benchmarks, os alocadores de registradores otimizados produzem código que é, em média, 29.70% mais rápido do que o código gerado por alocadores de registradores convencionais.

Palavras-chave: Linguagem de Programação, Compiladores, GPGPU, Analise Estática, Divergências, Alocação de Registradores, Rematerialização, SIMT, SIMD.

[1] Do inglês Graphics Processing Units

Abstract

The use of *graphics processing units* (GPUs) for accelerating *Data Parallel* workloads is the new trend on the computing market. This growing interest brought renewed attention to the Single Instruction Multiple Data (SIMD) execution model. SIMD machines give application developers tremendous computational power; however, programming them is stil challenging. In particular, developers must deal with memory and control flow divergences. These phenomena stem from a condition that we call *data divergence*, which occurs whenever *processing elements* (PEs) that run in lockstep see the same variable name holding different values. To deal with *divergences* this work introduces a new code analysis, called *Divergence Analysis with Affine Constraints*. Application developers and compilers can benefit from the information generated by this analysis with two different objectives. First, to improve code generate to machines that have vector instructions but cannot handle control divergence. Second, to optimize GPU code. To illustrate the last one, we present register allocators that relly on divergence information to better use GPU memory hierarchy. These optimized allocators produced GPU code that is 29.70% faster than the code produced by a conventional allocator when tested on a suite of well-known benchmarks.

Keywords: Programming Languages, Compilers, GPGPU, Static Analysis, Divergence, Register Allocation, Rematerialization, SIMT, SIMD.

List of Figures

List of Tables

Contents

Glossary

GPU	**Graphics Processing Unit**: Is a hardware accelerator. The first GPUs were built with specific purpose to accelerate the process of representing lines and arcs in bitmaps, to be displayed on the monitor. With time, GPUs incorporated hardware implementations of functions used to rasterize three dimensional scenarios. Later on, this accelerator turned to be massively parallel accelerators, capable of generic processing.
CUDA	**Compute Unified Device Architecture**: Is a parallel computing platform and programming model created by NVIDIA and implemented to the graphics processing units (GPUs) that they produce.
PTX	**Parallel Thread Execution**: Is a pseudo-assembly language used in Nvidia's CUDA programming environment. The nvcc compiler translates code written in CUDA, a C-like language, into PTX, and the graphics driver contains a compiler which translates the PTX into a binary code which can be run on the processing cores.
Code hoisting	Compiler optimizations techniques that reduce amount of instructions by moving duplicated instructions on different execution paths to a single instructions on a common path.
kernel	Function that executes in the GPU.
thread block	Is a set of *threads*, defined by the programmer, at a *kernel* call. A *thread block* executes in one *SM*, sharing its resources, such as cache memory and register file, equally among all *threads* of the block.
SM	**Stream Multiprocessor**: A *SIMD* processor. It executes *threads* of a *thread block* in *warps*.
warp	A group of GPU *threads* that execute on **lock-step**.
threads	Execution lines in the same *kernel*. GPU threads share global and shared memory, and have different mappings for the register file and local memory.

Acknowledgement

Many thanks to Sylvain Collange, Bruno Coutinho and Gregory Diamos for helping in the implementation of this work, to my adviser, Fernando Pereira, for being a source of inspiration and knowledge, and to my family, for the unconditional support and so much more that I can't describe in words.

Chapter 1

Introduction

A new trend in the computer market is to use GPUs as accelerators for general purpose applications. This raising popularity stems from the fact that GPUs are massively parallel. However GPUs impose new programming challenges onto application developers. These challenges are due, mostly to control and memory divergences. We present an new compiler analysis that identify divergence hazards and can help developers and automatic optimizations to generate faster GPU code. Our analysis can also assist the translation of GPU designed applications to vector instructions on conventional CPUs, that cannot handle divergence hazards.

1.1 How important GPUs are becoming

Increasing programmability and low hardware cost are boosting the use of GPUs as a tool to accelerate general purpose applications. This trend can be especially observed on the academic environment, as demonstrated in Google Trends[1] and TOP 500 supercomputers. The chart 1.1, taken from Google Trends, illustrates the rising popularity of CUDA[2] (see NVIDIA [2012]) and OpenCL[3] (see Khronos [2011]) languages among the Computer Science community. *TOP 500 supercomputers*, an organization that often ranks the 500 most powerful supercomputers on the world, demonstrates that by November 2012 (see TOP500 [2012]), 53 of these supercomputers use GPUs accelerators, where three of them are among the five most power efficient (Mflops / watt) and

[1] http://www.google.com/trends/

[2] Compute Unified Device Architecture: a parallel computing platform and programming model created by NVIDIA

[3] Open Computing Language: Open standard framework for writing programs that execute across heterogeneous platforms

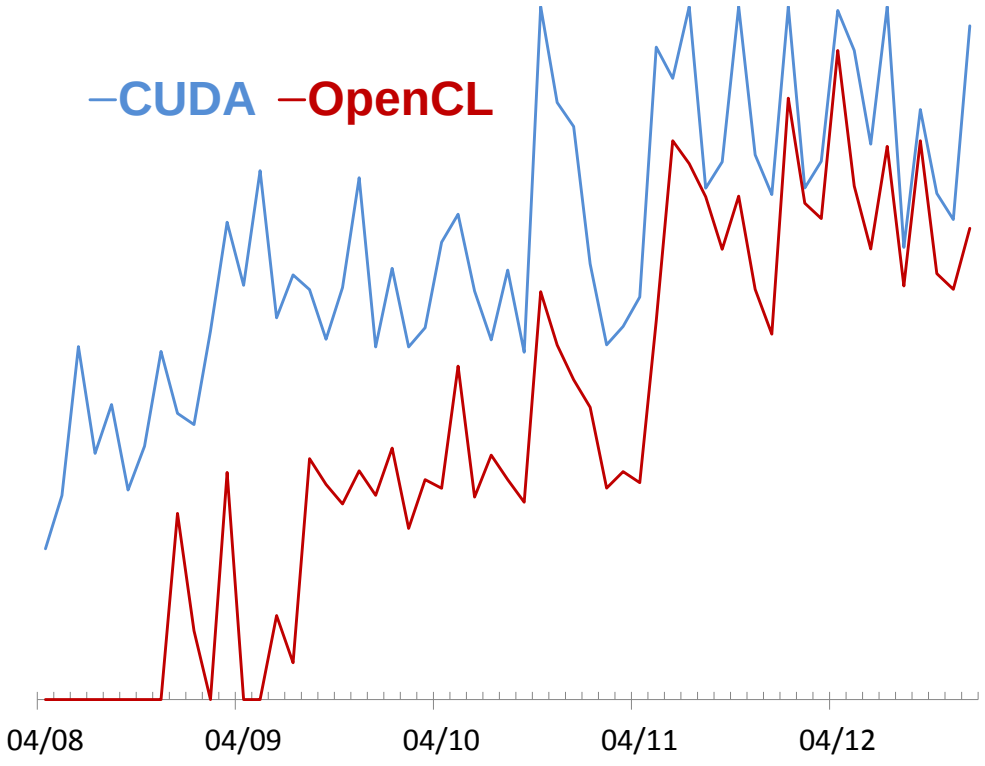


Figure 1.1. Google Trends chart: CUDA and OpenCL popularity over time

two among the ten most powerful.

This trend will continue, as academia and industry work on improving GPUs hardware and software. Efficient GPU algorithms were presented to solve problems as diverse as sorting (Cederman and Tsigas [2010]), gene sequencing (Sandes and de Melo [2010]), IP routing (Mu et al. [2010]) and program analysis (Prabhu et al. [2011]). Performance tests exposed by Ryoo et al. [2008] demonstrate that some programs can run over 100 times faster on GPUs than their CPU equivalent. Upcoming hardware will closely integrate GPUs and CPUs, as demonstrated by Boudier and Sellers [2011] and new models of heterogeneous hardware are being introduced (Lee et al. [2011]; Saha et al. [2009]).

GPUs are attractive because these processors are massively parallel and high throughput oriented. To illustrate GPUs characteristics, we are going to use NVIDIA's GeForce GTX-580 GPU series, a domestic high-end GPU. A GTX-580 has 512 *processing elements* (PEs) that can be simultaneously used by up to 24,576 threads. It delivers, approximately, up to 1500 Gflops[4] in single precision, much higher than the 62 Gflops

[4]Gflop: 10^9 floating-point operations per second

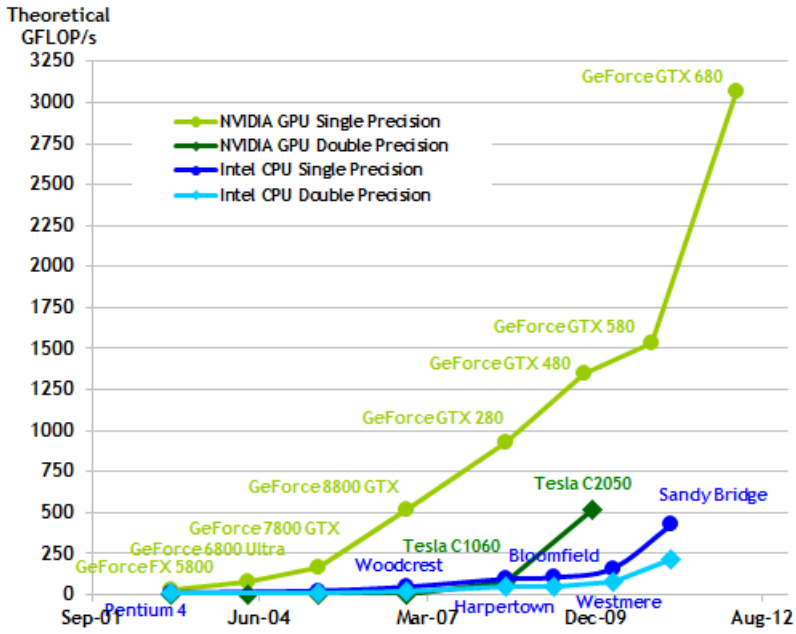


Figure 1.2. Chart taken from Nvidia's *CUDA C Programming Guide* that shows GPU and CPU peak performance over time

of a CPU at the same price range and release date. The chart 1.2 compares GPU to CPU theoretical computational power over time and 1.3 illustrates how domestic GPUs can outperform server CPUs when solving the same problem using FEKO[5] , a parallel library for electromagnetic simulation.

Compared to a regular CPU, a GPU chip devotes much more transistors to PEs and much fewer transistors to cache memory and flow control, as illustrated by figure 1.4. To reduce dramatically their control hardware, GPUs adopt a more restrictive programming model, and not every parallel application can benefit from this parallelism. These processors organize threads in groups that execute in lock-step, called *warps*.

1.2 Divergence

To understand the rules that govern threads in a *warp*, we can imagine that each *warp* has simultaneous access to many PEs, but uses only one instruction fetcher. As an example, the GTX-580 has 16 independent cores, called *Streaming Multiprocessors* (SM). Each SM can run 48 *warps* of 32 threads each, thus, each *warp* might execute 32 instances of the same instruction simultaneously. Regular applications, such as *Data parallel algorithms* studied by Hillis and Steele [1986], fare very well in GPUs, as we

[5]See http://www.feko.info/ for more details

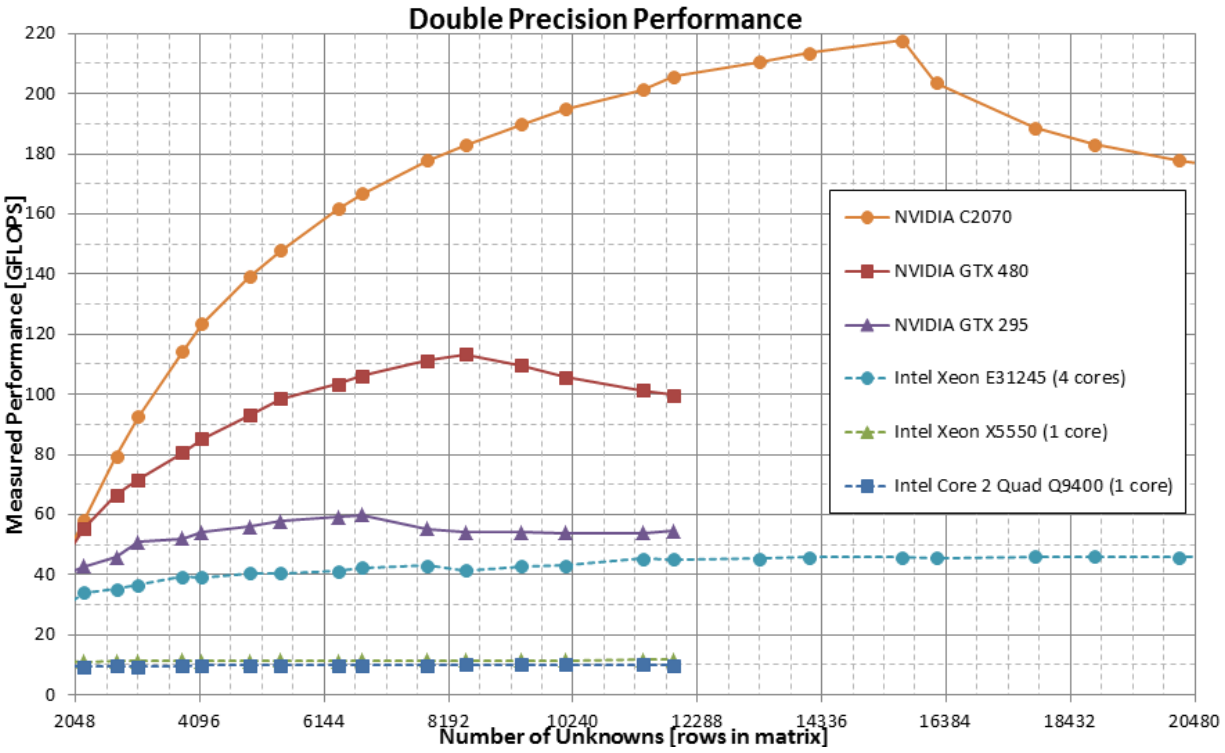


Figure 1.3. GPU and CPU performance on FEKO

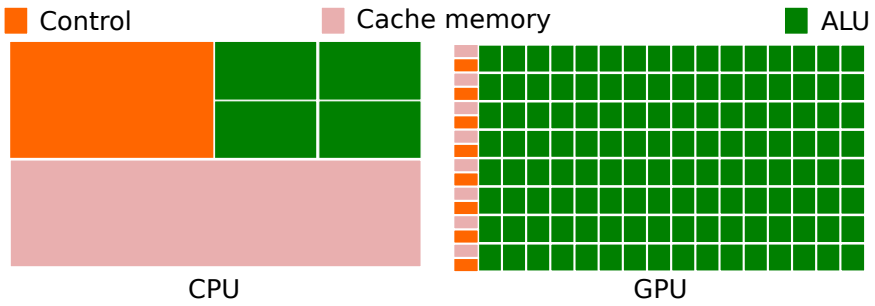


Figure 1.4. GPU and CPU design comparison

have the same operation being independently performed on different chunks of data. However, divergence may happen in less regular applications.

Definition 1 (Data divergence)**.** Data divergence *happens whenever the same variable name is mapped to different values by different threads. In this case we say that the variable is* divergent*, otherwise we call it* uniform*.*

As we will see in chapter 2, a *thread identifier* ($\mathtt{T}_{id}$) for instance, a special variable that gives each thread a unique number, is inherently divergent.

1.3 The problem of divergence

Data divergences are responsible for two phenomena that can compromise performance: memory and control flow divergence. *Control flow divergence* happens when threads in a *warp* follow different paths after processing the same branch instruction. If the branching condition is *data divergent*, then it might be true to some threads, and false to others. The code from both paths, following the branch instruction, is serialized. Given that each *warp* has access to only one instruction at a time some threads have to wait idly, while others execute. As pointed out by Zhang et al. [2011], in a worse case scenario, *control flow divergence* can reduce hardware utilization down to $3,125\%$ on a GTX-580, when only one among all 32 threads of a *warp* is active.

Memory divergence, a term coined by Meng et al. [2010], happens whenever a load or store instruction targeting data divergent addresses causes threads in a *warp* to access memory positions with bad locality. Usually each thread request a different memory position upon a input data access. Such event request that a huge amount of data to travel between memory and the GPU. To deal with these events GPUs rely on a very large memory communication bandwidth, being able to transfer a huge amount of data at once. However, the data requested need to be congruent to be transferred at once. If the memory access requested by a single instruction is apart by more than the memory communication bandwidth, more than one memory access is required to deliver the requested data of a single instruction. If each thread access a data that is very distant apart from each other, a single instruction could generate many memory transfers. Such events have been shown by Lashgar and Baniasadi [2011] to have even more performance impact than *control flow divergence*.

Optimizing an application to avoid divergence is problematic for two reasons. First, some parallel algorithms are intrinsically divergent; thus, threads will naturally disagree on the outcome of branches. Second, identifying divergences burdens the application developer with a tedious task, which requires a deep understanding of code that might be large and complex.

1.4 Our contribution

Our main goal is to provide compilers with techniques that allow them to understand and to improve divergent code. To meet such objective in Section 3.4 we present a static program analysis that identifies *data divergence*. We then expand this analysis, discussing, in Section 3.5, a more advanced algorithm that distinguishes *divergent* and

affine variables, e.g. variables that are affine expressions of *thread identifiers*. The two analyses discussed here rely on the classic notion of *Gated Static Single Assignment form* (see Ottenstein et al. [1990]; Tu and Padua [1995]), which we revise in Section 3.3. We formalize our algorithms by proving their correctness with regard to μ-SIMD, a core language that we describe in Section 3.2.

The divergence analysis is important in different ways. Firstly, it helps the compiler to optimize the translation of *Single Instruction Multiple Threads* (*SIMT*[6], see Patterson and Hennessy [2012]; Garland and Kirk [2010]; Nickolls and Dally [2010]; Habermaier and Knapp [2012]) languages to ordinary CPUs. We call *SIMT* languages those programming languages, such as C for CUDA and OpenCL, that are equipped with abstractions to handle divergence. Currently there exist many proposals (Diamos et al. [2010]; Karrenberg and Hack [2011]; Stratton et al. [2010]) to compile such languages to ordinary CPUs and they all face similar difficulties. Vector operations found in traditional architectures, such as the x86's SSE extension, do not support *control flow divergence* natively. Different than GPUs, where each data is processed by a different thread, that can be deactivated upon a *control flow divergence*, these instructions are executed by a single thread, that do the same operation on all data of a vector. If different data on a single vector need to be submitted to different instructions, these vector need to be reorganized on vector that all elements need to be submitted to the same treatment. Such problems have been efficiently addressed by Kerr et al. [2012], but still requires extra computation at *thread frontier* to recalculate active threads. This burden can be safely removed from the uniform, e.g., non-divergent, branches that we identify. Furthermore, the divergence analysis provides insights about memory access patterns, as demonstrated by Jang et al. [2010]. In particular, a uniform address means that all threads access the same location in memory, whereas an affine address means that consecutive threads access adjacent or regularly-spaced memory locations. This information is critical to generate efficient code for vector instruction sets that do not support fast memory gather and scatter (see Diamos et al. [2010]).

Secondly, in order to more precisely identify divergence, a common strategy is to use instrumentation based profilers. Coutinho et al. [2010] has done so, however, this approach may slowdown the target program by factors of over 1500 times! Our divergence analysis reduces the amount of branches that the profiler must instrument; hence, decreasing its overhead.

Thirdly, the divergence analysis improves the static performance prediction techniques

[6]This term was created by NVIDIA to describe their GPU execution model

used in SIMT architectures (see Baghsorkhi et al. [2010]; Zhang and Owens [2011]). For instance, Samadi et al. [2012] used adaptive compilers that target GPUs.

Finally, our analysis also helps the compiler to produce more efficient code to SIMD hardware. There exists a recent number of *divergence aware* code optimizations, such as Coutinho et al. [2011] *branch fusion*, and Zhang et al. [2011] *thread reallocation* strategy. We augment this family of techniques with a *divergence aware register allocator*. As shown in Chapter 4, we use divergence information to decide the best location of variables that have been spilled during register allocation. Our affine analysis is specially useful to this end, because it enables us to perform *Rematerialization* (see Briggs et al. [1992]) of values among SIMD processing elements. *Rematerialization* is a compiler optimization which saves time by recomputing a value instead of loading it from memory. It is typically tightly integrated with register allocation, where it is used as an alternative to spilling registers to memory.

All the algorithms that we describe are publicly available in the Ocelot compiler[7] (Diamos et al. [2010]). Our implementation in the Ocelot compiler consists of over 10,000 lines of open source code. Ocelot optimizes PTX code, the intermediate program representation used by NVIDIA's GPUs. We have compiled all the 177 CUDA kernels[8] from 46 applications taken from the Rodinia (Che et al. [2009]) and the NVIDIA SDK benchmarks. The experimental results given in Section 5 show that our implementation of the divergence analysis runs in linear time on the number of variables in the source program. The basic divergence analysis proves that about one half of the program variables, on average, are uniform. The affine constraints from Section 3.5 increase this number by 4%, and – more important – they indicate that about one fourth of the divergent variables are affine functions of some *thread identifier*. Finally, our *divergence aware register allocator* is effective: by rematerializing affine values, or moving uniform values to the GPU's shared memory, we have been able to speedup the code produced by Ocelot's original allocator by almost 30%.

