Divergence Analysis

DIOGO SAMPAIO and RAFAEL MARTINS DE SOUZA, Universidade Federal de Minas Gerais
CAROLINE COLLANGE, INRIA
FERNANDO MAGNO QUINTÃO PEREIRA, Universidade Federal de Minas Gerais

Growing interest in graphics processing units has brought renewed attention to the Single Instruction Multiple Data (SIMD) execution model. SIMD machines give application developers tremendous computational power; however, programming them is still 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 two processing elements (PEs) see the same variable name holding different values. This article introduces divergence analysis, a static analysis that discovers data divergences. This analysis, currently deployed in an industrial quality compiler, is useful in several ways: it improves the translation of SIMD code to non-SIMD CPUs, it helps developers to manually improve their SIMD applications, and it also guides the automatic optimization of SIMD programs. We demonstrate this last point by introducing the notion of a divergence-aware register spiller. This spiller uses information from our analysis to either rematerialize or share common data between PEs. As a testimony of its effectiveness, we have tested it on a suite of 395 CUDA kernels from well-known benchmarks. The divergence-aware spiller produces GPU code that is 26.21% faster than the code produced by the register allocator used in the baseline compiler.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors

General Terms: Languages, Design, Algorithms, Performance

Additional Key Words and Phrases: Static program analysis, divergence analysis, SIMD, graphics processing units, high performance

ACM Reference Format:
Sampaio, D., de Souza, R. M., Collange, C., and Pereira, F. M. Q. 2013. Divergence analysis. ACM Trans. Program. Lang. Syst. 35, 4, Article 13 (December 2013), 36 pages.
DOI: http://dx.doi.org/10.1145/2523815

1. INTRODUCTION

Increasing programmability and low hardware cost are boosting the use of graphical processing units (GPU) as a tool to run general-purpose applications. Illustrative examples of this new trend are the rising popularity of CUDA[1] AMD APP, [2] and OpenCL.[3] Running general-purpose programs in GPUs is attractive, because these processors are

[1]See *The CUDA Programming Guide, 1.1.1*.
[2]See *AMD APP Guide*.
[3]See *The OpenCL Specification, 1.0*.

This work is supported by FAPEMIG grant 2010/2. D. Sampaio has been supported by the Brazilian Research Council (CNPq). C. Collange was supported by In Web during her stay in Brazil.
Authors' addresses: D. Sampaio and F. Pereira, Computer Science Department, The Federal University of Minas Gerais, Antônio Carlos Avenue, 6613, Belo Horizonte, Brazil; C. Collange, INRIA Centre Rennes - Bretagne Atlantique, Campus de Beaulieu, 35042 Rennes Cedex, France; F. M. Q. Peieira (corresponding author): pronesto@gmail.com.

DOI: http://dx.doi.org/10.1145/2523815

massively parallel. As an example, the GeForce GTX 580 GPU series has 512 processing units that can be simultaneously used by up to 24,576 threads. Similar hardware has allowed the development of high-performance solutions to problems as diverse as sorting [Cederman and Tsigas 2009], gene sequencing [Sandes and de Melo 2010], IP routing [Mu et al. 2010], and program analysis [Prabhu et al. 2011]. These applications might outperform the equivalent CPU program by factors of over 100× [Ryoo et al. 2008]. This trend is likely to continue, as upcoming hardware more closely integrates GPUs and CPUs [Boudier and Sellers 2011], new models of heterogeneous hardware are introduced [Lee et al. 2011; Saha et al. 2009], and novel programming abstractions are developed for them [Cunningham et al. 2011; Dubach et al. 2012].

GPUs are highly parallel; however, due to their restrictive programming model, not every application can benefit from this parallelism. These processors organize threads in groups that execute in lock-step. Such groups are called *warps* in NVIDIA's jargon, or *wavefronts* in AMD's. To understand the rules that govern threads in the same warp, we can imagine that each warp has simultaneous access to many processing units but uses only one instruction fetcher. As an example, the GeForce GTX 590 has 32 streaming multiprocessors, and each of them can run 48 warps of 32 threads. Thus, each warp might execute 32 instances of the same instruction simultaneously. Regular applications, such as scalar vector multiplication, fare very well in GPUs, as we have the same operation being independently performed on different chunks of data. However, divergences may happen in less regular applications.