As a result of this research the following papers were published:

- **Spill Code Placement for SIMD Machines**
  Best paper award
  Sampaio, D. N., Gedeon, E., Pereira, F. M. Q., Collange, S.
  16th Brazilian Symposium, SBLP 2012, Natal, Brazil, September 23-28, 2012 Programming Languages, pp 12-26, Springer Berlin Heidelberg.

[7] Available at http://code.google.com/p/gpuocelot/
[8] A kernel is a special program function because it executes on the GPU

- **Divergence Analysis with Affine Constraints**
  Sampaio, D. N., Martins, R., Collange, S., Pereira, F. M. Q.
  24th International Symposium on Computer Architecture and High Performance Computing, New York City, USA
  SBAC-PAD 2012, pp 67-74.

Chapter 2

Related work

Although General Purpose GPU (GPGPU) programming is a recent technology, it is rapidly becoming popular, especially on the scientific community. A substantial body of work has been published about program optimizations that target GPUs specifically. The objective of this chapter is to discuss the literature that has been produced about compiler optimizations for GPUs. Therefore we further detail divergences via an example (Section 2.1), give notations on divergence optimizations (Section 2.2) and compare our divergence analysis against previous work.

2.1 Divergence

GPUs run in the so called SIMT execution model. The SIMT model combines MIMD, SPMD and SIMD semantics, but divergence is relevant only at the SIMD level. This semantics combination works as follows:

- Each GPU might have one or more cores (*SM*), that follow Flynn's *Multiple Instruction Multiple Data* (MIMD) model (see Flynn [1972]), that is, each *SM* can be assigned to execute code from a different kernel and *SMs* executing code from the same *kernel* are not synchronized.

- Threads from the same *kernel* are grouped in *Thread Blocks*. Each *Thread Block* is assigned to a *SM*. Each *Thread Block* is divided in fixed sets of threads called *warps*. Each *warp* in a *Thread Block* follows Darema's *Single Program Multiple Data* (SPMD) execution model (Darema et al. [1988]), that is, within a *Thread Block* all *warps* execute the same *kernel* but are scaled asynchronously.

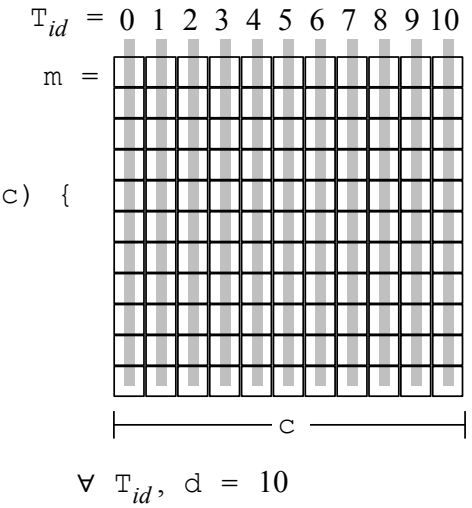
- Threads inside the same *warp* execute in **lock-step**, fitting Flynn's SIMD machines, that is, all threads of a *warp* execute simultaneously the same instruction.

We will use two artificial programs in Figure 2.1 to explain the notion of *divergence*. These functions, normally called *kernels*, are written in C for CUDA and run on graphics processing units. We will assume that these programs are executed by a number of threads, or PEs, according to the SIMD semantics. All the PEs see the same set of variable names; however, each one maps this environment onto a different address space. Furthermore, each PE has a particular set of identifiers. In C for CUDA this set includes the index of the thread in three different dimensions, e.g., `threadIdx.x`, `threadIdx.y` and `threadIdx.z`. We will denote this unique thread identifier by $\mathtt{T}_{id}$.

Each thread uses its $\mathtt{T}_{id}$ to find the data that it must process. Thus, in the *kernel* `avgSquare` each thread $\mathtt{T}_{id}$ is in charge of adding the elements of the $\mathtt{T}_{id}$-th column of `m`. Once leaving the loop, this thread will store the column average in $\mathtt{v[T}_{id}\mathtt{]}$. This is a divergent memory access: each thread will access a different memory address. However, modern GPUs can perform these accesses very efficiently, because they have good locality. In this example, addresses used by successive threads are contiguous (Ryoo et al. [2008]; Yang et al. [2010]). Control flow divergence will not happen in `avgSquare`. That is, each thread will loop the same number of times. Consequently, upon leaving the loop, every thread sees the same value at its image of variable `d`. Thus, we call this variable uniform.

Kernel `sumTriangle` presents a very different behavior. This rather contrived function sums up the columns in the superior triangle of matrix `m`; however, only the odd lines of a column contribute to the sum. In this case, the threads perform different amounts of work: the threads that has $\mathtt{T}_{id} = n$ will visit $n+1$ cells of `m`. After a thread leaves the loop, it must wait for the others. Processing resumes once all of them synchronize at line 12. At this point, each thread sees a different value stored at its image of variable `d`, which has been incremented $\mathtt{T}_{id} + 1$ times. Hence, we say that `d` is a *divergent* variable outside the loop. Inside the loop, `d` is *uniform*, because every active thread sees the same value stored at that location. Thus, all the threads active inside the loop take the same path at the branch in line 7. Therefore, a precise divergence analysis must split the live range of `d` into a *divergent* and a *uniform* part.

```
1 __global__ void avgSquare(float* m, float* v, int c) {
2   if (T_id < c) {
3     int d = 0;
4     float sum = 0.0F;
5     int N = T_id + c * c;
6     for (int i = T_id; i < N; i += c) {
7         sum += m[i];
8         d += 1;
9     }
10    v[tid] = sum / d;
11  }
12}
```




```
1 __global__ void sumTriangle(float* m, float* v, int c) {
2   if (T_id < c) {
3     int d = 0;
4     float sum = 0.0F;
5     int L = (T_id + 1) * c;
6     for (int i = T_id; i < L; i += c) {
7       if (d % 2) {
8         sum += m[i];
9       }
10      d += 1;
11    }
12    v[d-1] = sum;
13  }
14}
```

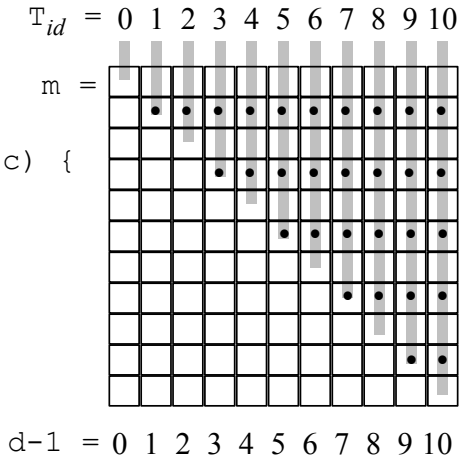


Figure 2.1. The gray lines on the right show the parts of matrix `m` processed by each thread. Following usual coding practices we represent the matrix in a linear format. Dots mark the cells that add up to the sum in line 8 of `sumTriangle`.

2.2 Divergence Optimizations

We call divergence optimizations the code transformation techniques that use the results of divergence analysis to generate better programs. Some of these optimizations deal with memory divergences; however, methods dealing exclusively with control flow divergence are the most common in the literature. As an example, the PTX programming manual[1] recommends replacing ordinary branch instructions (`bra`) proved to be non-divergent by special instructions (`bra.uni`), which are supposed to divert control to the same place for every active thread. Other examples of control flow divergence optimizations include *branch distribution*, *branch fusion*, *branch splitting*, *loop collapsing*, *iteration delaying* and *thread reallocation*. At run-time divergence control flow can be reduced with a more complex hardware as demonstrated in *large warps and two-level warp scheduling* and *SIMD re-convergence at thread frontiers*.

2.2.1 Optimizing divergent control flow

A common method to optimize divergent control flow is to minimize the amount of work done in divergent paths, such as *branch distribution* and *branch fusion*.

Branch distribution (Han and Abdelrahman [2011]) is a form of *code hoisting*[2] that works both at the Prolog and at the epilogue of a branch. This optimization merges code from potentially divergent paths to outside the branch. *Branch fusion* (Coutinho et al. [2011]), a generalization of *Branch distribution*, joins chains of common instructions present in two divergent paths if the additional execution costs of regenerating the same divergence is lower than the potential gain.

A number of compiler optimizations try to rearrange loops in order to mitigate the impact of divergence. Carrillo et al. [2009] proposed *branch splitting*, a way to divide a parallelizable loop enclosing a multi-path branch into multiple loops, each containing only one branch. Lee et al. [2009] designed *loop collapsing*, a compiler technique that reduces divergence inside loops when compiling OpenMP programs into C for CUDA. Han and Abdelrahman [2011] generalized Lee's approach proposing *iteration delaying*, a method that regroups loop iterations, executing those that take the same branch direction together.

Thread reallocation is a technique that applies on settings that combine the SIMD and

[1] PTX programming manual, 2008-10-17, SP-03483-001_v1.3, ISA 1.3

[2] techniques known as code hoisting tend to diminish the number of instructions of a program by unifying duplicated instructions on different control flow paths into a unique instruction in a common path

the SPMD semantics, like the modern GPUs. This optimization consists in regrouping divergent threads among *warps*, so that only one or just a few *warps* will contain divergent threads. Their main idea is that divergence occur based on input values, so they sort the input vectors applying the same modification over the threadIdx. It has been implemented at the software level by Zhang et al. [2010, 2011], and simulated at the hardware level by Fung et al. [2007]. However, this optimization must be used with moderation, as shown by Lashgar and Baniasadi [2011], unrestrained thread regrouping could lead to memory divergence.

At the hardware level, Narasiman et al. [2011] propose in *Improving GPU performance via large warps and two-level warp scheduling warps* with number of threads multiple of number of PEs in a Stream Multiprocessor by a factor n. A inner *warp* scheduler is capable of selecting for each PE, one among n threads, always preferring an active. Such mechanism can reduce the number of inactive threads selected for execution, but it also can lead to memory divergence.

Most GPUs only converge execution of divergent threads at the post-dominator block of divergent branches. Diamos et al. [2011] in *SIMD re-convergence at thread frontiers* minimize the amount of time that threads remain inactive by using a hardware capable of identifying blocks before the post dominator, called *thread frontiers*, where some threads of a *warp* might resume execution. The active threads set is recalculated at these points comparing the current PC against the PC of inactive threads.

2.2.2 Optimizing memory accesses

The compiler related literature describes optimizations that try to change memory access patterns in such a way to improve address locality. Recently, some of these techniques were adapted to mitigate the impact of memory divergence in modern GPUs.

Yang et al. [2010] and Pharr and Mark [2012] describe a suite of loop transformations to *coalesce* data accesses. Memory coalescing consists in the dynamic aggregation of contiguous locations into a single data access. Leissa et al. [2012] discuss several data layouts that improve memory locality in the SIMD execution model.

Rogers et al. [2012] propose in *Cache-Conscious Wavefront Scheduling* a novel *warp* scheduler that uses a scoreboard based on the *warp* memory access pattern and data stored in cache, being able to maximize L1 data cache reuse among *warps* of a *thread block*, archiving a 24% performance gain over the previous state of the art *warp* scheduler.

2.2.3 Reducing redundant work

The literature describes a few optimizations that use data divergence information to reduce the amount of work that the SIMD processing elements do. For instance, Collange et al. [2010] introduces *work unification*. This compiler technique leaves to only one thread the task of computing uniform values; hence, reducing memory accesses and hardware occupancy. Some computer architectures, such as Intel MIC[3] and AMD GCN[4], combine scalar and SIMD processing units. Although we are not aware of any work that tries to assign computations to the different units based on divergence information, we speculate that this is another natural application of the analysis that we present.

2.3 Divergence Analyses

Several algorithms have been proposed to find uniform variables. The first technique that we are aware of is the *barrier inference* of Aiken and Gay [1998]. This method, designed for SPMD machines, finds a conservative set of uniform[5] variables via static analysis. However, because it is tied to the SPMD model, Aiken and Gay's algorithm can only report uniform variables at global synchronization points.

The recent interest on graphics processing units has given a renewed impulse to this type of analysis, in particular with a focus on SIMD machines. For instance, Stratton et al. [2010] *variance analysis*, and Karrenberg and Hack [2011] *vectorization analysis* distinguish *uniform* and *divergent* variables. However, these analyses are tools used in the compilation of SPMD programs to CPUs with explicit SIMD instructions, and both compilers generate specific instructions to manage divergence *at run-time*. On the other hand, we focus on architectures in which divergence are managed implicitly by the hardware. A naive application of Stratton's and Karrenberg's approach in our static context may yield false negatives due to control dependency, although they work well in the dynamic scenario for which they were designed. For instance, Karrenberg's select and loop-blending functions are similar to the γ and η functions that we discuss in Section 3.3. However, select and blend are concrete instructions emitted during code generation, whereas our GSA functions are abstractions used statically.

Coutinho et al. [2011] proposed a divergence analysis that works in programs in the Gated Static Single Assignment (GSA) format (see Ottenstein et al. [1990]; Tu and

[3]See *Many Integrated Core Architecture* at `http://www.intel.com`. Last visit: Jan. 13
[4]See *Understanding AMD's Roadmap* at `http://www.anandtech.com/`. Last visit: Jan. 13
[5]Aiken and Gay would call these variables *single-valued*

Padua [1995]). Thus, Coutinho *et al.*'s analysis is precise enough to find out that variable d is uniform inside the loop in the kernel `sumTriangle`, and divergent outside. Nevertheless, their version is over-conservative because it does not consider affine relations between variables. For instance, in the kernel `avgSquare`, variables i and L are functions of $\mathtt{T}_{id}$. However, if we inspect the execution trace of this program, then we will notice that the comparison $i < L$ has always the same value for every thread. This fact happens because both variables are functions of two affine expressions of $\mathtt{T}_{id}$, whose combination cancel the $\mathtt{T}_{id}$ factor out, e.g.: $L = \mathtt{T}_{id} + c_1$ and $i = \mathtt{T}_{id} + c_2$; thus, $L - i = (1-1)\mathtt{T}_{id} \ + \ (c_1 - c_2)$. Therefore, a more precise divergence analysis requires some way to take affine relations between variables into consideration.

Figure 2.2 summarizes this discussion comparing the results produced by these different variations of the divergence analysis when applied on the kernels in Figure 2.1. We call *Data Dep.* a divergence analysis that takes data dependency into consideration, but not control dependency. In this case, a variable is uniform if it is initialized with constants or broadcast values, or, recursively, if it is a function of only uniform variables. This analysis would, incorrectly, flag variable `d` in `avgSquare`, as uniform. Notice that, because this analysis use the GSA intermediate representation, they distinguish the live ranges of variable d inside (d_{in}) and outside (d_{out}) the loops. The analysis that we present in Section 3.5 improves on the analysis that we discuss in Section 3.4 because it considers affine relations between variables. Thus, it can report that the loop in `avgSquare` is non-divergent, by noticing that the comparison $i < N$ has always the same value for every thread. This fact happens because both variables are functions of two affine expressions of $\mathtt{T}_{id}$, whose combination cancel the $\mathtt{T}_{id}$ factor out, e.g.: $N = \mathtt{T}_{id} + c_1$ and $i = \mathtt{T}_{id} + c_2$; thus, $N - i = (1-1)\mathtt{T}_{id} \ + \ (c_1 - c_2)$.

2.3.1 Chapter conclusion

This chapter provided some context on which the reader can situate himself. It defined *divergence data*, and showed this kind of data can degrade the performance of a GPU, via flow and memory divergences. It described possible optimizations that can benefit from the information of divergence analysis as motivation for our new technique, comparing its results against previously proposed techniques. As we will explain in the rest of this dissertation, our divergence analysis is more powerful than the previous approaches, because it relies on a more complex lattice. The immediate result is that we are able to use it to build a register allocator that would not be possible using only the results of the other techniques that have been designed before ours. In the next chapter we formalize our notion of divergence analysis with affine constraints.

	Data Dep.	Aiken	Karr.	Sec. 3.4	Sec. 3.5
c	U	U	U	U	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
m	U	U	U	U	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
v	U	U	U	U	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
i	D	D	ca	D	$0\mathtt{T}_{id}^2 + \mathtt{T}_{id} + \bot$
avgSquare					
N	D	D	c	D	$0\mathtt{T}_{id}^2 + \mathtt{T}_{id} + \bot$
$\mathtt{d}_{in}$	U	D	D	D	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
$\mathtt{d}_{out}$	U	D	D	D	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
sumTriangle					
L	D	D	D	D	$0\mathtt{T}_{id}^2 + \bot\mathtt{T}_{id} + \bot$
$\mathtt{d}_{in}$	U	D	D	U	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
$\mathtt{d}_{out}$	U	D	D	D	$\bot\mathtt{T}_{id}^2 + \bot\mathtt{T}_{id} + \bot$

Figure 2.2. We use U for uniform and D for Divergent variables. Karrenberg's analysis can mark variables in the format $1 \times \mathtt{T}_{id} + c, c \in \mathbb{N}$ as consecutive (c) or consecutive aligned (ca)

Chapter 3

Divergence Analyses

In this chapter we describe two divergence analyses. We start by giving in Section 3.1 an informal overview on the basics of divergence analysis. We describe how our technique of using affinity with the $\mathtt{T}_{id}$ augments the precision compared to a simple implementation. In Section 3.4 we formalize a simple and fast implementation. This initial analysis helps us to formalize the second algorithm, presented in Section 3.5, a slower, yet more precise technique because it can track affine relations between program variables, an extra information vital to our novel register allocator. Our affine divergence aware register allocator uses the GPU's memory hierarchy in a better way when compared to previous register allocators. This better usage comes out of the fact that our allocator decreases the amount of memory used when affine variables are selected for spilling. It does so by trading memory space by extra computations, which we perform as value rematerialization. We formalize our divergence analysis by describing how it operates on a toy SIMD model, which we explain in Section 3.2. Our analysis requires the compiler to convert the target program to a format called *Gated Static Single Assignment form*. We describe this format in Section 3.3.

3.1 Overview

Divergence Analyses, in the GPU context, are compiler static analyses that classify variables accordingly to, either all *threads* will see it with the same value or not. As in the Definition 1, the conservative set of variables classified as *uniform variables* will hold variables that, at compile time, are known to contain the same value among all *threads* at run-time. The set of *divergent variables* hold the variables that might contain different values among *threads* at run-time.

To be able to classify variables as *divergent* and *uniform*, *divergence analyses* rely on:

1. *Data divergence* origins are known at compile time, and are pre-classified as *divergent variables*. Divergence origins are:

   a) **Threads identifiers ($\mathtt{T}_{id}$)**: It is known to be a unique value per *thread*.

   b) **Atomic operations results**: Each *thread* might receive a different value after an atomic operation.

   c) **Local Memory Loads**: Local memory has a unique mapping per *thread*, so values loaded from it might be different among threads.

2. *Divergent variables* can only affect variables that dependent on them. There are two types of dependency among variables:

   a) Data dependency.

   b) Control dependency.

Variables dependency are represented in a oriented graph. If a variable v is dependent on the variable $\mathtt{T}_{id}$, then there is an arrow from $\mathtt{T}_{id}$ to v. **Data dependency** is easily extracted from the instructions of the program. To every instruction, such as $a = b + c$, the left side (a) is dependent on the right side (b, c). **Control dependency** is a little more tricky to be detected, and require a special program representation to be identified, as described in details in Section 3.3. But a simple definition is, if variable p controls which value is assigned to a variable v, then v is control dependent on p. The fluxogram in Figure 3.1 illustrates data and control dependency.

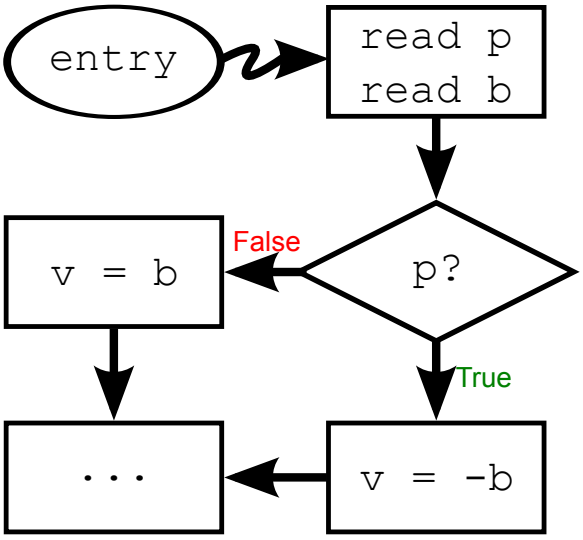


Figure 3.1. Variable `v` is **data dependent** of variable `b` and **control dependent** of variable `p`.

The simple divergence analysis builds the *variable dependency* graph and uses a *graph reachability* algorithm to detect all variables dependent on divergent variables, as illustrated in Figure 3.2.

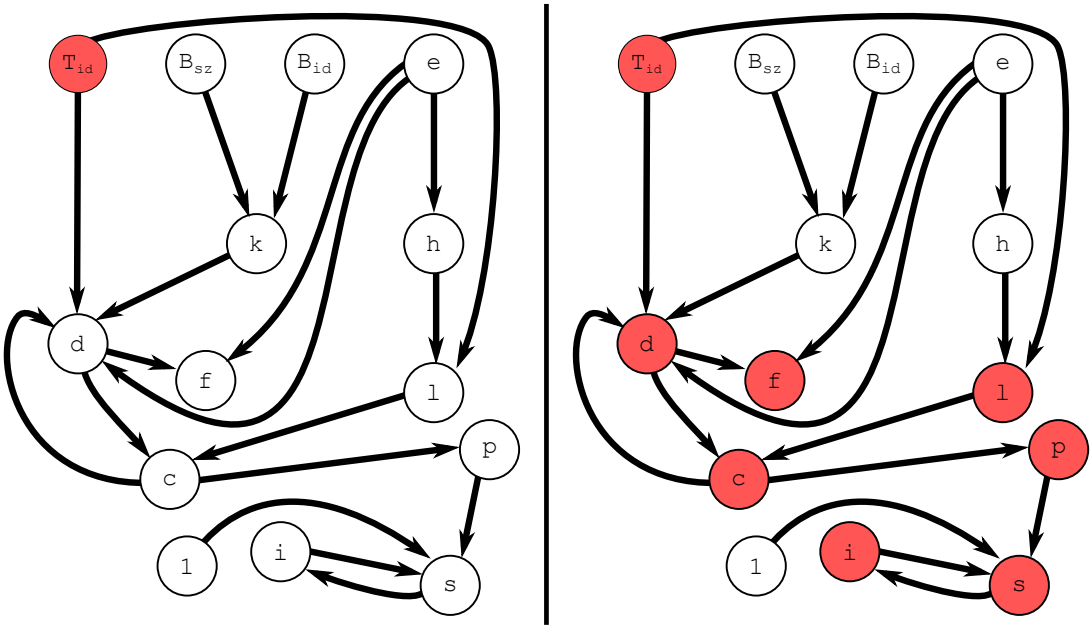


Figure 3.2. On the left the initial state of the variable dependency graph. On the right, variables states after *graph reachability* applied from *divergence sources*

The divergence analysis with affine constraints is capable do give more details about the nature of the values that are going to be store in the variables. It uses the technique to keep variables affinity of degree one to the $\mathtt{T}_{id}$. That is, all variables have an abstract state, compound by two elements (a, b) that describe the encountered value as $a \times \mathtt{T}_{id} + b$. Each of this elements can be assigned three possible values:

- $\top$: The analysis did not process the affinity of the variable.
- c: A constant value that is known at compile time.
- $\bot$: At compile time is not possible to define any information about the factor.

Based on these possible element states, all possible abstract states are:

- $(\top, \top)$: The variable is still not processed by the analysis.
- $(0, c)$: The variable is *uniform* and will hold a constant value that is known at compile time.
- $(0, \bot)$: The variable is *uniform*, but the value is not known at compile time.
- (c_1, c_2): The variable is *divergent*, but we name it as *affine* because it is possible to track the affinity with $\mathtt{T}_{id}$. It is possible to the compiler use the constants c_1 and c_2 to write instructions that rematerialize the variable value.
- $(c_1, \bot)$: The variable is *divergent*, but we name it as *affine* because it is possible to track the affinity with $\mathtt{T}_{id}$. It is possible to the compiler use the constant c and a *uniform variable* to write instructions that rematerialize the variable value.
- $(\bot, \bot)$: The variable is *divergent* or no information about the value at run-time is known during compilation.

The affine technique rely on a *dependency graph* that takes into consideration the operations among variables, and uses different propagation rules to determine the output abstract state based on the input ones. Special GPU variables, such as $\mathtt{T}_{id}$, results of atomic instructions, constant values, *kernel* parameters have their values statically predefined. All variables that have their abstract state predefined are inserted in the work-list. For each variable in the work-list we remove it and use propagation rules to compute the abstract state of all variables that depend on it. Always that the abstract state of a variable is altered, that variable is included in the work-list. We iterate until

the work-list is empty and the graph reach a fixed-point. Figure 3.3 demonstrates this process.

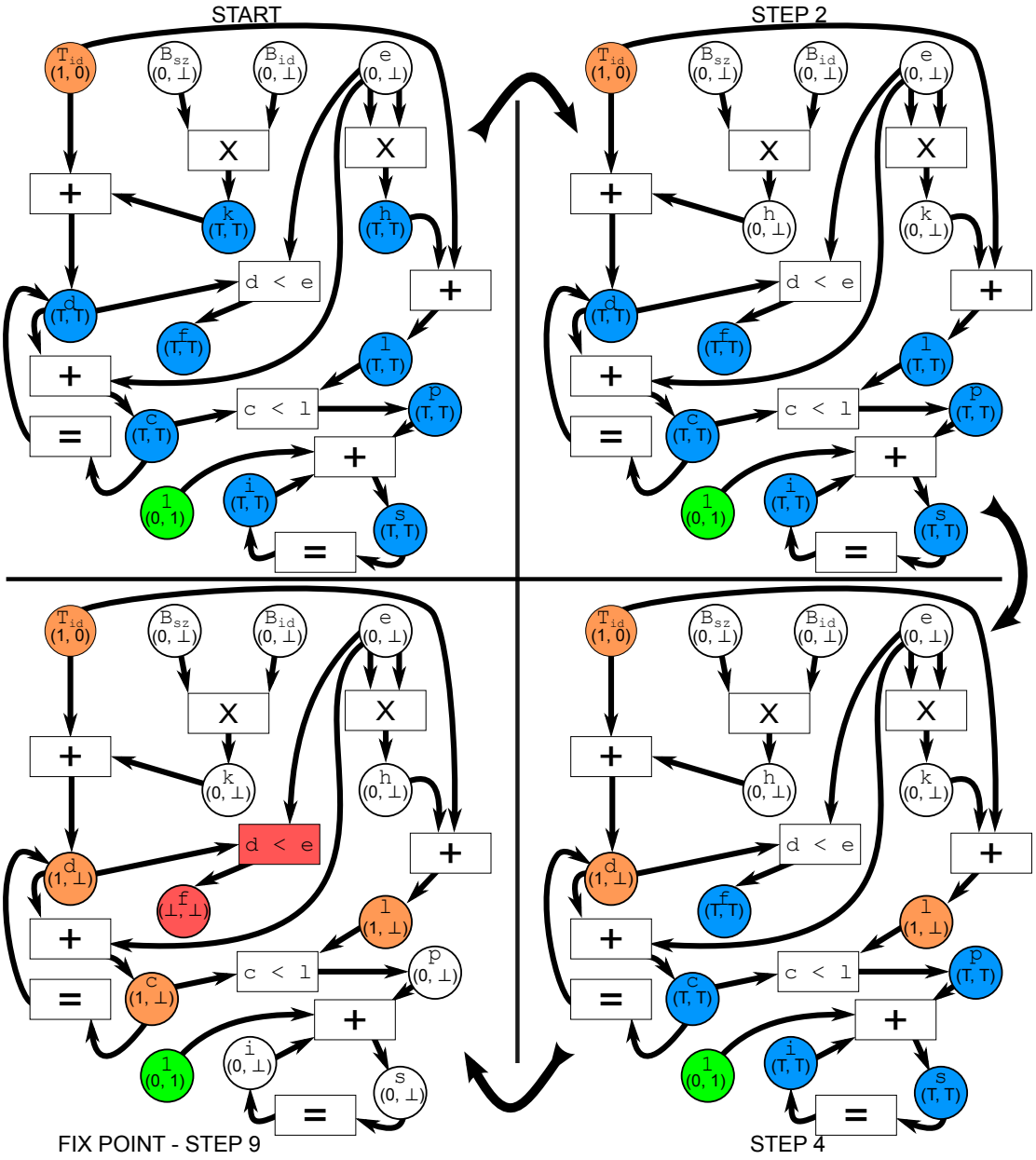


Figure 3.3. On the top left the initial state of the variable dependency graph. Blue variables have a undefined abstract state. Especial variables, such as $\mathtt{T}_{id}$, $\mathtt{B}_{id}$ and $\mathtt{B}_{sz}$, constants, such as `1`, and function arguments, such as `e`, have predefined abstract states. From these variables the state of all others are defined.

Now that we described how both techniques of divergence analysis work, we are going to prove and formalize its correctness. In Section 3.2 we introduce μ-SIMD, a tool

language that describes the rules followed by *threads* that execute in a same *warp*. In Section 3.3 we demonstrate how we transform our program *CFG* so it can inform *control dependencies*. In Section 3.4 we prove the *Simple Divergence* technique, as it help us to prove the correctness of our *Affine Divergence Analysis* in Section 3.5.

3.2 The Core Language

In order to formalize our theory, we adopt the same model of SIMD execution independently described by Bougé and Levaire [1992] and Farrell and Kieronska [1996]. We have a number of PEs executing instructions in lock-step, yet subject to *partial execution*. In the words of Farrel *et al.*, "All processing elements execute the same statement at the same time with the internal state of each processing element being either active or inactive." [Farrell and Kieronska, 1996, p.40]. The archetype of a SIMD machine is the *ILLIAC IV* Computer (see Bouknight et al. [1972]), and there exist many old programming languages that target this model (Abel et al. [1969]; Bouknight et al. [1972]; Brockmann and Wanka [1997]; Hoogvorst et al. [1991]; Kung et al. [1982]; Lawrie et al. [1975]; Perrott [1979]). The recent developments in graphics cards brought new members to this family. The SIMT execution model is currently implemented as a multi-core SIMD machine – CUDA being a framework that coordinates many SIMD processors. We formalize the SIMD execution model via a core language that we call μ-SIMD, and whose syntax is given in Table 3.1. We do not reuse the formal semantics of Bougé *et al.* or Farrell *et al.* because they assume high-level languages, whereas our techniques are better described at the assembly level. Notice that our model will not fit *vector instructions*, popularly called SIMD, such as *Intel's MMX and SSE extensions*, because they do not support partial execution, rather following the semantics of Carnegie Mellon's Vcode (Blelloch and Chatterjee [1990]). An interpreter for μ-SIMD, written in Prolog, plus many example programs, are available in our web-page[1].

We define an abstract machine to evaluate μ-SIMD programs. The state M of this machine is determined by a tuple with five elements: $(\Theta, \Sigma, \Pi, P, pc)$, which we define in Table 3.2. A processing element is a pair (t, σ), uniquely identified by the natural t, referred by the special variable $\mathtt{T}_{id}$. The symbol σ represents the PE's local memory, a function that maps variables to integers. The local memory is individual to each PE; however, these functions have the same domain. Thus, $v \in \sigma$ denotes a vector of variables, each of them private to a PE. PEs can communicate through a shared array Σ. We use Θ to designate the set of active PEs. A program P is a map of

[1] http://divmap.wordpress.com/

Labels	::=	$l \subset \mathbb{N}$
Constants (C)	::=	$c \subset \mathbb{N}$
Variables (V)	::=	$\mathtt{T}_{id} \cup \{v_1, v_2, \ldots\}$
Operands ($V \cup C$)	::=	$\{o_1, o_2, \ldots\}$
Instructions	::=	
– (jump if zero/not zero)	|	$\mathtt{bz/bnz}\ v, l$
– (unconditional jump)	|	$\mathtt{jump}\ l$
– (store into shared memory)	|	$\uparrow v_x = v$
– (load from shared memory)	|	$v = \downarrow v_x$
– (atomic increment)	|	$v \overset{a}{\leftarrow} v_x + 1$
– (binary addition)	|	$v_1 = o_1 + o_2$
– (binary multiplication)	|	$v_1 = o_1 \times o_2$
– (other binary operations)	|	$v_1 = o_1 \oplus o_2$
– (simple copy)	|	$v = o$
– (synchronization barrier)	|	`sync`
– (halt execution)	|	`stop`

Table 3.1. The syntax of μ-Simd instructions

(Local memory)	$\sigma \subset \mathit{Var} \mapsto \mathbb{Z}$
(Shared vector)	$\Sigma \subset \mathbb{N} \mapsto \mathbb{Z}$
(Active PEs)	$\Theta \subset (\mathbb{N} \times \sigma)$
(Program)	$P \subset \mathit{Lbl} \mapsto \mathit{Inst}$
(Sync stack)	$\Pi \subset \mathit{Lbl} \times \Theta \times \mathit{Lbl} \times \Theta \times \Pi$

Table 3.2. Elements that constitute the state of a μ-Simd program

labels to instructions. The *program counter* (pc) is the label of the next instruction to be executed. The machine contains a *synchronization stack* Π. Each node of Π is a tuple $(l_{id}, \Theta_{done}, l_{next}, \Theta_{todo})$ that denotes a point where divergent PEs must synchronize. These nodes are pushed into the stack when the PEs diverge in the control flow. The label l_{id} denotes the conditional branch that caused the divergence, Θ_{done} are the PEs that reached the synchronization point, whereas Θ_{todo} are the PEs waiting to execute. The label l_{next} indicates the instruction where Θ_{todo} will resume execution.

The result of executing a μ-Simd abstract machine is a pair (Θ, Σ). Figure 3.2 describes the big-step semantics of instructions that change the program's control flow. A program terminates if $P[pc] = \mathtt{stop}$. The semantics of conditionals is more elaborate. Upon reaching $\mathtt{bz}\ v, l$ we evaluate v in the local memory of each active PE. If $\sigma(v) = 0$ for every PE, then Rule Bf moves the flow to the next instruction, i.e., $pc+1$. Similarly, if $\sigma(v) \neq 0$ for every PE, then in Rule Bt we jump to the instruction at $P[l]$. However, if we get distinct values for different PEs, then the branch is *divergent*.

$$\textbf{split}(\Theta, v) = (\Theta_0, \Theta_n) \textbf{ where}$$
$$\Theta_0 = \{(t,\sigma) \mid (t,\sigma) \in \Theta \textbf{ and } \sigma[v] = 0\}$$
$$\Theta_n = \{(t,\sigma) \mid (t,\sigma) \in \Theta \textbf{ and } \sigma[v] \neq 0\}$$

$$\textbf{push}([], \Theta_n, pc, l) = [(pc, [], l, \Theta_n)]$$

$$\textbf{push}((pc', [], l', \Theta'_n) : \Pi, \Theta_n, pc, l) = \Pi' \textbf{ if } pc \neq pc'$$
$$\textbf{where } \Pi' = (pc, [], l, \Theta_n) : (pc', [], l', \Theta'_n) : \Pi$$

$$\textbf{push}((pc, [], l, \Theta'_n) : \Pi, \Theta_n, pc, l) = (pc, [], l, \Theta_n \cup \Theta'_n) : \Pi$$

Table 3.3. The auxiliary functions used in the definition of μ-SIMD

In this case, in Rule BD we execute the PEs in the "then" side of the branch, keeping the other PEs in the sync-stack to execute them later. Stack updating is performed by the **push** function in Figure 3.3. Even the non-divergent branch rules update the synchronization stack, so that, upon reaching a barrier, i.e, a `sync` instruction, we do not get stuck trying to pop a node. In Rule SS, if we arrive at the barrier with a group Θ_n of PEs waiting to execute, then we resume their execution at the "else" branch, keeping the previously active PEs into hold. Finally, if we reach the barrier without any PE waiting to execute, in Rule SP we synchronize the "done" PEs with the current set of active PEs, and resume execution at the next instruction after the barrier. Notice that, in order to avoid deadlocks, we must assume that a branch and its corresponding synchronization barrier determine a *single-entry-single-exit* region in the program's CFG [Ferrante et al., 1987, p.329].

Table 3.2 shows the semantics of the rest of μ-SIMD's instructions. A tuple $(t, \sigma, \Sigma, \iota)$ denotes the execution of an instruction ι by a PE (t, σ). All the active PEs execute the same instruction at the same time. We model this phenomenon by showing, in Rule TL, that the order in which different PEs process ι is immaterial. Thus, an instruction such as $v = c$ causes every active PE to assign the integer c to its local variable v. We use the notation $f[a \mapsto b]$ to denote the updating of function f; that is, $\lambda x.x = a \ ? \ b : f(x)$. The store instruction might lead to a data-race, i.e., two PEs trying to write on the same location in the shared vector. In this case, the result is undefined due to Rule TL. We guarantee atomic updates via $v \overset{a}{\leftarrow} v_x + 1$, which reads the value at $\Sigma(\sigma(v_x))$, increments it by one, and stores it back. This result is also copied to $\sigma(v)$, as we see in Rule AT. In Rule BP we use the symbol $\otimes$ to evaluate binary operations, such as addition and multiplication, using the semantics usually seen in arithmetic.

Figure 3.4 (left) shows the kernel `sumTriangle` from Figure 2.1 written in μ-SIMD. To

$$\text{(Sp)}\quad \frac{P[pc] = \texttt{stop}}{(\Theta, \Sigma, \emptyset, P, pc) \to (\Theta, \Sigma)}$$

$$\text{(Bt)}\quad \frac{\begin{array}{c}P[pc] = \texttt{bz}\ v, l \\ \textbf{split}(\Theta, v) = (\emptyset, \Theta) \qquad \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \\ (\Theta, \Sigma, \Pi', P, l) \to (\Theta', \Sigma')\end{array}}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$\text{(Bf)}\quad \frac{\begin{array}{c}P[pc] = \texttt{bz}\ v, l \\ \textbf{split}(\Theta, v) = (\Theta, \emptyset) \qquad \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \\ (\Theta, \Sigma, \Pi', P, pc + 1) \to (\Theta', \Sigma')\end{array}}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$\text{(Bd)}\quad \frac{\begin{array}{c}P[pc] = \texttt{bz}\ v, l \\ \textbf{split}(\Theta, v) = (\Theta_0, \Theta_n) \qquad \textbf{push}(\Pi, \Theta_n, pc, l) = \Pi' \\ (\Theta_0, \Sigma, \Pi', P, pc + 1) \to (\Theta', \Sigma')\end{array}}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$\text{(Ss)}\quad \frac{\begin{array}{c}P[pc] = \texttt{sync} \\ \Theta_n \neq \emptyset \\ (\Theta_n, \Sigma, (pc', \Theta_0, l, \emptyset) : \Pi, P, l) \to (\Theta', \Sigma')\end{array}}{(\Theta, \Sigma, (pc', \emptyset, l, \Theta_n) : \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$\text{(Sp)}\quad \frac{\begin{array}{c}P[pc] = \texttt{sync} \\ (\Theta_n, \Sigma, (\_, \emptyset, \_, \Theta_0) : \Pi, P, pc + 1) \to (\Theta', \Sigma')\end{array}}{(\Theta_0 \cup \Theta_n, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$\text{(Jp)}\quad \frac{\begin{array}{c}P[pc] = \texttt{jump}\ l \\ (\Theta, \Sigma, \Pi, P, l) \to (\Theta', \Sigma')\end{array}}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

$$\text{(It)}\quad \frac{\begin{array}{c}P[pc] = \iota \\ \iota \notin \{\texttt{stop}, \texttt{bnz}, \texttt{bz}, \texttt{sync}, \texttt{jump}\} \\ (\Theta, \Sigma, \iota) \to (\Theta', \Sigma') \qquad (\Theta', \Sigma', \Pi, pc + 1, \Theta'', \Sigma'')\end{array}}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta'', \Sigma'')}$$

Table 3.4. The semantics of μ-Simd: control flow operations. For conciseness, when two hypotheses hold we use the topmost one. We do not give evaluation rules for `bnz`, because they are similar to those given for `bz`

$$\text{(MM)}\quad \frac{\Sigma(v) = c}{\Sigma \vdash v = c} \qquad \text{(MT)}\quad t, \sigma \vdash \mathtt{T}_{id} = t \qquad \text{(MV)}\quad \frac{v \neq \mathtt{T}_{id} \quad \sigma(v) = c}{t, \sigma \vdash v = c}$$

$$\text{(TL)}\quad \frac{(t, \sigma, \Sigma, \iota) \to (\sigma', \Sigma') \qquad (\Theta, \Sigma', \iota) \to (\Theta', \Sigma")}{(\{(t, \sigma)\} \cup \Theta, \Sigma, \iota) \to (\{(t, \sigma')\} \cup \Theta', \Sigma")}$$

$$\text{(CT)}\quad (t, \sigma, \Sigma, v = c) \to (\sigma \setminus [v \mapsto c], \Sigma)$$

$$\text{(AS)}\quad \frac{t, \sigma \vdash v' = c}{(t, \sigma, \Sigma, v = v') \to (\sigma \setminus [v \mapsto c], \Sigma)}$$

$$\text{(LD)}\quad \frac{t, \sigma \vdash v_x = c_x \qquad \Sigma \vdash c_x = c}{(t, \sigma, \Sigma, v =\!\downarrow v_x) \to (\sigma \setminus [v \mapsto c], \Sigma)}$$

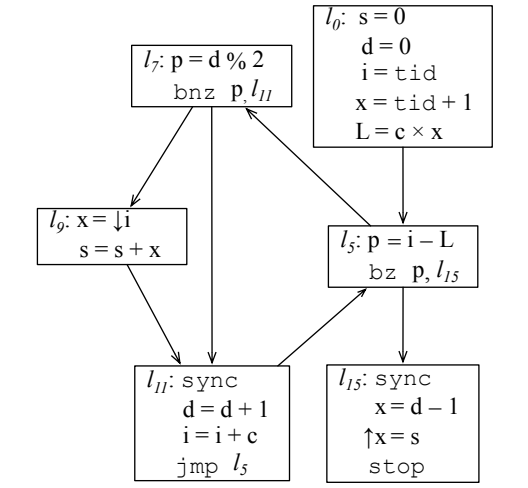
$$\text{(ST)}\quad \frac{t, \sigma \vdash v_x = c_x \qquad t, \sigma \vdash v = c}{(t, \sigma, \Sigma, \uparrow v_x = v) \to (\sigma, \Sigma \setminus [c_x \mapsto c])}$$

$$\text{(AT)}\quad \frac{t, \sigma \vdash v_x = c_x \qquad \Sigma \vdash c_x = c \qquad c' = c + 1}{(t, \sigma, \Sigma, v \xleftarrow{a} v_x + 1) \to (\sigma \setminus [v \mapsto c'], \Sigma \setminus [c_x \mapsto c'])}$$

$$\text{(BP)}\quad \frac{t, \sigma \vdash v_2 = c_2 \qquad t, \sigma \vdash v_3 = c_3 \qquad c_1 = c_2 \otimes c_3}{(t, \sigma, \Sigma, v_1 = v_2 \oplus v_3) \to (\sigma \setminus [v_1 \mapsto c_1], \Sigma)}$$