Data divergence occurs if the same variable name is mapped to different values in the environments of distinct processing elements. In this case, we say that the variable is *divergent*, otherwise we call it *uniform*. A thread identifier, for instance, is inherently divergent. Data divergence is responsible for two phenomena that can compromise performance: *memory* and *control-flow divergences*. Control-flow divergences happen when threads in a warp follow different paths after processing the same branch. If the branching condition is data divergent, then it might be true to some threads and false to others. Given that each warp has access to only one instruction at a time, some threads have to wait idly, while others execute. Memory divergences, a term coined by Meng et al. [2010], happen whenever a load or store instruction targeting data-divergent addresses causes threads to access memory positions with bad locality. Such events have been shown to have even more performance impact than control-flow divergences [Lashgar and Baniasadi 2011]. Optimizing an application to avoid divergences 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.

The main goal of this article is to provide compilers with techniques that help them understand and improve divergent code. To meet such an objective, in Section 3.3, we present a static program analysis that identifies data divergences. We then expand this analysis, discussing in Section 3.4 a more advanced algorithm that distinguishes divergent and affine variables, for example, variables that are affine expressions of thread identifiers. The two analyses that we discuss in this article rely on the classic notion of Gated Static Single Assignment form [Ottenstein et al. 1990; Tu and Padua 1995], which we revisit in Section 3.2. We formalize our algorithms by proving their correctness with regard to μ-SIMD, a core language that we describe in Section 3.1.

The divergence analysis is important in different ways. First, it helps the compiler to optimize the translation of SIMD languages to ordinary CPUs. We call SIMD languages those programming languages, such as C for CUDA and OpenCL, that are equipped with abstractions to handle divergences. Currently, there exist many

proposals to compile such languages to ordinary CPUs [Diamos et al. 2010; Karrenberg and Hack 2011; Stratton et al. 2010], and they all face similar difficulties. Vectorial operations found in traditional architectures, such as the x86's SSE extension, do not support divergences natively. Thus, compilers need to produce very inefficient code to handle this phenomenon at the software level. This burden can be safely removed from the uniform, for example, nondivergent, branches that we identify. Furthermore, the divergence analysis provides insights about memory access patterns [Jang et al. 2010]. In particular, a uniform address means that 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 generating efficient code for vectorial instruction sets that do not support fast memory gather and scatter [Diamos et al. 2010].

Second, in order to more precisely identify divergences, a common strategy is to use instrumentation-based profilers. However, this approach may slow down the target program by factors of over 1500× [Coutinho et al. 2013]! Our divergence analysis reduces the number of branches that the profiler must instrument, hence decreasing its overhead. Third, the divergence analysis improves the static performance prediction techniques used in SIMD architectures [Baghsorkhi et al. 2010; Zhang and Owens 2011]. Such methods are used, for instance, by adaptive compilers that target GPUs [Samadi et al. 2012]. 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.'s [2011] *branch fusion* and Zhang et al.'s [2011] thread reallocation strategy. In this article, we augment this family of techniques with a *divergence-aware register spiller*. As we will show in Section 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 a form of *rematerialization* [Briggs et al. 1992] of values among SIMD processing elements.

All the algorithms that we describe in this article are publicly available in the Ocelot compiler [Diamos et al. 2010]. This implementation consists of over 10,000 lines of open-source code. Ocelot optimizes PTX, the intermediate program representation used by NVIDIA's GPUs. We have compiled all 395 CUDA kernels taken from the Rodinia [Che et al. 2009], Parboil [Stratton et al. 2012], and 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 43.98% of the variables we have found in our benchmarks are uniform. The affine constraints from Section 3.4 increase this number by 2%, and, more importantly, they indicate that about one fifth, that is, 20.70%, of the divergent variables are affine functions of some thread identifier. Finally, our divergence-aware register spiller is effective: by rematerializing affine values or moving uniform values to the GPU's shared memory, we have been able to speed up the code produced by Ocelot's original register allocator by 26.20%.

This article closes our three years of work in divergence analysis for SIMD architectures. Our first publication in this field [Coutinho et al. 2011] introduced the divergence analysis that we discuss in Section 3.3. At that time, we chose to describe this static analysis as an instance of the more general graph reachability problem, following an earlier approach adopted by Scholz et al. [2008] to detect tainted-flow vulnerabilities in programs. Presently, we have opted to depart from the graph reachability framework in favor of a constraint oriented-notation, because, as we see in Section 3.3, this new notation simplifies our correctness proofs. The extended divergence analysis from Section 3.4 was presented in late 2012 [Sampaio et al.

2012b]. In that work, we mentioned our divergence-aware register spiller; however, in this article, we explain it in much deeper detail, following a previous description given at the Brazilian Symposium on Programming Languages [Sampaio et al. 2012a].

2. BACKGROUND

A modern graphics processing unit usually provides developers with a large number of threads arranged in small groups called warps. Different warps execute independently of each other, following Darema's Single Program Multiple Data (SPMD) execution model [Darema et al. 1988]. On the other hand, the threads inside the same warp execute in lock-step, fitting Flynn's Single Instruction Multiple Data (SIMD) machines [Flynn 1972]. This combination of SPMD and SIMD semantics is one of the characteristics of the so-called Single Instruction Multiple Threads (SIMT) execution model [Garland and Kirk 2010; Nickolls and Kirk 2009; Nickolls and Dally 2010]. In this article, we will focus on the SIMD characteristics of a typical GPU, because divergences are relevant only at this level.

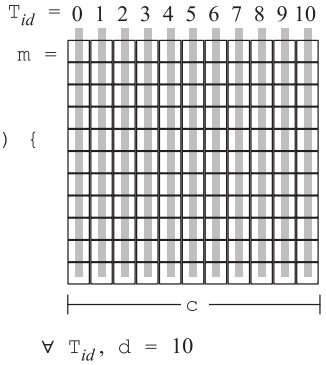
We will use the two artificial programs in Figure 1 to explain the notion of divergences. These functions, normally called *kernels*, are written in C for CUDA, and run on graphics processing units. We will assume that these programs are executed by a number of threads, or processing elements, according to the SIMD semantics. All the processing elements see the same set of variable names; however, each one maps this environment onto a different address space. Furthermore, each processing element has a particular set of identifiers. In C for CUDA, this set includes the index of the thread in three different dimensions, for example, `threadIdx.x`, `threadIdx.y`, and `threadIdx.z`. At the hardware level, a processing element has access to more identifiers, such as its position inside the warp (`%laneid`). For this discussion, just the understanding that a thread has a unique identifier is enough. In the rest of this article, we will denote this unique thread identifier by $\mathtt{T}_{id}$.

Each processing element uses its unique identifier to find the data that it must process. Thus, in the kernel `avgSquare`, each thread $\mathtt{T}_{id}$ is in charge of summing up the elements of the $\mathtt{T}_{id}$-th column of `m`. Once leaving the loop, this PE will store the average of the sum in $\mathtt{v}[\mathtt{T}_{id}]$. This is a divergent memory access: different addresses will be simultaneously accessed by many threads. 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 divergences 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 indices of a column contribute to the sum. In this case, the threads perform different amounts of work: the PE 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.

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

```
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}
```

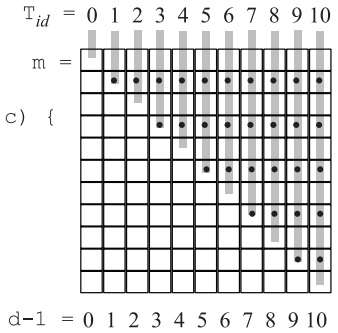


Fig. 1. Two kernels written in C for CUDA. The gray lines on the right show the parts of matrix `m` processed by each thread. Following usual coding practices, we represent the matrix in a linear format. Dots mark the cells that add up to the sum in Line 8 of `sumTriangle`.

exclusively with control-flow divergences are the most common in the literature. As an example, the PTX programmer's manual[4] recommends replacing ordinary branch instructions (`bra`) proved to be nondivergent with 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*.