Table 3.5. The operational semantics of μ-SIMD: data and arithmetic operations

keep the figure clean, we only show the label of the first instruction present in each basic block. This program will be executed by many threads, in lock-step; however, in this case, threads perform different amounts of work: the PE that has $\mathtt{T}_{id} = n$ will visit $n + 1$ cells of the matrix. After a thread leaves the loop, it must wait for the others. Processing resumes once all of them synchronize at label l_{15}. At this point, each thread sees a different value stored at $\sigma(\mathtt{d})$, which has been incremented $\mathtt{T}_{id} + 1$ times. Figure 3.4 (Right) illustrates divergence via a snapshot of the execution of the program seen on the left. We assume that our running program contains four threads: $t_0, \ldots, t_3$. When visiting the branch at label l_6 for the second time, the predicate p is 0 for thread t_0, and 1 for the other PEs. In face of this divergence, t_0 is pushed onto Π, the stack of waiting threads, while the other threads continue executing the loop. When the branch is visited a third time, a new divergence will happen, this time causing t_1 to be stacked for later execution. This pattern will happen again with thread t_2, although we do not show it in Figure 3.4. Once t_3 leaves the loop, all the threads



Cycle	Instruction	t_0	t_1	t_2	t_3
14	$l_5 : p = i - L$	✓	✓	✓	✓
15	$l_6 :$ `bz` p, l_{15}	✓	✓	✓	✓
16	$l_7 : p = d \ \% \ 2$	•	✓	✓	✓
17	$l_8 :$ `bz` p, l_{11}	•	✓	✓	✓
18	$l_9 : x = \downarrow i$	•	✓	✓	✓
19	$l_{10} : s = s + x$	•	✓	✓	✓
20	$l_{11} :$ `sync`	•	✓	✓	✓
21	$l_{12} : d = d + 1$	•	✓	✓	✓
22	$l_{13} : i = i + c$	•	✓	✓	✓
23	$l_{14} :$ `jmp` l_5	•	✓	✓	✓
24	$l_5 : p = i - L$	•	✓	✓	✓
25	$l_6 :$ `bz` p, l_{15}	•	✓	✓	✓
26	$l_7 : p = d \ \% \ 2$	•	•	✓	✓
27	$l_8 :$ `bz` p, l_{11}	•	•	✓	✓
...					
44	$l_5 :$ `bz` p, l_{15}	•	•	•	✓
45	$l_{15} :$ `sync`	✓	✓	✓	✓
46	$l_{16} : x = d - 1$	✓	✓	✓	✓

Execution trace. If a thread t executes an instruction at a cycle j, we mark the entry (t, j) with the symbol ✓. Otherwise, we mark it with the symbol •

Figure 3.4. Execution trace of a μ-SIMD program

synchronize via the `sync` instruction at label l_{15}, and resume lock-step execution.

3.3 Gated Static Single Assignment Form

To better handle control dependency between program variables, we work with μ-SIMD programs in *Gated Static Single Assignment* form (GSA, see Ottenstein et al. [1990]; Tu and Padua [1995]). Figure 3.5 shows the program in Figure 3.4 converted to GSA form. This intermediate program representation differs from the well-known *Static Single Assignment* proposed in Cytron et al. [1991] form because it augments ϕ-functions with the predicates that control them. The GSA form uses three special instructions: μ, γ and η functions, defined as follows:

- γ functions represent the joining point of different paths created by an "if-then-else" branch in the source program. The instruction $v = \gamma(p, o_1, o_2)$ denotes $v = o_1$ if p, and $v = o_2$ if $\neg p$;
- μ functions, which only exist at loop headers, merge initial and loop-carried

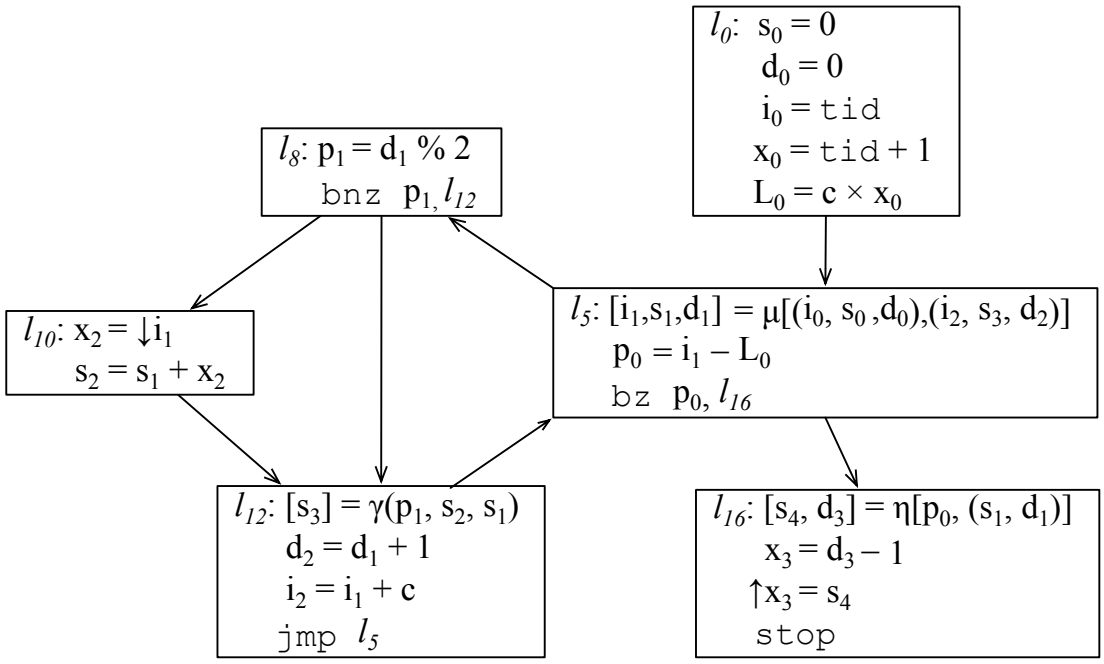


Figure 3.5. The program from Figure 3.4 converted into GSA form

values. The instruction $v = \mu(o_1, o_2)$ represents the assignment $v = o_1$ in the first iteration of the loop, and $v = o_2$ in the others.

- η functions represent values that leave a loop. The instruction $v = \eta(p, o)$ denotes the value of o assigned in the last iteration of the loop controlled by predicate p.

We use the almost linear time algorithm demonstrated by Tu and Padua [1995] to convert a program into GSA form. According to this algorithm, γ and η functions exist at the post-dominator of the branch that controls them. A label l_p post-dominates another label l if, and only if, every path from l to the end of the program goes through l_p. Fung et al. [2007] shown that re-converging divergent PEs at the immediate post-dominator of the divergent branch is nearly optimal with respect to maximizing hardware utilization. Although Fung *et al.* discovered situations in which it is better to do this re-convergence past l_p, they are very rare. Thus, we assume that each γ or η function encodes an implicit synchronization barrier, and omit the `sync` instruction from labels where any of these functions is present. These special functions are placed at the beginning of basic blocks. We use Appel's parallel copy semantics (see Appel [1998]) to evaluate these functions, and we denote these parallel copies using Hack's

matrix notation described in Hack and Goos [2006]. For instance, the μ assignment at l_5, in Figure 3.5 denotes two parallel copies: either we perform $[i_1, s_1, d_1] = (i_0, s_0, d_0)$, in case we are entering the loop for the first time, or we do $[i_1, s_1, d_1] = (i_2, s_3, d_2)$ otherwise.

We work on GSA-form programs because this intermediate representation allows us to transform *control dependency* into *data dependency* when calculating uniform variables. Given a program P, a variable $v \in P$ is data dependent on a variable $u \in P$ if either P contains some assignment instruction $P[l]$ that defines v and uses u, or v is data dependent on some variable w that is data dependent on u. For instance, the instruction $p_0 = i_1 - L_0$ in Figure 3.5 causes p_0 to be data dependent on i_1 and L_0. On the other hand, a variable v is control dependent on u if u controls a branch whose outcome determines the value of v. For instance, in Figure 3.4, s is assigned at l_{10} if, and only if, the branch at l_8 is taken. This last event depends on the predicate p; hence, we say that s is control dependent on p. In the GSA-form program of Figure 3.5, we have that variable s has been replaced by several new variables $s_i, 0 \leq i \leq 4$. We model the old control dependence from s to p by the γ assignment at l_{12}. The instruction $[s_3] = \gamma(p_1, s_2, s_1)$ tells that s_3 is data dependent on s_1, s_2 and p_1, the predicate controlling the branch at l_9.

3.4 The Simple Divergence Analysis

The simple divergence analysis reports if a variable v is *uniform* or *divergent*. We say that a variable is *uniform* if, at compile time, it is know that all threads will see the same value for this variable at run-time. A *uniform* variable meets the condition in Definition 2. If, at compile time, it is not possible to define if the values seen by all threads for a variable are the same, this variable is called a *divergent* variable. For example, loading data from different memory positions into a same variable name might not cause a *data divergence*, but that is not possible to define at compile time. In cases like this the analysis is conservative and classifies these variables as *divergent*. In order to find statically a conservative approximation of the set of uniform variables in a program we solve the constraint system in Figure 3.6. In Figure 3.6 we let $[\![v]\!]$ denote the abstract state associated to variable v. This abstract state can be an element of the lattice $\top < U < D$. This lattice is equipped with a meet operator $\wedge$, such that $a \wedge a = a$, $a \wedge \top = \top \wedge a = a, a \in \{U, D\}$, and $U \wedge D = D \wedge U = D$. In Figure 3.6 we use $o_1 \oplus o_2$ for any binary operator, including addition and multiplication. Similarly, we use $\oplus o$ for any unary operator, including loads.

$$v = c \times \mathtt{T}_{id} \quad [\textsc{TidD}] \quad [\![v]\!] = D \qquad v = \oplus o \quad [\textsc{AsgD}] \quad [\![v]\!] = [\![o]\!]$$

$$v \overset{a}{\leftarrow} v_x + c \quad [\textsc{AtmD}] \quad [\![v]\!] = D \qquad v = c \quad [\textsc{CntD}] \quad [\![v]\!] = U$$

$$v = \gamma[p, o_1, o_2] \quad [\textsc{GamD}] \quad \frac{[\![p]\!] = U}{[\![v]\!] = [\![o_1]\!] \wedge [\![o_2]\!]} \qquad v = \eta[p, o] \quad [\textsc{EtaD}] \quad \frac{[\![p]\!] = U}{[\![v]\!] = [\![o]\!]}$$

$$v = o_1 \oplus o_2 \quad [\textsc{GbzD}] \quad [\![v]\!] = [\![o_1]\!] \wedge [\![o_2]\!]$$

$$v = \gamma[p, o_1, o_2] \text{ or } v = \eta[p, o] \quad [\textsc{PdvD}] \quad \frac{[\![p]\!] = D}{[\![v]\!] = D}$$

$$v = \mu[o_1, \ldots, o_n] \quad [\textsc{RmuD}] \quad [\![v]\!] = [\![o_1]\!] \wedge [\![o_2]\!] \wedge \ldots \wedge [\![o_n]\!]$$

Figure 3.6. Constraint system used to solve the simple divergence analysis

Definition 2 (Uniform variable)**.** *A variable $v \in P$ is uniform if, and only if, for any state $(\Theta, \Sigma, \Pi, P, pc)$, and any $\sigma_i, \sigma_j \in \Theta$, we have that $i, \sigma_i \vdash v = c$ and $j, \sigma_j \vdash v = c$.*

Sparse Implementation. If we see the inference rules in Figure 3.6 as transfer functions, then we can bind them directly to the nodes of the source program's dependence graph. Furthermore, none of these transfer functions is an identity function, as a quick inspection of the rules in Figure 3.8 reveals. Therefore, our analysis admits a *sparse* implementation, as defined by Choi *et al.* Choi et al. [1991]. In the words of Choi *et al.*, sparse data-flow analyses are convenient in terms of space and time because (i) useless information is not represented, and (ii) information is forwarded directly to where it is needed. Because the lattice used in Figure 3.6 has height two, that constraint system can be solved in two iterations of a unification-based algorithm. Moreover, if we initialize every variable's abstract state to U, then the analysis admits a straightforward solution based on graph reachability. As we see from the constraints, a variable v is divergent if either it (i) is assigned a factor of $\mathtt{T}_{id}$, as in Rule TIDD; or (ii) it is defined by an atomic instruction, as in Rule ATMD; or (iii) it is the left-hand side of an instruction that uses a divergent variable. From this observation, we let a *data dependence graph* G that represents a program P be defined as follows: for each variable $v \in P$, let n_v be a vertex of G, and if P contains an instruction that defines variable v, and uses variable u, then we add an edge from n_u to n_v. To find the divergent variables of P, we start from n_{tid}, plus the nodes that represent variables defined by atomic instructions, and mark every variable that is reachable from this set of nodes.

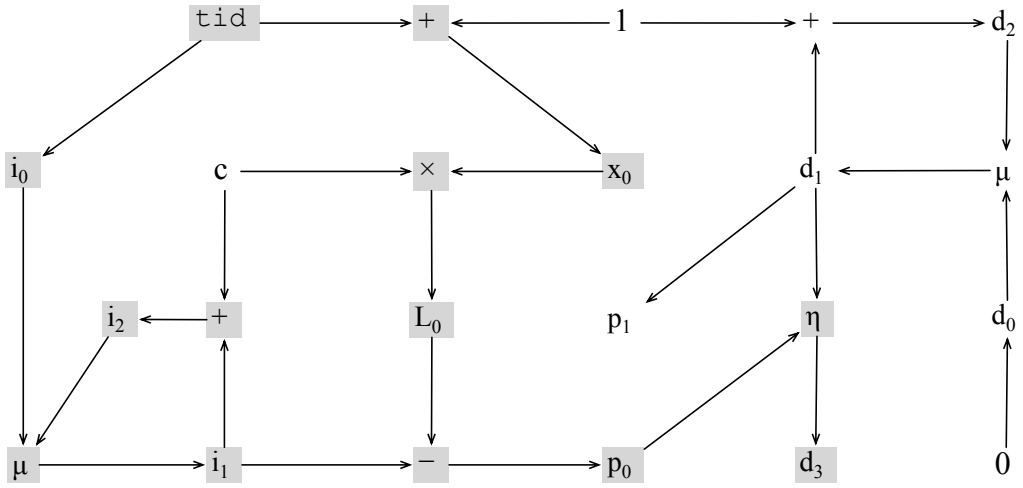


Figure 3.7. The dependence graph created for the program in Figure 3.5. We only show the *program slice* (Weiser [1981]) that creates variables p_1 and d_3. Divergent variables are colored gray.

Moving on with our example, Figure 3.7 shows the data dependence graph created for the program in Figure 3.5. Surprisingly, we notice that the instruction `bnz` p_1, l_{12} cannot cause a divergence, even though the predicate p_1 is data dependent on variable d_1, which is created inside a divergent loop. Indeed, variable d_1 is not divergent, although the variable p_0 that controls the loop is. We prove the non-divergence of d_1 by induction on the number of loop iterations. In the first iteration, every thread sees $d_1 = d_0 = 0$. In subsequent iterations we have that $d_1 = d_2$. Assuming that at the n-th iteration every thread still in the loop sees the same value of d_1, then, the assignment $d_2 = d_1 + 1$ concludes the induction step. Nevertheless, variable d is divergent outside the loop. In this case, we have that d is renamed to d_3 by the η-function at l_{16}. This η-function is data-dependent on p_0, which is divergent. That is, once the PEs synchronize at l_{16}, they might have re-defined d_1 a different number of times. Although this fact cannot cause a divergence inside the loop, divergence might still happen outside it.

Theorem 1. *Let P be a μ-Simd program, and $v \in P$. If $[\![v]\!] = U$, then v is uniform.*

Proof. The proof is a structural induction on the constraint rules used to derive $[\![v]\!] = U$:

- Rule CntD: by Rule Ct, in Table 3.2, we have that $\sigma_i(v) = c$ for every i.

- Rule AsgD: if $[\![o]\!] = U$, then by induction we have that $\sigma_i(o) = c$ for every i. By Rule As in Table 3.2 we have that $\sigma_i(v) = \sigma_i(o)$ for every i.

- Rule GbzD: if $[\![o_1]\!] = U$ and $[\![o_2]\!] = U$, by induction we have $\sigma_i(o_1) = c_1$ and $\sigma_i(o_2) = c_2$ for every i. By Rule Bp in Table 3.2 we have that $\sigma_i(v) = c_1 \oplus c_2$ for every i.

- Rule GamD: if $[\![p]\!] = U$, then by induction we have that $\sigma_i(p) = c$ for every i. By Rules Bt or Bf in Table 3.2 we have that all the PEs branch to the same direction. Thus, by the definition of γ-function, v will be assigned the same value o_i for every thread. We then apply the induction hypothesis on o_i.

- Rule EtaD: similar to the proof for Rule GamD.

□

3.5 Divergence Analysis with Affine Constraints

The previous analysis is not precise enough to point that the loop in the kernel `avgSquare` (Figure 2.1) is non-divergent. In this section we fix that omission by equipping the simple divergence analysis with the capacity to associate affine constraints with variables. Let C be the lattice formed by the set of integers $\mathbb{Z}$ augmented with a top element $\top$ and a bottom element $\bot$, plus a meet operator $\wedge$. Given $\{c_1, c_2\} \subset \mathbb{Z}$, Figure 3.6 defines the meet operator, and the abstract semantics of μ-Simd's multiplication and addition. Notice that in Figure 3.6 we do not consider $\top \times a$, for any $a \in C$. This is safe because

1. we are working only with strict programs, i.e., programs in SSA form in which every variable is defined before being used,

2. we process the instructions in a pre-order traversal of the program's dominance tree,

3. in a SSA form program, the definition of a variable always dominates every use of it (see Budimlic et al. [2002]).

4. upon definition, as we shall see in Figure 3.8, every variable receives an abstract value different from $\top$.

$\wedge$	$\top$	c_1	$\bot$
$\top$	$\top$	c_1	$\bot$
c_2	c_2	$c_1 \wedge c_2$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

$\times$	0	c_1	$\bot$
0	0	0	0
c_2	0	$c_1 \times c_2$	$\bot$
$\bot$	0	$\bot$	$\bot$

$+$	c_1	$\bot$
c_2	$c_1 + c_2$	$\bot$
$\bot$	$\bot$	$\bot$

Table 3.6. Abstract semantics of the meet multiplication and addition operators used in the divergence analysis with affine constraints. We let $c_i \in \mathbb{Z}$

We let $c_1 \wedge c_2 = \bot$ if $c_1 \neq c_2$, and $c \wedge c = c$ otherwise. Similarly, we let $c \wedge \bot = \bot \wedge c = \bot$. Notice that C is the lattice normally used in constant propagation; hence, for a proof of monotonicity, see [Aho et al., 2006, p.633-635]. We define A as the product lattice $C \times C$. If (a_1, a_2) are elements of A, we represent them using the notation $a_1\mathtt{T}_{id} + a_2$. We define the meet operator of A as follows:

$$(a_1\mathtt{T}_{id} + a_2) \ \wedge \ (a'_1\mathtt{T}_{id} + a'_2) = (a_1 \wedge a'_1)\mathtt{T}_{id} + (a_2 \wedge a'_2)$$

We let the constraint variable $[\![v]\!] = a_1\mathtt{T}_{id} + a_2$ denote the abstract state associated with variable v. We determine the set of divergent variables in a μ-SIMD program P via the constraint system seen in Figure 3.8. Initially we let $[\![v]\!] = (\top, \top)$ for every v defined in the text of P, and $[\![c]\!] = (0, c)$ for each $c \in \mathbb{Z}$.

Because our underlying lattice has height two, and we are using a product lattice with two sets, the propagation of control flow information is guaranteed by Nielson et al. [1999] to terminate in at most five iterations. Each iteration is linear on the size of the dependence graph, which might be quadratic on the number of program variables, if we allow γ and μ functions to have any number of parameters. Nevertheless, we show in Chapter 5 that our analysis is linear in practice. As an example, Figure 3.9 illustrates the application of the new analysis on the dependence graph first seen in Figure 3.7. Each node has been augmented with its abstract state, i.e., the results of the divergence analysis with affine constraints. This abstract state tells if the variable is uniform or not, as we prove in Theorem 2. Furthermore, if the PEs see v as the same affine function of their thread identifiers, e.g., $v = c_1\mathtt{T}_{id} + c_2, c_1, c_2 \in \mathbb{Z}$, then we say that v is *affine*.

Theorem 2. *If* $[\![v]\!] = 0\mathtt{T}_{id} + a, a \in C$, *then* v *is uniform.*

If $[\![v]\!] = c\mathtt{T}_{id} + a, a \in C, c \in \mathbb{Z}, c \neq 0$, *then* v *is affine.*

$$v = c \times \mathtt{T}_{id} \quad [\textsc{TidA}] \quad [\![v]\!] = c\mathtt{T}_{id} + 0 \qquad v = v' \quad [\textsc{AsgA}] \quad [\![v]\!] = [\![v']\!]$$

$$v \xleftarrow{a} v_x + c \quad [\textsc{AtmA}] \quad [\![v]\!] = \bot\mathtt{T}_{id} + \bot \qquad v = c \quad [\textsc{CntA}] \quad [\![v]\!] = 0\mathtt{T}_{id} + c$$

$$v = \oplus o \quad [\textsc{GuzA}] \quad \frac{[\![o]\!] = 0\mathtt{T}_{id} + a}{[\![v]\!] = 0\mathtt{T}_{id} + (\oplus a)} \qquad v = \oplus o \quad [\textsc{GunA}] \quad \frac{[\![o]\!] = a_1\mathtt{T}_{id} + a_2 \qquad a_1 \neq 0}{[\![v]\!] = \bot\mathtt{T}_{id} + \bot}$$

$$v =\downarrow v_x \quad [\textsc{LduA}] \quad \frac{[\![v_x]\!] = 0\mathtt{T}_{id} + a}{[\![v]\!] = 0\mathtt{T}_{id} + \bot} \qquad v =\downarrow v_x \quad [\textsc{LddA}] \quad \frac{[\![v_x]\!] = a_1\mathtt{T}_{id} + a_2, a_1 \neq 0}{[\![v]\!] = \bot\mathtt{T}_{id} + \bot}$$

$$v = \gamma[p, o_1, o_2] \quad [\textsc{GamA}] \quad \frac{[\![p]\!] = 0\mathtt{T}_{id} + a}{[\![v]\!] = [\![o_1]\!] \wedge [\![o_2]\!]} \qquad v = \eta[p, o] \quad [\textsc{EtaA}] \quad \frac{[\![p]\!] = 0\mathtt{T}_{id} + a}{[\![v]\!] = [\![o]\!]}$$

$$v = o_1 + o_2 \quad [\textsc{SumA}] \quad \frac{[\![o_1]\!] = a_1\mathtt{T}_{id} + a_1' \qquad [\![o_2]\!] = a_2\mathtt{T}_{id} + a_2'}{[\![v]\!] = (a_1 + a_2)\mathtt{T}_{id} + (a_1' + a_2')}$$

$$v = o_1 \times o_2 \quad [\textsc{MlvA}] \quad \frac{[\![o_1]\!] = a_1\mathtt{T}_{id} + a_1' \qquad [\![o_2]\!] = a_2\mathtt{T}_{id} + a_2' \qquad a_1, a_2 \neq 0}{[\![v]\!] = \bot\mathtt{T}_{id} + \bot}$$

$$v = o_1 \times o_2 \quad [\textsc{MlcA}] \quad \frac{[\![o_1]\!] = a_1\mathtt{T}_{id} + a_1' \qquad [\![o_2]\!] = a_2\mathtt{T}_{id} + a_2' \qquad a_1 \times a_2 = 0}{[\![v]\!] = (a_1 \times a_2' + a_1' \times a_2)\mathtt{T}_{id} + (a_1' \times a_2')}$$

$$v = o_1 \oplus o_2 \quad [\textsc{GbzA}] \quad \frac{[\![o_1]\!] = 0\mathtt{T}_{id} + a_1' \qquad [\![o_2]\!] = 0\mathtt{T}_{id} + a_2'}{[\![v]\!] = 0\mathtt{T}_{id} + (a_1' \oplus a_2')}$$

$$v = o_1 \oplus o_2 \quad [\textsc{GbnA}] \quad \frac{[\![o_1]\!] = a_1\mathtt{T}_{id} + a_1' \qquad [\![o_2]\!] = a_2\mathtt{T}_{id} + a_2' \qquad a_1, a_2 \neq 0}{[\![v]\!] = \bot\mathtt{T}_{id} + \bot}$$

$$v = \gamma[p, o_1, o_2] \text{ or } v = \eta[p, o] \quad [\textsc{PdvA}] \quad \frac{[\![p]\!] = a\mathtt{T}_{id} + a', a \neq 0}{[\![v]\!] = \bot\mathtt{T}_{id} + \bot}$$

$$v = \mu[o_1, \ldots, o_n] \quad [\textsc{RmuA}] \quad [\![v]\!] = [\![o_1]\!] \wedge [\![o_2]\!] \wedge \ldots \wedge [\![o_n]\!]$$

Figure 3.8. Constraint system used to solve the divergence analysis with affine constraints of degree one

Proof. The proof is by structural induction on the rules in Figure 3.8. We will show a few cases:

- CNTA: a variable initialized with a constant is uniform, given Rule CT in Table 3.2. Rule CNTA assigns the coefficient zero to the abstract state of this variable.
- SUMA: if the hypothesis holds by induction, then we have four cases to consider. (i) If v_1 and v_2 are uniform, then $[\![v_1]\!] = 0\mathtt{T}_{id} + a_1$, and $[\![v_2]\!] = 0\mathtt{T}_{id} + a_2$, where $a_1, a_2 \in C$. Thus, $[\![v]\!] = (0+0)\mathtt{T}_{id} + (a_1 + a_2)$. By hypothesis, a_1 and a_2 have the same value for every processing element, and so do $a_1 + a_2$. (ii) If v_1 and v_2 are affine, then we have $[\![v_1]\!] = c_1\mathtt{T}_{id} + a_1$, and $[\![v_2]\!] = c_2\mathtt{T}_{id} + a_2$, where $c_1, c_2 \in \mathbb{Z}$ and $a_1, a_2 \in C$. Thus, $[\![v]\!] = (c_1 + c_2)\mathtt{T}_{id} + (a_1 + a_2)$, and the result holds for the same

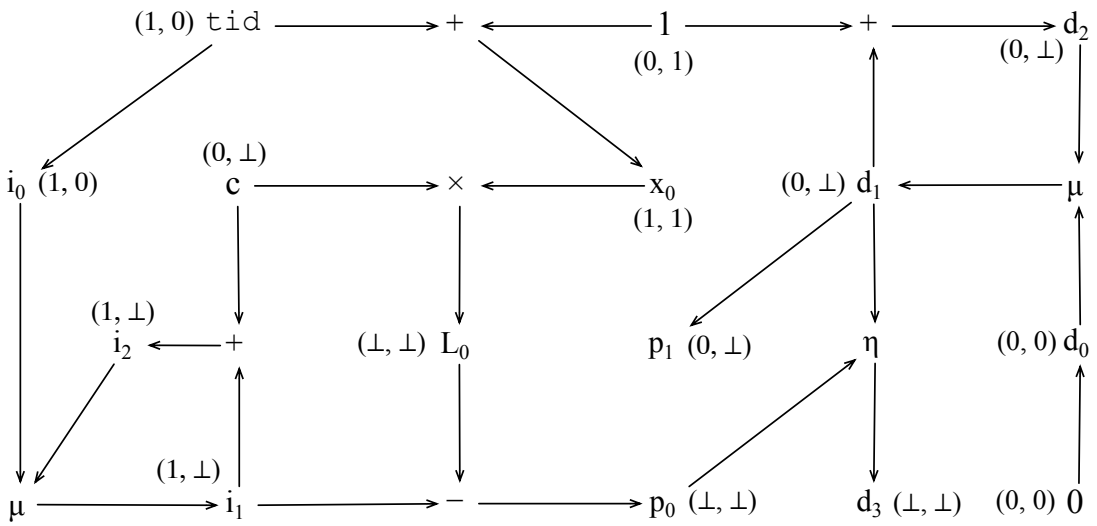


Figure 3.9. Results of the divergence analysis with affine constraints for the program slice first seen in Figure 3.7

reasons as in (i). (iii) It is possible that $c_1 = -c_2$; thus, $c_1 + c_2 = 0$. Because v_1 and v_2 are affine, each variable is made off a factor of $\mathtt{T}_{id}$ plus a constant parcel a for every PE. The sum of these constant parcels, e.g., $a_1 + a_2$ is still constant for every PE; hence, v is uniform. (iv) Finally, if one of the operands of the sum is divergent, then v will be divergent, given our abstract sum operator defined in Figure 3.6. These four cases abide by the semantics of addition, if we replace $\oplus$ by $+$ in Rule Bp of Table 3.2.

- EtaA: we know that p is uniform; hence, by either Rule Bt or Bf in Tablea 3.2, PEs reach the end of the loop at the same time. If o is uniform, it has the same value for every PE at the end of the loop. If it is affine, it has the same $\mathtt{T}_{id}$ coefficient at that moment. Thus, v is either uniform or affine, by Rule As from Tablea 3.2.

- GamA: by hypothesis we know that $[\![v]\!] = 0\mathtt{T}_{id} + a$. Thus, by induction we know that p is uniform. A branch on a uniform variable leads all the threads on the same path, due to either Rule Bt or Bf in Tablea 3.2. There are then three cases to consider, depending on $[\![o_1]\!]$and $[\![o_2]\!]$. (i) If $[\![o_1]\!] = 0\mathtt{T}_{id} + c_1$ and $[\![o_2]\!] = 0\mathtt{T}_{id} + c_2$, then by induction these two variables are uniform, and their meet is also uniform. (ii) If $[\![o_1]\!] = c\mathtt{T}_{id} + c_1$ and $[\![o_2]\!] = c\mathtt{T}_{id} + c_1$, then by induction

these two variables are affine, with the same coefficient of $\mathtt{T}_{id}$. Their meet is also affine with a $\mathtt{T}_{id}$ coefficient equal to c. (iii) Otherwise, we conservatively assign $[\![v]\!]$ the $\bot$ coefficient as defined by the $\wedge$ operator.

The other rules are similar. □

The divergence analysis with affine constraints subsumes the simple divergence analysis of Section 3.4, as Corollary 1 shows.

Corollary 1. *If the simple divergence analysis says that variable v is uniform, then the divergence analysis with affine constraints says that v is uniform.*

Proof. Because both analyses use the same intermediate representation, they work on the same program dependence graph. In Section 3.4's analysis, v is uniform if it is a function of only uniform variables, e.g., $v = f(v_1, \ldots, v_n)$, and every $v_i, 1 \leq i \leq n$ is uniform. From Theorem 1, we know that if $[\![v_i]\!] = 0\mathtt{T}_{id} + c_i$ for every $i, 1 \leq i \leq n$, then v is uniform. □

Is there a case for higher-degree polynomials? Our analysis, as well as constant propagation, are special cases of a more extensive framework which we call *the divergence analysis with polynomial constraints*. In the general case, we let $[\![v_i]\!] = a_n\mathtt{T}_{id}^n + a_{n-1}\mathtt{T}_{id}^{n-1} + \ldots + a_1\mathtt{T}_{id} + a_0$, where $a_i \in C, 1 \leq i \leq n$. Addition and multiplication of polynomials follow the usual algebraic rules. The rules in Figure 3.8 use polynomials of degree one. Constant propagation uses polynomials of degree zero.

There are situations in which polynomials of degree two let us find more affine variables. The extra precision comes out of Theorem 3. Consider, for instance, the program in Figure 3.10, which assigns to each processing element the task of initializing the rows of a matrix `m` with one's. The degree-one divergence analysis would conclude that variables i_0, i_1 and i_2 are divergent. However, the degree-two analysis finds that the highest coefficient of any of these variables is zero; thus flagging them as affine functions of $\mathtt{T}_{id}$. In our benchmarks the degree-2 analysis marked 39 more variables, out of almost 10,000, as affine, when compared to the degree-1 analysis. We did not try higher level as the gains for a second degree level where minimal.

Theorem 3. *If $[\![v]\!] = 0\mathtt{T}_{id}^2 + a_1\mathtt{T}_{id} + a_0, a_1, a_0 \in C$, then v is affine function of $\mathtt{T}_{id}$.*

This proof is also a structural induction on the extended constraint rules for polynomials of degree two. We omit it, because it is very similar to the proof of Theorem 2.

```
void s(int* m, int n) {
  int i0 = n * tid, k = (tid + 1) * n;
  while (i1 = γ(i0, i2); i1 < k) {
    m[i1] = 1;
    i2 = i1 + 1;
  }
}
```

	(a_1, a_0)	(a_2, a_1, a_0)
n	$(0, \bot)$	$(0, 0, \bot)$
i_0	$(\bot, 0)$	$(0, \bot, 0)$
i_1	$(\bot, \bot)$	$(0, \bot, \bot)$
i_2	$(\bot, \bot)$	$(0, \bot, \bot)$
k	$(\bot, \bot)$	$(0, \bot, \bot)$

Figure 3.10. An example where a higher degree polynomial improves the precision of the simple affine analysis

3.6 Chapter conclusion

In this chapter we formalized *divergence analysis with affine constraints*, explaining how it tracks variables relations to the $\mathtt{T}_{id}$. To pave the path that leads to our divergence analysis, we had to explain the notion of data and control dependence and how GSA helps to transform control dependence into data dependence. We also proved that our divergence analysis is correct based on the μ-SIMD language.

We also demonstrated that previously proposed divergence analysis were capable to classify variables only as *uniform* and *divergent*, as our *Divergence Analysis with Affine Constraints* (Section 3.5) is capable to classify variables as: *constant*, *uniform*, *constant affine*, *affine* and *divergent*. *Constant* and *uniform* variables hold the same value in all threads. Although variables classified as *constant affine* and *affine* are know to hold different values among the threads, at compile time is known a recipe, to recalculate these variables bases on the $\mathtt{T}_{id}$. At last, no information is known, at compile time, of run-time values kept by *divergent* variables.

We will present in Chapter 4 a compiler optimization that uses information generated by divergence analyses to write better GPU code and in Chapter 5 comparisons between the results archived by such optimizations.

Chapter 4

Divergence Aware Register Allocation

"Because of the central role that register allocation plays, both in speeding up the code and in making other optimizations useful, it is one of the most important - if not the most important - of the optimizations."

Hennessy and Patterson [2006][Appendix B; pg. 26]

Register allocation is the compiler task of finding storage to program variables. Different from programming to a CPU, where the programmer only sees one type of memory and the hardware is responsible to handle the cache memory, GPUs give different memory mappings to the programmer, each with different time access or visibility among threads. Finding an optimal register allocation is a very hard task; thus, many compilers resort to heuristics to deal with it. A very popular heuristics, in this case, is graph coloring (see Chaitin [1982]; Hennessy and Patterson [2006]). In this Chapter we introduce the *divergence aware register allocator*. This register allocator is able to spill variables into different memories of the GPU, depending on the divergence classification these spilled values.

Our optimizations rely on increasing data sharing among threads of the same *warp*, making it possible to replace slow access to off-chip memory by the fast access of the shared memory faster memory, held on-chip and shared among threads of a SM. The GPU L1 cache memory and shared memory use the same structure and therefore have the same access time. Lashgar and Baniasadi [2011] showed that a high cache

miss rate due a memory access with bad locality can have more performance effect than *control divergence* due the discrepancy of on-chip and off-chip memory access time. Rogers et al. [2012] introduced a *warp* scheduler that takes into consideration data stored in the cache memory to decide which *warp* to execute. His scheduler obtained a 27% performance gain over the previous state of the art *warp* scheduler, demonstrating the impact of better GPU cache memory usage. It also demonstrated that a theoretical GPU with 8MB of cache per SM could boost the Instructions per Clock of cache sensitive applications by factor up to 34 times, when compared to a cache of 32KB, a cache size found in many real-world GPUs. Rogers et al. [2012] work focus on maximizing re-usage of data stored on L1 cache among *warps*, as our optimization propose to reduce the amount of cache used by a *warp*, as shared memory access are not cached and we would use only one memory position to store common values.

On this chapter we will give further details on GPUs memory hierarchy on Section 4.1, and how conventional register allocator behave when GPU variables are spilled. Section 4.2 demonstrates how to adapt conventional register allocator to become divergence aware. This allocator generates code that suits better the needs of the GPU memory hierarchy.

4.1 GPU memory hierarchy

Similar to traditional register allocation, we are interested in finding storage area to the values produced during program execution. However, in the context of graphics processing units, we have different types of memory to consider:

- **Registers:** these are the fastest storage regions. A traditional GPU might have a very large number of registers, for instance, one streaming multiprocessor (SM) of a GTX 580 GPU has 32,768 registers. However, running 1,536 threads at the same time, this SM can afford at most 20 registers to each thread in order to achieve maximum hardware occupancy.

- **Shared memory:** this fast memory is addressable by each thread in flight, and usually is used as a scratchpad cache. It must be used carefully, to avoid common parallel hazards, such as data races. Henceforth we will assume that accessing data in the shared memory is less than 3 times slower than in registers.

- **Local memory:** this off-chip memory is private to each thread. Modern GPUs provide a cache to the local memory, which is as fast as the shared memory. We will assume that a cache miss is 100 times more expensive than a hit.

- **Global memory:** this memory is shared among all the threads in execution, and is located in the same chip area as the local memory. The global memory is also cached. We shall assume that it has the same access times as the local memory.

As we have seen, the local and the global memories might benefit from a cache, which uses the same access machinery as the shared memory. Usually this cache is small: the GTX 580 has 64KB of fast memory, out of which 48KB are given to the shared memory by default, and only 16KB are used as a cache. This cache area must be further divided between global and local memories.

Given this hardware configuration, we see that the register allocator has the opportunity to keep a single image per *warp* of any spilled value that is uniform. This optimization is very beneficial in terms of time. According to Ryoo et al. [2008], the shared memory has approximately the same latency as an on-chip register access, whereas a non-cached access to the local memory is 200-300 times slower. A divergence aware register allocator has a second advantage: it tends to improve memory locality. The GPU's cache space is severely limited, as it has to be partitioned among the massive number of threads running concurrently. In fact, the capacity of the cache might be much lower than the capacity of the register file itself (see Nickolls and Dally [2010]). When moving non-divergent variables to the shared memory, we only need to store one instance per *warp*, rather than one instance per thread. Thus, the divergence aware register allocator may provide up to a 32-fold improvement in cache locality.

Figure 4.1 shows the instance of the register allocation problem that we obtain from the kernel `avgSquare` in Figure 2.1. There are many ways to model register allocation. We use an approach called *linear scan* (see Poletto and Sarkar [1999]). Thus, we linearize the control flow graph of the program, finding an arbitrary ordering of basic blocks, in such a way that each *live range* is seen as an interval. We use bars to represent the live ranges of the variables. The live range of a variable is the collection of program points where that variable is *alive*. A variable v is *alive* at a program point p if v is used at a program point p' that is reachable from p on the control flow graph, and v is not redefined along this path. The colors of the bars represent the abstract state of the variables, as determined by the divergence analysis.

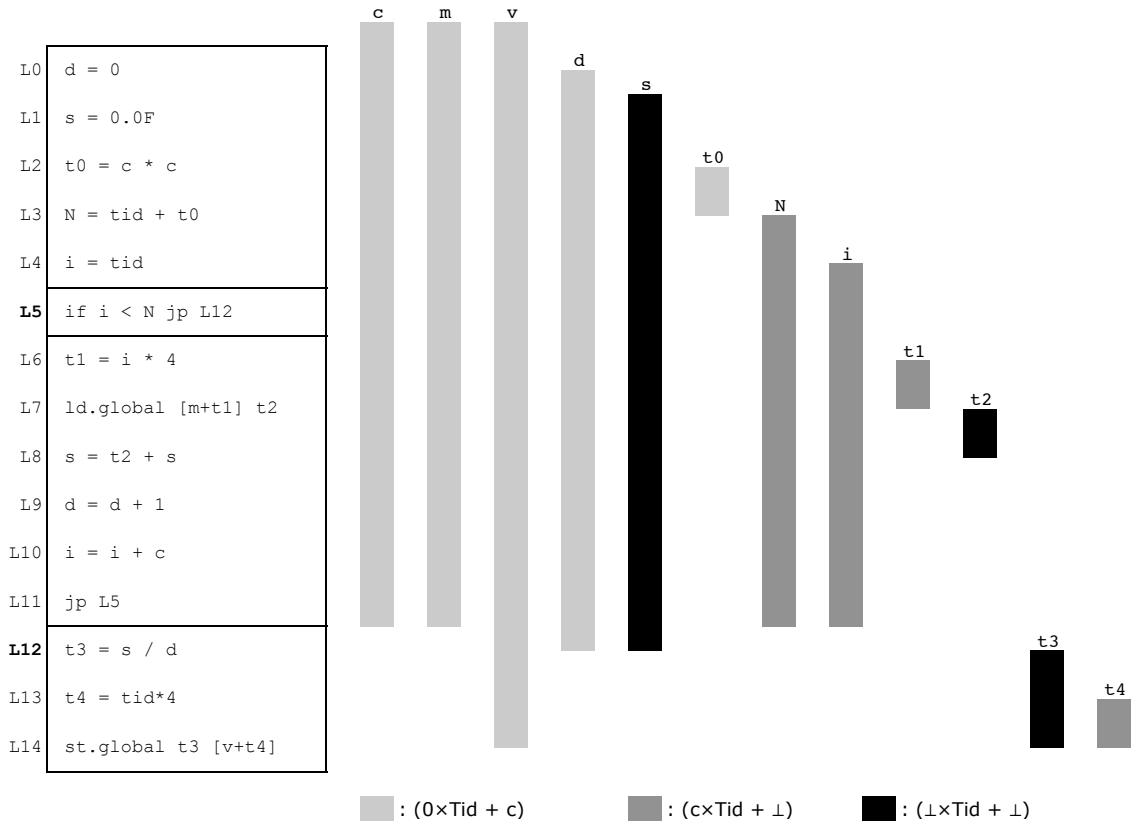


Figure 4.1. The register allocation problem for the kernel `avgSquare` in Figure 2.1

Current register allocators for graphics processing units place spilled values in the local memory. Figure 4.2 illustrates this approach. In this example, we assume a *warp* with two PEs, each one having access to three registers. Given this configuration, variables `s`, `d` and `N` had to be spilled. Thus, each of these variables receive a slot in local memory. The spilled data must be replicated once for each processing element, as each of them has a private local memory area. Accessing data from the local memory is an expensive operation, because this region is off-chip. To mitigate this problem, modern GPUs provide a cache to the local and to the global memories. However, because the number of threads using the cache is large – in the order of thousands – and the cache itself is small, e.g., 16KBs, cache misses are common. In the next section we show that it is possible to improve this situation considerably, by taking the results of the divergence analysis into consideration.