Optimizing Divergent Control Flow. Branch distribution [Han and Abdelrahman 2011] is a form of code hoisting that works both at the prologue and at the epilogue of a branch. This optimization merges code inside potentially divergent program paths. Branch fusion [Coutinho et al. 2011], a generalization of branch distribution, joins chains of common instructions present in two divergent paths. A number of compiler optimizations try to rearrange loops in order to mitigate the impact of divergences. Carrillo et al. [2009] have proposed branch splitting, a way to divide a parallelizable loop

[4]PTX programmer's manual, 2008-10-17, SP-03483-001_v1.3, ISA 1.3.

enclosing a multipath branch into multiple loops, each containing only one branch. Lee et al. [2009] have designed loop collapsing, a compiler technique that they use to reduce divergences inside loops when compiling OpenMP programs into C for CUDA. Later, 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 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. 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]. This optimization must be used with moderation, because Lashgar and Baniasadi [2011, Section 4.A] have shown that unrestrained thread regrouping could lead to memory divergences.

Optimizing Divergent Memory Accesses. The compiler-related literature describes optimizations that try to change memory access patterns in such a way as to improve address locality. Recently, some of these techniques have been adapted to mitigate the impact of memory divergences 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 of 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 and that mitigate the impact of data divergences.

Reducing Redundant SIMD Work. The literature describes a few optimizations that use divergence information to reduce the amount of work that the SIMD processing elements do. For instance, Collange et al. [2009] introduced *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[5] and AMD GCN[6], combine scalar and vector processing units. Capitalizing on this observation, a recent work by Lee et al. [2013] uses divergence analysis to assign computations to either scalar or vector processing units.

A Comparison between Previous Divergence Analyses and Our Approaches. Several algorithms have been proposed in the literature to find uniform variables. It is also generally assumed that industrial compilers, like AMD's or Nvidia's, implement some sort of divergence analysis, such as Grover's algorithm [Grover et al. 2009]. As an example, the AMD GCN compiler is able to target scalar units with uniform instructions. Furthermore, these uniform instructions can use scalar registers instead of shared memory, a capacity that would require techniques similar to those we describe in Section 4. Nevertheless, such industrial solutions are not open to the public. 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[7] 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.

Recent interest in graphics processing units has given a renewed impulse to this type of analysis, in particular with a focus on SIMD machines. The first description of a divergence analysis targeting the execution model of a GPU that we are aware of is attributed to Stratton et al. [2010], who called it variance analysis. The description of Stratton's et al.'s work is too brief to allows us to compare it with our techniques, but

[5]See *Intel pushes for HPC space with Knights Corner*. hhttp://www.thinq.co.uk. Last accessed June 2012.
[6]See *Understanding AMD's Roadmap.* http://www.anandtech.com/. Last accessed June 2012.
[7]Aiken and Gay would call these variables *single-valued*.

an extended version of variance analysis appears in a patent application by Grover et al. [2009]. From the patent description, we infer that variance analysis is similar to our divergence analysis from Section 3.3, except that it does not distinguish different abstract states of variables inside and outside loops. We obtain this distinction from our intermediate representation, the gated static single assignment form, which splits live ranges of variables that escape loops. The variance analysis has been further expanded by Lee et al. [2013], who proposed using it to separate scalar and vector operations in an SIMD program. Lee et al. mention the possibility of combining their variance analysis with Collange's affine analysis [Collange et al. 2009] to optimize memory accesses. However, the single-paragraph description of their approach [Lee et al. 2013, Section 3.5] does not give us enough details to compare it with our algorithm from Section 3.4.