	Program	register file						local						global		
		PE0			PE1			PE0			PE1					
		r0	r1	r2	r0	r1	r2	0	1	2	0	1	2	0	1	2
L0	d = 0													c	m	v
L1	**st.local d [1]**	d			d									c	m	v
L2	s = 0.0F	d			d				d			d		c	m	v
L3	**st.local s [0]**	d	s		d	s			d			d		c	m	v
L4	**ld.global [0] c**	d	s		d	s		s	d		s	d		c	m	v
L5	t0 = c * c	d	s	c	d	s	c	s	d		s	d		c	m	v
L6	N = tid + t0	t0	s	c	t0	s	c	s	d		s	d		c	m	v
L7	**st.local N [2]**	t0	s	N	t0	s	N	s	d		s	d		c	m	v
L8	i = tid	t0	s	N	t0	s	N	s	d	N	s	d	N	c	m	v
L9	**ld.local [2] N**	i	s	N	i	s	N	s	d	N	s	d	N	c	m	v
L10	if i < N jp L24	i	s	N	i	s	N	s	d	N	s	d	N	c	m	v
L11	t1 = i * 4	i	s	N	i	s	N	s	d	N	s	d	N	c	m	v
L12	**ld.global [1] m**	i	s	t1	i	s	t1	s	d	N	s	d	N	c	m	v
L13	ld.global [m+t1] t2	i	m	t1	i	m	t1	s	d	N	s	d	N	c	m	v
L14	**ld.local [0] s**	i	m	t2	i	m	t2	s	d	N	s	d	N	c	m	v
L15	s = t2 + s	i	s	t2	i	s	t2	s	d	N	s	d	N	c	m	v
L16	**st.local s [0]**	i	s	t2	i	s	t2	s	d	N	s	d	N	c	m	v
L17	**ld.local [1] d**	i	s	t2	i	s	t2	s	d	N	s	d	N	c	m	v
L18	d = d + 1	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
L19	**st.local d [1]**	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
L20	**ld.global [0] c**	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
L21	i = i + c	i	s	c	i	s	c	s	d	N	s	d	N	c	m	v
L22	jp L9	i	s	c	i	s	c	s	d	N	s	d	N	c	m	v
L23	**ld.local [1] d**	i	s	c	i	s	c	s	d	N	s	d	N	c	m	v
L24	t3 = s / d	i	s	d	i	s	d	s	d	N	s	d	N	c	m	v
L25	t4 = tid*4	t3	s	d	t3	s	d	s	d	N	s	d	N	c	m	v
L26	**ld.global [2] v**	t3	t4	d	t3	t4	d	s	d	N	s	d	N	c	m	v
L27	st.global t3 [v+t4]	t3	t4	v	t3	t4	v	s	d	N	s	d	N	c	m	v

Figure 4.2. Traditional register allocation, with spilled values placed in local memory

4.2 Adapting a Traditional Register Allocator to be Divergence Aware

To accommodate the notion of local memory in μ-SIMD, we augment its syntax with two instructions to manipulate this memory. An instruction such as $v =\Downarrow v_x$ denotes a load of the value stored at local memory address v_x into v. The instruction $\Uparrow v_x = v$ represents a store of v into the local memory address v_x. The table in Figure 4.2 shows how we replace loads and stores to the local memory by more efficient instructions. The figure describes a re-writing system: we replace loads-to and stores-from local memory

	$[\![v]\!]$	Load sequence	Store sequence
(i)	$(0, c)$	$v = c$	$\emptyset$
(ii)	$(0, \bot)$	$v =\downarrow v_x$	$\uparrow v_x = v$
(iii)	(c_1, c_2)	$v = c_1\mathtt{T}_{id} + c_2$	$\emptyset$
(iv)	$(c, \bot)$	$t =\downarrow v_x; v = c\mathtt{T}_{id} + t$	$t = v_x - c\mathtt{T}_{id}; \uparrow v_x = v$

Rewriting rules that replace loads ($v =\Downarrow v_x$) and stores ($\Uparrow v_x = v$) to local memory with faster instructions. The arrows $\uparrow, \downarrow$ represent accesses to shared memory.

Figure 4.3. Using faster memory for spills

by the sequences in the table, whenever the variable has the abstract state in the second column. In addition to moving uniform values to shared memory, we propose a form of Briggs's style rematerialization Briggs et al. [1992] that suits SIMD machines. The lattice that we use in Figure 3.8 is equivalent to the lattice used by Briggs *et al.* in their rematerialization algorithm. Thus, we can naturally perform rematerialization for an uniform variable which has statically known-values, i.e., $[\![v_x]\!] = (0\mathtt{T}_{id}, c)$, as in line (i) of Figure 4.2 or $[\![v_x]\!] = (c_1\mathtt{T}_{id}, c_2)$, as in line (iii). For the other uniform or affine variables we can move the location of values from the local memory to the shared memory, as we show in lines (ii) and (iv).

Figure 4.4 shows the code that we generate for the program in Figure 4.1. The most apparent departure from the allocation given in Figure 4.2 is the fact that we have moved to shared memory some information that was originally placed in local memory. Variable `d` has been shared among different threads. Notice how the stores at labels `L1` and `L19` in Figure 4.2 have been replaced by stores to shared memory in labels `L1` and `L20` of Figure 4.4. Similar changes happened to the instructions that load `d` from local memory in Figure 4.2. Variable `N` has also been shared; however, contrary to `d`, `N` is not uniform, but affine. If the spilled variable v is an affine expression of the thread identifier, then its abstract state is given by $[\![v]\!] = c\mathtt{T}_{id} + x$, where c is a constant known statically, and x is only known at execution time. In order to implement variable sharing in this case, we must extract x, the unknown part of v, and store it in shared memory. Whenever necessary to reload v, we must get back from shared memory its dynamic component x, and then rebuild v's value from the thread identifier and x. In line `L7` we have stored `N`'s dynamic component. In lines `L9` and `L10` we rebuild the value of `N`, an action that re-writes the load from local memory seen at line `L9` of Figure 4.2.

	Program	register file						local		shared		global		
		PE0			PE1			PE0	PE1					
		r0	r1	r2	r0	r1	r2	0	0	0	1	0	1	2
L0	d = 0											c	m	v
L1	**st.shared d [0]**	d			d							c	m	v
L2	s = 0.0F	d			d					d		c	m	v
L3	**st.local s [0]**	d	s		d	s				d		c	m	v
L4	**ld.global [0] c**	d	s		d	s		s	s	d		c	m	v
L5	t0 = c * c	d	s	c	d	s	c	s	s	d		c	m	v
L6	N = tid + t0	t0	s	c	t0	s	c	s	s	d		c	m	v
L7	**st.shared t0 [1]**	t0	s	N	t0	s	N	s	s	d		c	m	v
L8	i = tid	t0	s	N	t0	s	N	s	s	d	t0	c	m	v
L9	**ld.shared [1] t0**	i	s	N	i	s	N	s	s	d	t0	c	m	v
L10	**N = tid + t0**	i	s	t0	i	s	t0	s	s	d	t0	c	m	v
L11	if i < N jp L24	i	s	N	i	s	N	s	s	d	t0	c	m	v
L12	t1 = i * 4	i	s	N	i	s	N	s	s	d	t0	c	m	v
L13	**ld.global [1] m**	i	s	t1	i	s	t1	s	s	d	t0	c	m	v
L14	ld.global [m+t1] t2	i	m	t1	i	m	t1	s	s	d	t0	c	m	v
L15	**ld.local [0] s**	i	m	t2	i	m	t2	s	s	d	t0	c	m	v
L16	s = t2 + s	i	s	t2	i	s	t2	s	s	d	t0	c	m	v
L17	**st.local s [0]**	i	s	t2	i	s	t2	s	s	d	t0	c	m	v
L18	**ld.shared [0] d**	i	s	t2	i	s	t2	s	s	d	t0	c	m	v
L19	d = d + 1	i	s	d	i	s	d	s	s	d	t0	c	m	v
L20	**st.shared d [0]**	i	s	d	i	s	d	s	s	d	t0	c	m	v
L21	**ld.global [0] c**	i	s	d	i	s	d	s	s	d	t0	c	m	v
L22	i = i + c	i	s	c	i	s	c	s	s	d	t0	c	m	v
L23	jp L9	i	s	c	i	s	c	s	s	d	t0	c	m	v
L24	**ld.shared [0] d**	i	s	c	i	s	c	s	s	d	t0	c	m	v
L25	t3 = s / d	i	s	d	i	s	d	s	s	d	t0	c	m	v
L26	t4 = tid*4	t3	s	d	t3	s	d	s	s	d	t0	c	m	v
L27	**ld.global [2] v**	t3	t4	d	t3	t4	d	s	s	d	t0	c	m	v
L28	st.global t3 [v+t4]	t3	t4	v	t3	t4	v	s	s	d	t0	c	m	v

Figure 4.4. Register allocation with variable sharing.

4.2.1 Handling multiple *warps*

Graphics processing units are not exclusively SIMD machines. Rather, they combine into a single card many SIMD units, or *warps*. Our divergence analysis finds uniform variables per *warp*. Therefore, in order to implement the divergence aware register allocator, we must partition the shared memory among all the *warps* that might run simultaneously. The main advantage of this partitioning is that we do not need to synchronize accesses to the shared memory among different *warps*. On the other hand, the register allocator requires more space in the shared memory. That is, if the allocator finds out that a given program demands N bytes to accommodate the spilled values, and the target GPU runs up to M *warps* simultaneously, then this allocator will need

$M \times N$ bytes in shared memory.

We have presented possible optimizations to the register allocation process based on *data divergence* information. In the next chapter we will present experimental results comparing our optimizations to a standard register allocator and in chapter 6 we will discuss some implementations limitations and how our analysis and optimizations can be improved.

4.3 Chapter conclusion

This chapter demonstrated the process of register allocation, that selects storage for program variables. A traditional GPU register allocator stores into the *local memory* variables that do not fit on registers. The *local memory* has a very high access latency and each *thread* has its own *local memory* mapping, so one memory position is used for each *thread*.

Divergence aware register allocators use divergence analysis to better use GPUs memory hierarchy. Whenever a variable is not *divergent* they can be stored in the *shared memory*, that is much faster than the *local memory*. When store in *local memory* a variable only uses one memory position is used for all threads in a *warp*.

The next chapter will expose experimental results that prove that *divergence aware register allocator* can generate better code to GPUs than a conventional register allocator.

Chapter 5

Experiments

In this chapter we show empirically that, in the context of a GPU, a divergent aware register allocator produces code that outperforms the code produced by a traditional register allocator. In Section 5.1 we describe the hardware and applications used in our experiments, in Section 5.2 we describe the programs used in our tests and in Section 5.3 we evaluate our ideas in terms of the run-time of the code that we produce, the time that our analysis takes to run, and the precision of this analysis.

5.1 Tests

5.1.1 Hardware

We have implemented our divergence analysis plus the divergence aware register allocator on top of the Ocelot open source compiler, revision 1560 of November/2011. We run Ocelot on a quad-core AMD Phenom II 925 processor. Each core has a 2.8GHz CPU clock. This CPU also hosts the GPU that we use to run the kernels: a NVIDIA GTX 570 (Fermi) graphics processing unit.

5.2 Benchmarks

We have successfully tested our divergence analysis in all the 177 different CUDA kernels that we took from the Rodinia (see Che et al. [2009]) and NVIDIA SDK 3.1 benchmark suites. These benchmarks give us 31,487 PTX instructions. We chose to report numbers to the 40 kernels with the longest running times that our divergence aware register allocator produces. The 40 slowest kernels give us over 7,000 PTX

instructions and 9,000 variables – in the GSA-form programs – to analyze. We have more variables than instructions because of the definitions produced by the η, γ and μ functions used to create the GSA intermediate program representation.

5.3 Results

Run-time of the divergence analysis with affine constraints: Figure 5.1 compares the run-time of the two divergence analyses seen in Sections 3.4 and 3.5. On the average, the divergence analysis with affine constraints of degree two is 1.39x slower, even though its lattice of abstract values has height 9, and the lattice used in the simple analysis has height two. The affine analysis of Section 3.5 took 58.6 msecs to go over all the 177 kernels. We measure time in CPU ticks, as given by the `rdtsc` x86 instruction. We have plotted the number of variables per program, considering the 40 chosen kernels only. There is a strong correlation between run-time and number of variables: the coefficient of determination for the simple analysis is 0.957, and for the affine analysis is 0.936. These correlations indicate that in practice both analyses are linear on the number of variables in the target program. Figure 5.2 demonstrates how the our analysis execution time is linear with the number of variables in a kernel, by demonstrating that the averate time spent per kernel variable is almost constant.

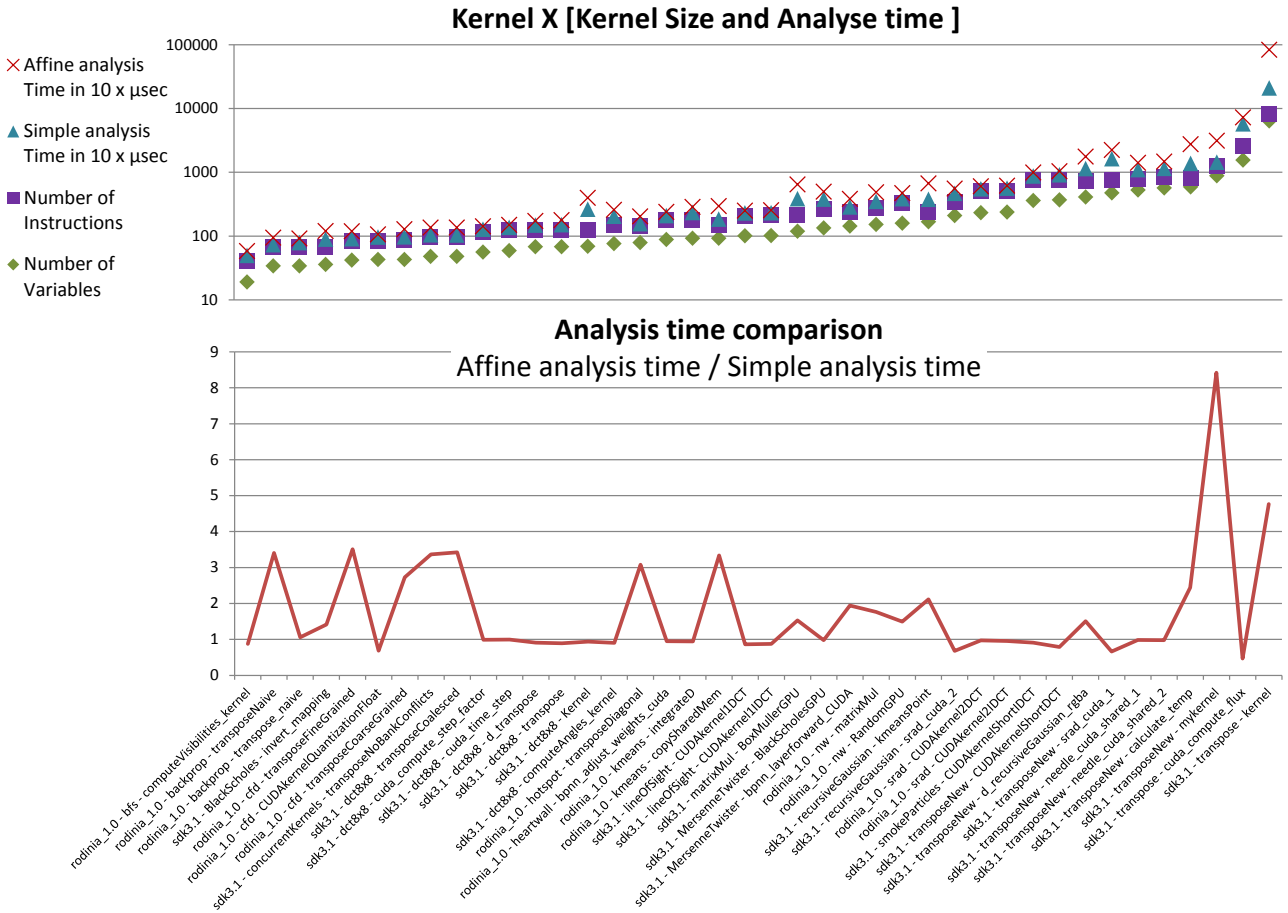


Figure 5.1. Points in the X-axis are kernels, sorted by the number of variables they contain. **Top**: Time, in microseconds, to run the divergent analyses compared with the number of variables and instructions per kernel in GSA-form. **Bottom**: Analyses execution time ratio, given affine analysis over time of simple analysis.

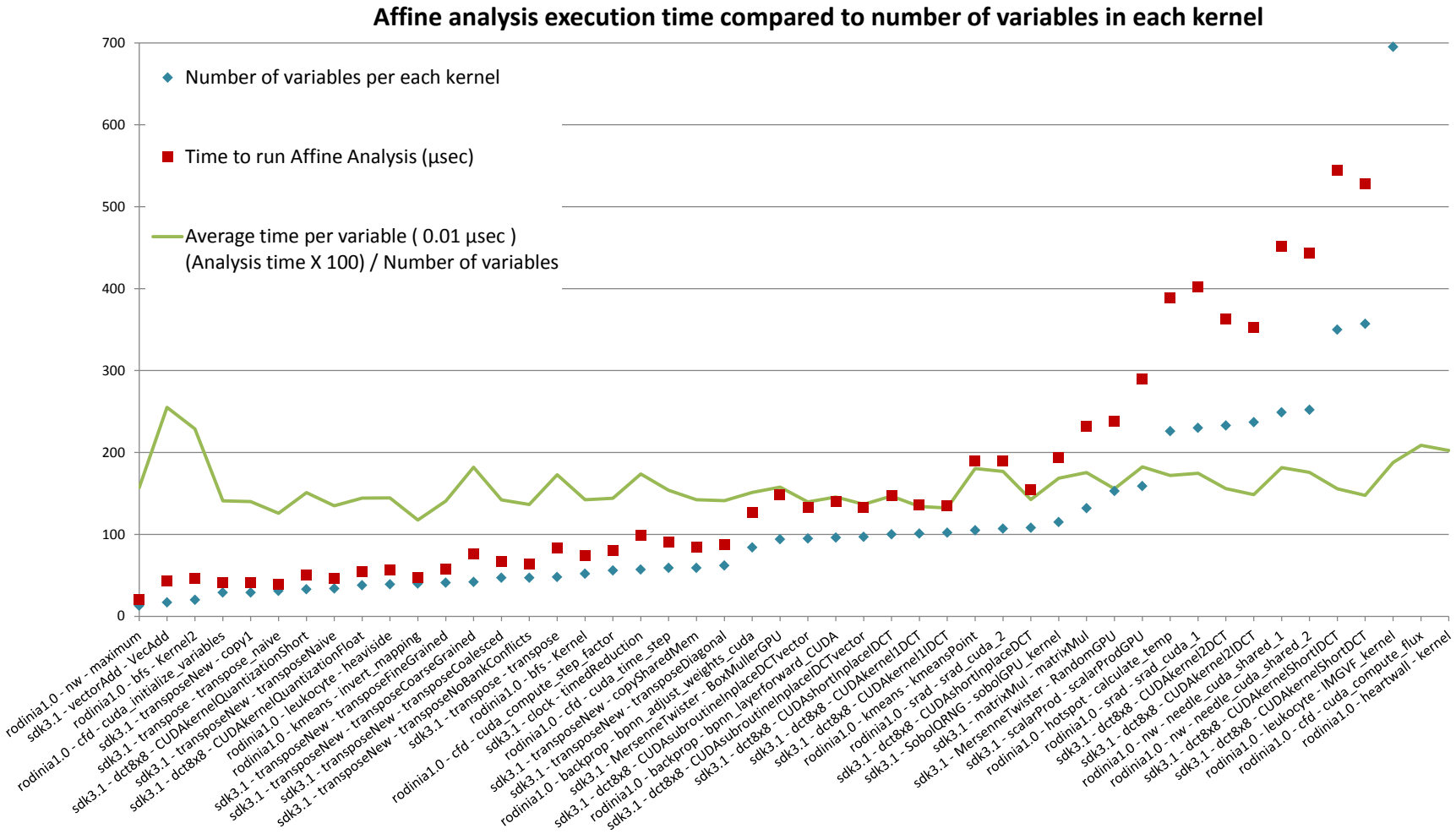


Figure 5.2. The affine analysis execution time grows linearly with the number of variables on the kernel. The two biggest kernels have their variable count and execution time out of the chart, but the ratio of analysis time per number of variables remains low.

Precision of the divergence analysis with affine constraints: Figure 5.5 compares the precision of the simple divergent analysis from Section 3.4, and the analysis with affine constraints from Section 3.5. The simple analysis reports that 63.78% of the variables are divergent, while the affine analysis gives 58.81%. However, whereas the simple divergence analysis only marks a variable as uniform or not, the affine analysis can find that a non-trivial proportion of the divergent variables are affine functions of some thread identifier. Figure 5.3: Variables distribution among abstract states inferred by the divergence analysis with affine constraints of degree two for all analyzed kernels. In that figure, we let $a_2\mathtt{T}_{id}^2 + a_1\mathtt{T}_{id} + a_0 = (a_2, a_1, a_0)$. Even though we report that 58.81% of the variables are divergent, i.e., have $a_1 \neq 0$, 24.84% of them are affine functions of some thread identifier.

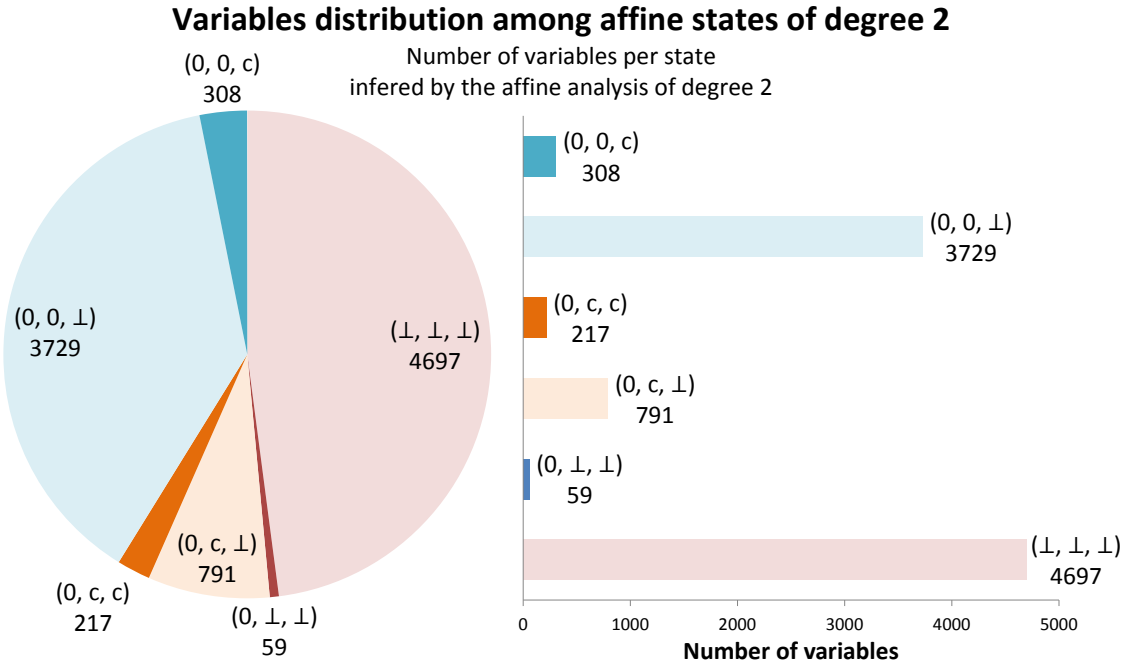


Figure 5.3. Variables count by abstract state set, as inferred by the divergence analysis with affine constraints of degree two.

Comparing different degrees of polynomials: An important question is: which polynomial degree to use in the divergence analysis with affine constraints? We have found that the affine analysis of degree two adds negligible improvement over the analysis of degree one. The latter misses 39 uniform variables that the former captures

in almost 10,000 variables. We have not found any situation in which higher degrees would improve on the second-degree analysis. Given that the run-time difference between the first and second degree analyses is insignificant, we have adopted the latter as the default in our implementation. Figure 5.4 shows the variables distribution per affine state on each of the analyzed kernels.

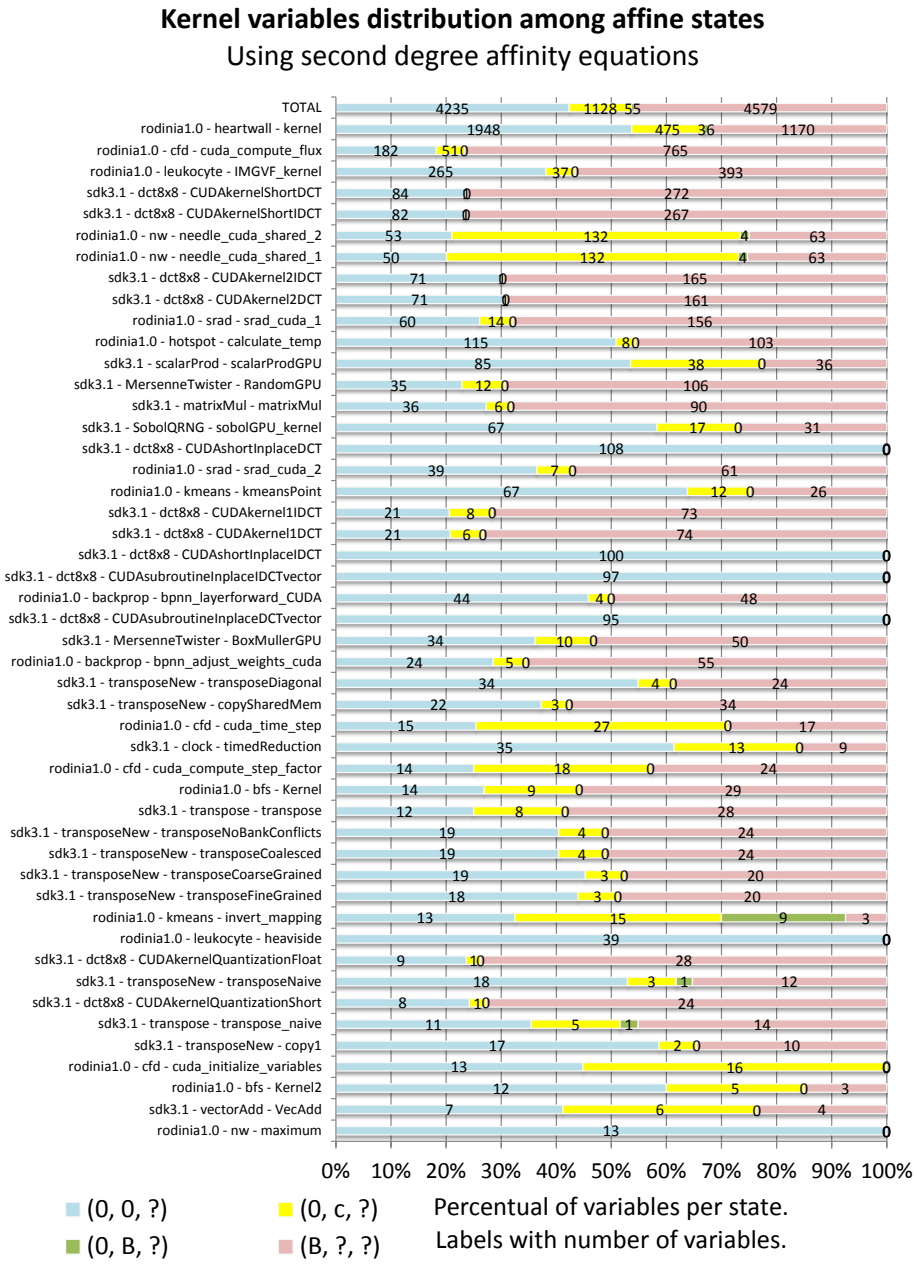


Figure 5.4. Variables affine classification per kernel. Vertical bars name kernels, horizontal bar distribution among affine states, up to 100%.

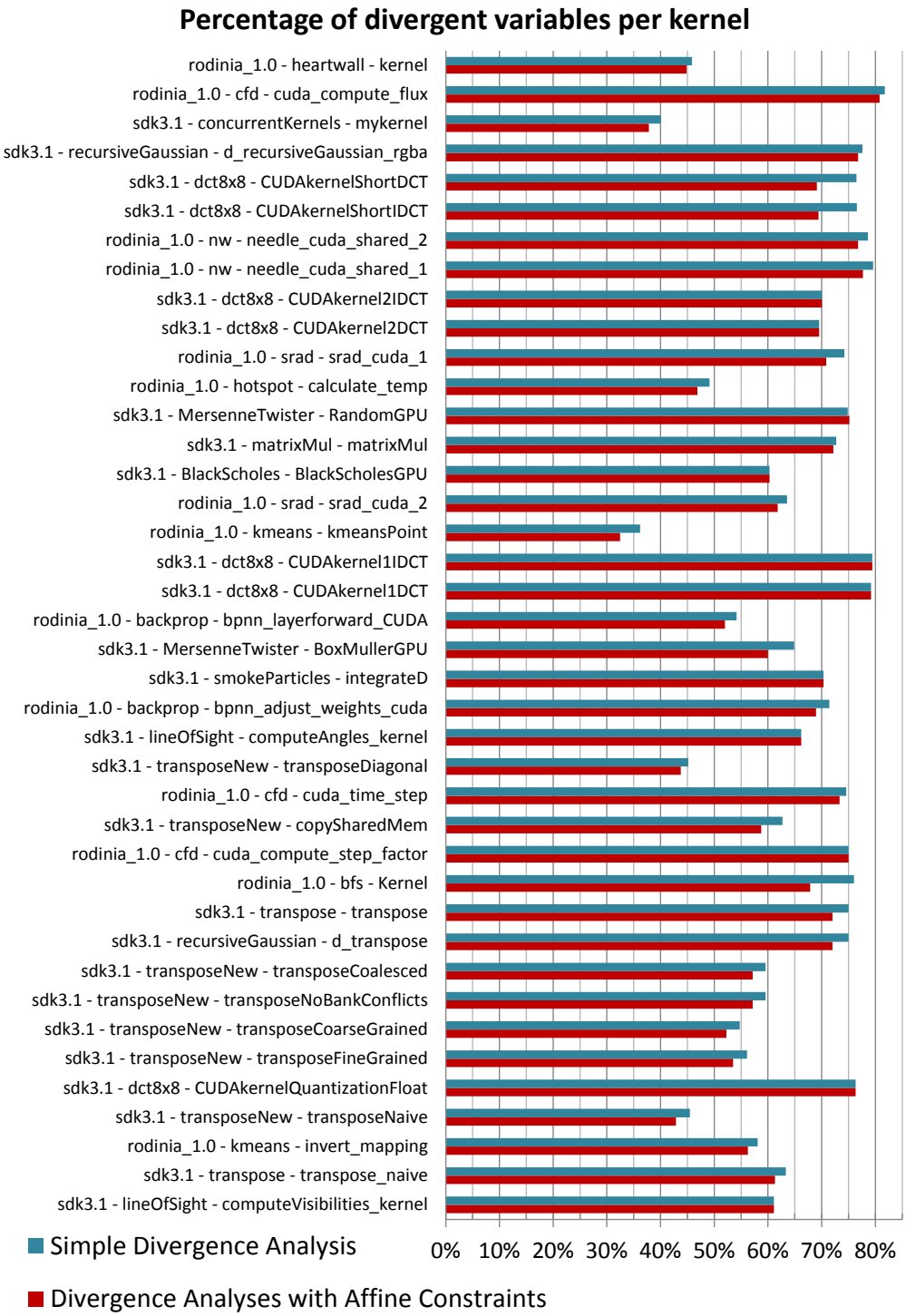


Figure 5.5. Percentage of divergent variables reported by our divergence analysis with affine constraints and the divergence analysis of Coutinho *et al.* Coutinho et al. [2011]. Kernels are sorted by the number of variables, as seen in Figure 5.1.

Register allocation: Figure 5.6 compares three execution time per kernel different implementations of divergence aware register allocators. We use, as a baseline, the linear scan register allocator Poletto and Sarkar [1999] that is publicly available in the Ocelot distribution. All the other allocators are implemented as re-written patterns that change the spill code inserted by linear scan according to the rules in Figure 4.2. All the four allocators use the same policy to assign variables to registers and to compute spilling costs. The divergence aware allocators are: **Divergence Allocator**: which moves to shared memory the variables that the simple divergence analysis of Section 3.4 marks as uniform. This allocator can only use the second rule in Figure 4.2; **Rematerialization Allocator**: Which does not use shared memory, but tries to eliminate stores and replace loads by rematerializations of spilled values that are affine functions of $\mathtt{T}_{id}$ with known constants. This allocator uses only the first and third rules in Figure 4.2; **Affine Allocator**: Which uses all the four rules in Figure 4.2 guided by the analysis of Section 3.5 with polynomials of degree two.

We report time for each kernel individually. Although kernels run in the GPU, we measure their run-time in CPU ticks, by synchronizing the start and end of each kernel call with the CPU, and recompute the time in seconds. We have run each benchmark 15 times and the variance is negligible. We take about one and a half hours to execute the 40 benchmarks 15 times on our GTX 570 GPU. Linear Scan and Rematerialization register allocators use nine registers, whereas Divergent and Affine use eight, because these two allocators must reserve one register to load the base addresses that each *warp* receives in shared memory to place spill code. Each kernel has access to 48KB of shared memory, and 16KB of cache for the local memory. In this experiments we are reserving the 16KB cache to local memory only, i.e., the kernels have been compiled with the option `-dlcm=cg`; thus, loads from global memory are not cached. On the average, all the divergence aware register allocators improve on Ocelot's original linear scan. The code produced by Rematerialization register allocator, which only improves spilling via rematerialization, is 7.31% faster than the code produced by linear scan. Divergent register allocator gives a speedup of 12.75%, and Affine register allocator gives a speedup of 29.70%. These numbers are the geometric mean over the results reported in Figure 5.6. There are situations when both, Divergent and Affine register allocators produce code that is slower than the original linear scan algorithm. This fact happens because (i) the local memory has access to a 16KB cache that is as fast as shared memory; (ii) loads and stores according to rule four of Figure 4.2 take three instructions each: a type conversion, a multiply add, and the memory access itself; and (iii) Divergent and Affine register allocators insert into the kernel some setup code to

delimit the storage area that is given to each *warp*.

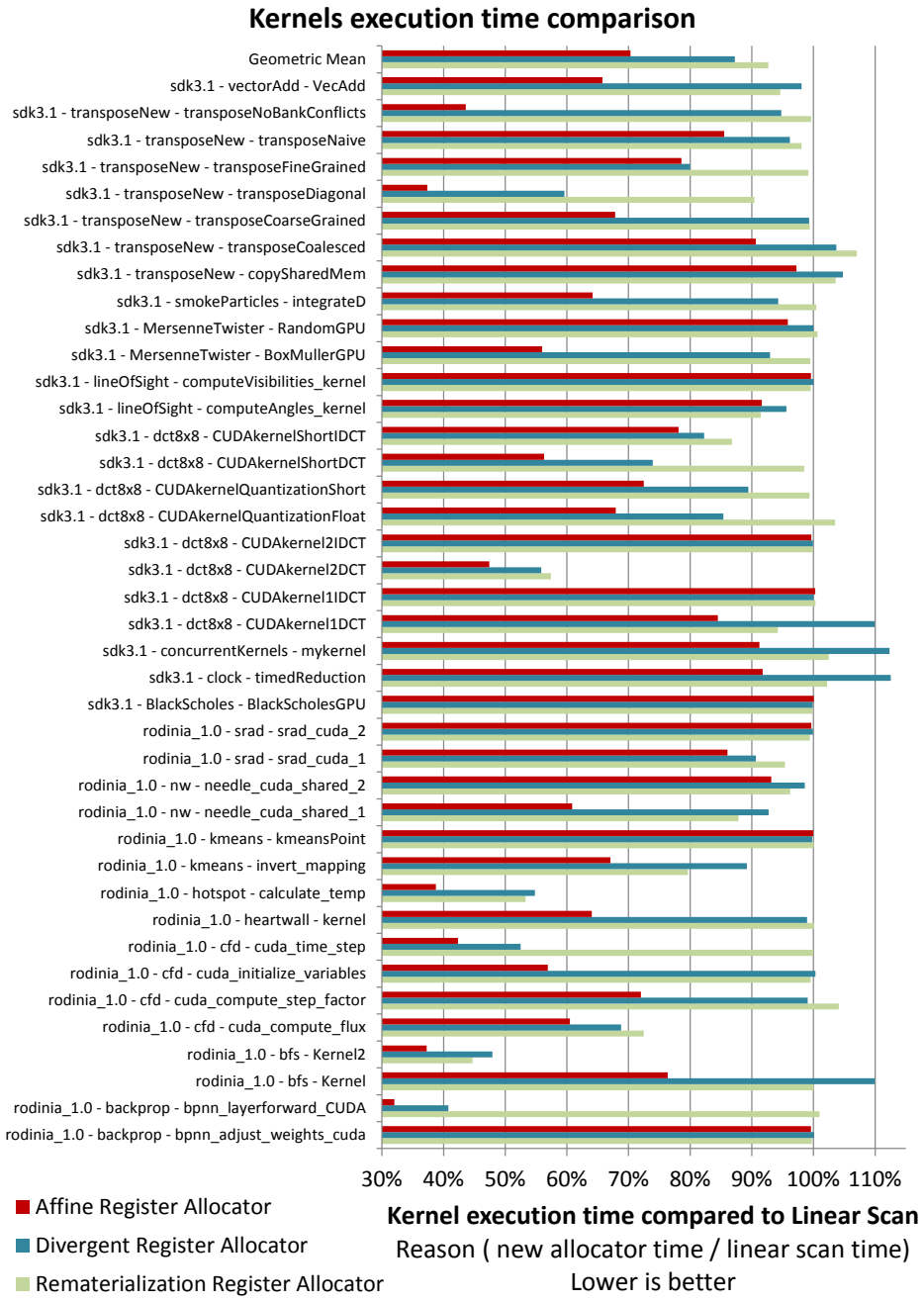


Figure 5.6. Relative speedup of different register allocators. Every bar is normalized to the time given by Ocelot's linear scan register allocator, that is, bars that are shorter than 100% represent speedups, bar that are larger represent slowdown.

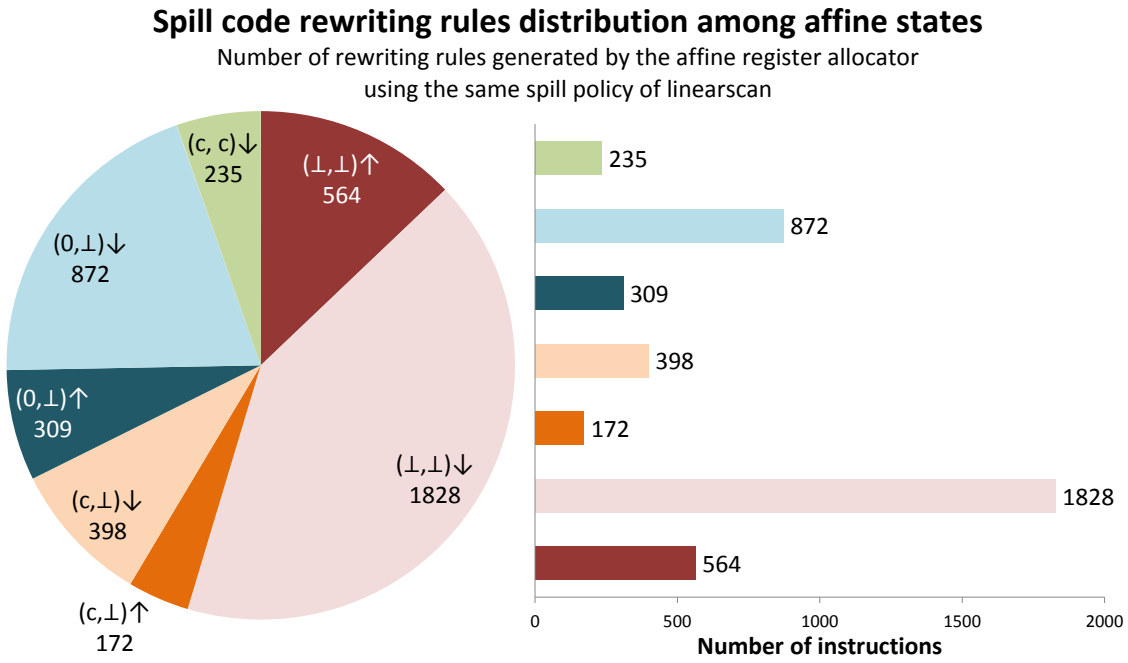


Figure 5.7. Distribution of the rewriting rules given in Figure 4.2. We omit the first coefficient of the tuples, as they are zero.

Figure 5.7 shows how many times each rewriting pattern in Figure 4.2 has been used during register allocation by Affine register allocator. We use $\Downarrow$ and $\Uparrow$ to denote loads-from and stores-to local memory; similarly, $\downarrow$ and $\uparrow$ represent loads-from and stores-to shared memory. The tuples $(0, \bot)$, (c_1, c_2) and $(c, \bot)$ refer to the second, third and fourth lines of Figure 4.2, respectively. We did not find occasion to rematerialize constants; thus, the rules in the first line of Figure 4.2 have not been used. An entry such as $(c, \bot) \downarrow$ indicates that 877 loads from local memory have been replaced by loads from shared memory plus multiply-add instructions, according to the fourth line, first column of Figure 4.2. As we see in Figure 5.7, Affine register allocator has been able to replace almost half of all the spill code with faster instruction sequences, closely following the proportion of abstract states found by the divergence analysis. Interestingly, many loop limits have the abstract state $(0, c_1, c_2), c_1, c_2 \in \mathbb{Z}$. These variables are good spill candidates, as they have long live ranges, and tend to be only used once in the program code. Both Rematerialization and Affine register allocators could take benefit from the affine analysis to rematerialize these variables whenever necessary. Figure 5.8 (iii) shows the proportion of spill code inserted by Affine register allocator in each benchmark.

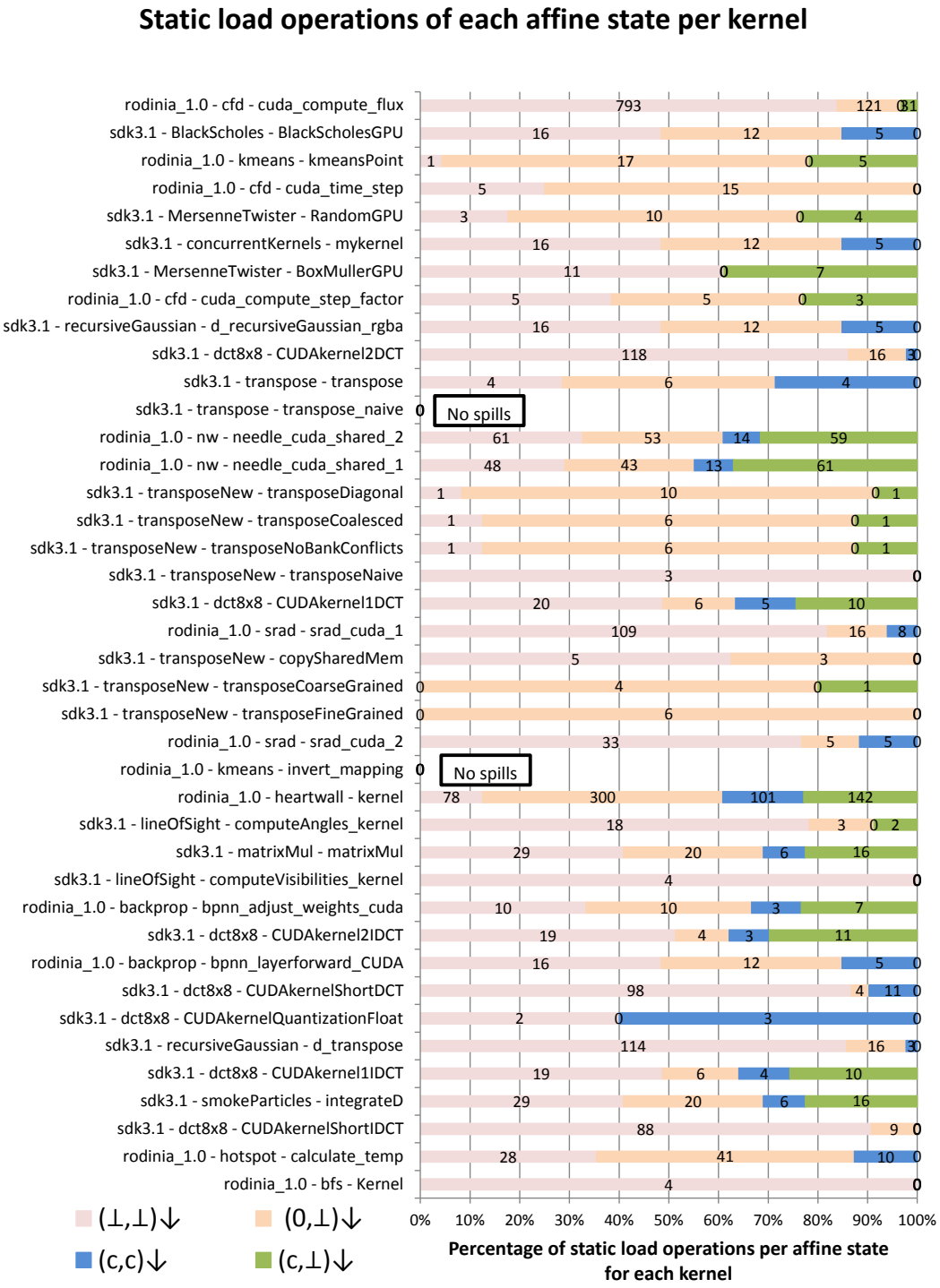


Figure 5.8. Distribution of the rewriting rules given in Figure 4.2. We omit the first coefficient of the tuples, as they are zero.

5.4 Chapter conclusion

We demonstrated that our divergence analysis and divergence aware register allocator improve the state-of-the-art code generators that target GPUs. Although we have been able to achieve very good performance numbers, our work still offers room for improvement. In the next chapter we will discuss some limitations of our allocator, and will provide pointers for future directions for this research.

Chapter 6

Conclusion

The use of GPUs for general purpose computing is still recent and there is still many possible improvements to make hardware and software more efficient. Our techniques rely on well-known methods, such as *Rematerialization* (Briggs et al. [1992]) and *constant propagation* (Callahan et al. [1986]). We have combined these classic optimizations to design what is probably the most advanced register allocator available to GPUs.

Divergence analysis helps developers and compilers to better understand the behavior of programs that execute on SIMD environments. We discussed two different implementations of this analysis. The first, seen in Section 3.4, has a simple and very efficient implementation. The second, seen in Section 3.5, is more elaborated, and provides better precision.

We demonstrated that *divergence aware register allocator* can better use GPUs computing power and memory hierarchy in a scenario where not all variables fit in register and spill code must be added. We limited our tests to a environment with NVIDIA GPUs, but we believe that these techniques will work in any SIMD-like environment with shared memory.

As a result of this research the following papers were published:

- **Spill Code Placement for SIMD Machines**
  Best paper award
  Sampaio, D. N., Gedeon, E., Pereira, F. M. Q., Collange, S.
  16th Brazilian Symposium, SBLP 2012, Natal, Brazil, September 23-28, 2012
  Programming Languages, pp 12-26, Springer Berlin Heidelberg.

- **Divergence Analysis with Affine Constraints**
  Sampaio, D. N., Martins, R., Collange, S., Pereira, F. M. Q.
  24th International Symposium on Computer Architecture and High Performance Computing, New York City, USA
  SBAC-PAD 2012, pp 67-74.

6.1 Limitations

Although our *divergence analysis with affine constraints* gives a more precise result than any other known divergence analysis we are aware of, for a SIMT environment, its precision is still far from optimal, because:

1. It only tracks variable affinity modules the `thread.Idx` identifier, considering `threadIdx.y` and `threadIdx.z` as uniform variables. Few, or none, of the applications compiled in our tests use multidimensional thread indexing, however, if that happens and the x dimension is not a multiple of the number of processing units in a *SM*, these 2 variables will diverge inside a *warp* and our analysis might classify a divergent variable as uniform. We kept only x affinity relationship due the fact that most applications do not use multidimensional indexing arrays and that we do not have access to the y and z dimensions at compile time because Ocelot works only over ptx, the intermediary code of NVIDIA's compiler.

2. At the hardware level, a processing element has an identifier inside a *warp*, the `%laneid` ($\mathtt{l}_{id}$) variable. We do not track variables affinity upon this identifier, and consider it as a divergent variable.

3. The analysis does not take into consideration neither the bounds that types impose on variables, nor the behavior of instructions in face of exceptional conditions. Behavior modifications are important to understand the instructions behavior upon abnormal situations, such as overflows. As an example, a multiply instruction of constants could overflow the resulting variable capacity. If the instruction modifier tells so, only the lower valued bits are kept. This is commonly used when generating bit masks. As our analysis do not identify variable limits it might rematerialize wrong values.

4. `Ocelot` is limited do read and write *ptx* code, the intermediary language used by NVIDIA's GPUs. In practice, this code is further compiled into GPU binary code before being loaded, and this JIT compiler may change the code that we

generate. If that happens, our register allocation may suffer some modifications before our code makes its way to the hardware.

6.2 Future work

To solve the problem of incorrectly naming $\mathtt{T}_{id}$ indexes y and z as uniform we propose a code specialization, where optimal code is generated for each possible case, whenever these indexes are used. Branch tests must be inserted to decide at run-time which code to run. The problem with this solution is that the program size can be, in a worst case scenario, multiplied by 3 (possible cases where both y and z are uniform, divergent and where y is divergent and z uniform). However, if at compile time we knew all values uses for block indexes, we can treat each kernel call differently. If we want to prevent the usage of a shared memory position to store affine values related to these indexes it is also required to track dependency upon these indexes, what could increase significantly the analysis run-time and *rematerialization* cost when a variable depends on more than one of these indexes. A variable v would now be written as the equation $v = a_1x + a_2y + a_3z + b$ if we opt for a first degree affinity, and in a worse case scenario a rematerialization would require 4 shared memory positions and 7 instructions to store and load.

Although tracking y and z indexes might add many operations, it is not common to find variables that depend simultaneously upon the $\mathtt{T}_{id}$ and the $\mathtt{l}_{id}$. This is an empirical observation verified in all benchmarks used in this dissertation. In such case, a variable v defined as $v = a_1\mathtt{T}_{id} + a_2\mathtt{l}_{id} + b$ will always have either a_1 or a_2 equal to zero, and the rematerialization cost would not increase.

To solve the problem of wrong rematerialization values, *range analysis* (see Cousot and Cousot [1977]) can be combined with our divergence analysis, and variables type and behavior modifications can be reproduced by the analysis, what would prevent from incorrect values defining. This limitation can further improve our analysis, as we can consider bit shifts and division instructions, as well as analyzing floating point values instead of integer only.

Finally, to solve the problem of requiring a third party compiler after doing our optimizations we would require to write a compiler back-end that produced binary code for the GPU, a hard task as the GPU ISAs are proprietary and undocumented.

A possible move that would help on every of these future works would be to use `LLVM` instead of `Ocelot`, as the official GPGPU NVIDIA compiler uses it, and allows some

access to the intermediary code.

6.3 Final thoughts

This dissertation has made important advances in the suite of techniques that compilers use to generate code for GPUs. We hope that these advances will be even more important in the coming years, as the popularity of GPUs as an alternative for high performance computing is likely to continue.

Bibliography

Abel, N. E., Budnik, P. P., Kuck, D. J., Muraoka, Y., Northcote, R. S., and Wilhelmson, R. B. (1969). Tranquil: a language for an array processing computer. In *Proceedings of the May 14-16, 1969, spring joint computer conference*, AFIPS '69 (Spring), pages 57--73, New York, NY, USA. ACM.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Aiken, A. and Gay, D. (1998). Barrier inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 342--354, New York, NY, USA. ACM.

Appel, A. W. (1998). Ssa is functional programming. *SIGPLAN Not.*, 33(4):17--20.

Baghsorkhi, S. S., Delahaye, M., Patel, S. J., Gropp, W. D., and Hwu, W.-m. W. (2010). An adaptive performance modeling tool for gpu architectures. *SIGPLAN Not.*, 45(5):105--114.

Blelloch, G. and Chatterjee, S. (1990). Vcode: A data-parallel intermediate language. In *In Proceedings of the 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 471--480.

Boudier and Sellers (2011). Memory system on Fusion APUs.

Bougé, L. and Levaire, J.-L. (1992). Control structures for data-parallel simd languages: semantics and implementation. *Future Gener. Comput. Syst.*, 8(4):363--378.

Bouknight, Denenberg, McIntyre, Randall, Sameh, and Slotnick (1972). The Illiac IV system. In *Proceedings of the IEEE*, volume 60, pages 369 – 388. IEEE.

Briggs, P., Cooper, K. D., and Torczon, L. (1992). Rematerialization. In Feldman, S. I. and Wexelblat, R. L., editors, *PLDI*, pages 311–321. ACM.

Brockmann, K. and Wanka, R. (1997). Efficient oblivious parallel sorting on the maspar mp-1. In *Proceedings of the 30th Hawaii International Conference on System Sciences: Software Technology and Architecture - Volume 1*, HICSS '97, pages 200--, Washington, DC, USA. IEEE Computer Society.

Budimlic, Z., Cooper, K. D., Harvey, T. J., Kennedy, K., Oberg, T. S., and Reeves, S. W. (2002). Fast copy coalescing and live-range identification. *SIGPLAN Not.*, 37(5):25--32.

Callahan, D., Cooper, K. D., Kennedy, K., and Torczon, L. (1986). Interprocedural constant propagation. *SIGPLAN Not.*, 21(7):152--161.

Carrillo, S., Siegel, J., and Li, X. (2009). A control-structure splitting optimization for gpgpu. In *Proceedings of the 6th ACM conference on Computing frontiers*, CF '09, pages 147--150, New York, NY, USA. ACM.

Cederman, D. and Tsigas, P. (2010). Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4--4:1.24.

Chaitin, G. J. (1982). Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 98--105, New York, NY, USA. ACM.

Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44--54, Washington, DC, USA. IEEE Computer Society.

Choi, J.-D., Cytron, R., and Ferrante, J. (1991). Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 55--66, New York, NY, USA. ACM.

Collange, S., Defour, D., and Zhang, Y. (2010). Dynamic detection of uniform and affine vectors in gpgpu computations. In *Proceedings of the 2009 international conference on Parallel processing*, Euro-Par'09, pages 46--55, Berlin, Heidelberg. Springer-Verlag.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238--252. ACM.

Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Meira Jr., W. (2010). Performance debugging of gpgpu applications with the divergence map. In *Proceedings of the 2010 22nd International Symposium on Computer Architecture and High Performance Computing*, SBAC-PAD '10, pages 33--40, Washington, DC, USA. IEEE Computer Society.

Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Meira Jr., W. (2011). Divergence analysis and optimizations. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 320--329, Washington, DC, USA. IEEE Computer Society.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451--490.

Darema, F., George, D. A., Norton, V. A., and Pfister, G. F. (1988). A single-program-multiple-data computational model for epex/fortran. *Parallel Computing*, pages 11–24.

Diamos, G., Ashbaugh, B., Maiyuran, S., Kerr, A., Wu, H., and Yalamanchili, S. (2011). Simd re-convergence at thread frontiers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 477--488, New York, NY, USA. ACM.

Diamos, G. F., Kerr, A. R., Yalamanchili, S., and Clark, N. (2010). Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353--364, New York, NY, USA. ACM.

Farrell, C. A. and Kieronska, D. H. (1996). Formal specification of parallel simd execution. *Theor. Comput. Sci.*, 169(1):39--65.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987). The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319--349.

Flynn, M. J. (1972). Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948--960.

Fung, W. W. L., Sham, I., Yuan, G., and Aamodt, T. M. (2007). Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 407--420, Washington, DC, USA. IEEE Computer Society.

Garland, M. and Kirk, D. B. (2010). Understanding throughput-oriented architectures. *Commun. ACM*, 53(11):58--66.

Habermaier, A. and Knapp, A. (2012). On the correctness of the simt execution model of gpus. In *ESOP*, pages 316–335.

Hack, S. and Goos, G. (2006). Optimal register allocation for ssa-form programs in polynomial time. *Inf. Process. Lett.*, 98(4):150--155.

Han, T. D. and Abdelrahman, T. S. (2011). Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 3:1--3:8, New York, NY, USA. ACM.

Hennessy, J. L. and Patterson, D. A. (2006). *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Hillis, W. D. and Steele, Jr., G. L. (1986). Data parallel algorithms. *Commun. ACM*, 29(12):1170--1183.

Hoogvorst, P., Keryell, R., Matherat, P., and Paris, N. (1991). Pomp or how to design a massively parallel machine with small developments. In *PARLE (1)*, pages 83–100.

Jang, B., Schaa, D., Mistry, P., and Kaeli, D. (2010). Static memory access pattern analysis on a massively parallel gpu.

Karrenberg, R. and Hack, S. (2011). Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 141--150, Washington, DC, USA. IEEE Computer Society.

Kerr, A., Diamos, G., and Yalamanchili, S. (2012). Dynamic compilation of data-parallel kernels for vector processors. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 23--32, New York, NY, USA. ACM.

Khronos, G. (2011). *The OpenCL Specification*.

Kung, S.-Y., Arun, K. S., Gal-Ezer, R. J., and Bhaskar Rao, D. V. (1982). Wavefront array processor: Language, architecture, and applications. *IEEE Trans. Comput.*, 31(11):1054--1066.

Lashgar, A. and Baniasadi, A. (2011). Performance in gpu architectures: Potentials and distances. In *9th Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD11), in conjunction with ISCA-38*, pages 75--81.

Lawrie, D. H., Layman, T., Baer, D., and Randal, J. M. (1975). Glypnira programming language for illiac iv. *Commun. ACM*, 18(3):157--164.

Lee, S., Min, S.-J., and Eigenmann, R. (2009). Openmp to gpgpu: a compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101--110.

Lee, Y., Avizienis, R., Bishara, A., Xia, R., Lockhart, D., Batten, C., and Asanović, K. (2011). Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. *SIGARCH Comput. Archit. News*, 39(3):129--140.

Leissa, R., Hack, S., and Wald, I. (2012). Extending a c-like language for portable simd programming. *SIGPLAN Not.*, 47(8):65--74.

Meng, J., Tarjan, D., and Skadron, K. (2010). Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, 38(3):235--246.

Mu, S., Zhang, X., Zhang, N., Lu, J., Deng, Y. S., and Zhang, S. (2010). Ip routing processing with graphic processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 93--98, 3001 Leuven, Belgium, Belgium. European Design and Automation Association.

Narasiman, V., Shebanow, M., Lee, C. J., Miftakhutdinov, R., Mutlu, O., and Patt, Y. N. (2011). Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 308--317, New York, NY, USA. ACM.

Nickolls, J. and Dally, W. J. (2010). The gpu computing era. *IEEE Micro*, 30(2):56--69.

Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

NVIDIA, C. (2012). *NVIDIA CUDA C - Programming Guide*.

Ottenstein, K. J., Ballance, R. A., and MacCabe, A. B. (1990). The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. *SIGPLAN Not.*, 25(6):257--271.

Patterson, D. A. and Hennessy, J. L. (2012). *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press.

Perrott, R. H. (1979). A language for array and vector processors. *ACM Trans. Program. Lang. Syst.*, 1(2):177--195.

Pharr, M. and Mark, W. (2012). ispc: A spmd compiler for high-performance cpu programming. *Proceedings of Innovative Parallel Computing (InPar)*.

Poletto, M. and Sarkar, V. (1999). Linear scan register allocation. *ACM Trans. Program. Lang. Syst.*, 21(5):895--913.

Prabhu, T., Ramalingam, S., Might, M., and Hall, M. (2011). Eigencfa: accelerating flow analysis with gpus. *SIGPLAN Not.*, 46(1):511--522.

Rogers, T. G., O'Connor, M., and Aamodt, T. M. (2012). Cache-conscious wavefront scheduling. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45 '12, New York, NY, USA. ACM.

Ryoo, S., Rodrigues, C. I., Baghsorkhi, S. S., Stone, S. S., Kirk, D. B., and Hwu, W.-m. W. (2008). Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73--82, New York, NY, USA. ACM.

Saha, B., Zhou, X., Chen, H., Gao, Y., Yan, S., Rajagopalan, M., Fang, J., Zhang, P., Ronen, R., and Mendelson, A. (2009). Programming model for a heterogeneous x86 platform. *SIGPLAN Not.*, 44(6):431--440.

Samadi, M., Hormati, A., Mehrara, M., Lee, J., and Mahlke, S. (2012). Adaptive input-aware compilation for graphics engines. *SIGPLAN Not.*, 47(6):13--22.

Sandes, E. F. O. and de Melo, A. C. M. (2010). Cudalign: using gpu to accelerate the comparison of megabase genomic sequences. *SIGPLAN Not.*, 45(5):137--146.

Stratton, J. A., Grover, V., Marathe, J., Aarts, B., Murphy, M., Hu, Z., and Hwu, W.-m. W. (2010). Efficient compilation of fine-grained spmd-threaded programs for

multicore cpus. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 111--119, New York, NY, USA. ACM.

TOP500, p. (2012). List november 2012.

Tu, P. and Padua, D. (1995). Efficient building and placing of gating functions. *SIGPLAN Not.*, 30(6):47--55.

Weiser, M. (1981). Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439--449, Piscataway, NJ, USA. IEEE Press.

Yang, Y., Xiang, P., Kong, J., and Zhou, H. (2010). A gpgpu compiler for memory optimization and parallelism management. *SIGPLAN Not.*, 45(6):86--97.

Zhang, E. Z., Jiang, Y., Guo, Z., and Shen, X. (2010). Streamlining gpu applications on the fly: thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 115--126, New York, NY, USA. ACM.

Zhang, E. Z., Jiang, Y., Guo, Z., Tian, K., and Shen, X. (2011). On-the-fly elimination of dynamic irregularities for gpu computing. *SIGARCH Comput. Archit. News*, 39(1):369--380.

Zhang, Y. and Owens, J. D. (2011). A quantitative performance analysis model for gpu architectures. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, HPCA '11, pages 382--393, Washington, DC, USA. IEEE Computer Society.