Another variation of divergence analysis has been recently proposed by researchers from Saarland University: *vectorization analysis* [Karrenberg and Hack 2011]. Vectorization analysis can track some affine relations between variables in the SIMD execution model. In particular, it can identify which variables hold values that are consecutively spaced between successive threads. Yet, contrary to our approach, the vectorization analysis does not take control-flow dependences into consideration when determining the abstract state of variables. This omission is not a problem in their scenario, because the vectorization analysis is a technique used in the compilation of SPMD programs to CPUs with explicit SIMD instructions. Their compiler generates specific instructions to manage divergences at runtime. However, a naïve application of Karrenberg's analyses in our static context may wrongly report that a divergent variable is uniform due to control dependences. As an example of this behavior, Karrenberg's select and loop-blending functions are similar to the γ and η functions that we discuss in Section 3.2. Nevertheless, select and blend are concrete instructions emitted during code generation, whereas our GSA functions are abstractions used statically. Karrenberg and Hack have recently proposed their version of divergence analysis [Karrenberg and Hack 2012]. We believe that their algorithm is equivalent to the design of Grover Grover et al. [2009] and Lee et al. [2013]. These three analyses mark variable d as divergent inside the two loops of Figure 1, whereas any of our analyses sets it as uniform. As a consequence, our divergence-aware spiller of Section 4 can spill d to the same shared slot for all the threads active inside the loop. Outside the loop of `sumTriangle`, like the related work, we mark d as divergent. Moreover, while the other analyses would consider the branch at Line 7 of `sumTriangle` as divergent, we mark it as uniform. There is one further difference between our methods and these previous works in terms of implementation. We use the GSA form to obtain a sparse analysis, whereas Grover et al. Karrenberg et al. and Lee et al. have opted for a dense style that binds information to pairs of variables and program points.

Figure 2 summarizes this discussion, comparing the results produced by these different variations of the divergence analysis when applied on the kernels in Figure 1. We call *Data Dep.* a divergence analysis that takes data dependencies into consideration, but not control dependencies. In this case, a variable is uniform if it is initialized with constants or broadcasted values or, recursively, if it is a function of only uniform variables. This analysis would incorrectly flag variable `d` in `sumTriangle` as uniform. Notice that because this article's analyses use the GSA intermediate representation, they distinguish the live ranges of variable `d` before ($\mathtt{d}_{bf}$), inside ($\mathtt{d}_{lp}$), and after ($\mathtt{d}_{af}$) the loops. The analysis that we present in Section 3.4 improves on the analysis that we discuss in Section 3.3, because it considers affine relations between variables. Thus, it can report that the loop in `avgSquare` is nondivergent by noticing that the comparison `i < N` always has the same value for every thread. This fact happens because both variables are functions of two affine expressions of $\mathtt{T}_{id}$ whose combination cancels the $\mathtt{T}_{id}$

	Data Dep.	Aiken	Karr.11	Grover, Lee.13, Karr.12	Sec. 3.3	Sec. 3.4
c	U	U	U	U	U	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
m	U	U	U	U	U	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
v	U	U	U	U	U	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
i	D	D	ca	D	D	$0\mathtt{T}_{id}^2 + \mathtt{T}_{id} + \bot$
avgSquare						
N	D	D	c	D	D	$0\mathtt{T}_{id}^2 + \mathtt{T}_{id} + \bot$
$\mathtt{d}_{bf}$	U	D	U	U	U	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + 0$
$\mathtt{d}_{lp}$	U	D	D	D	U	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
$\mathtt{d}_{af}$	U	D	D	D	D	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
sumTriangle						
L	D	D	D	D	D	$0\mathtt{T}_{id}^2 + \bot\mathtt{T}_{id} + \bot$
$\mathtt{d}_{bf}$	U	D	U	U	U	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + 0$
$\mathtt{d}_{lp}$	U	D	D	D	U	$0\mathtt{T}_{id}^2 + 0\mathtt{T}_{id} + \bot$
$\mathtt{d}_{af}$	U	D	D	D	D	$\bot\mathtt{T}_{id}^2 + \bot\mathtt{T}_{id} + \bot$

Fig. 2. A comparison between different versions of divergence analyses. We use U for uniform and D for divergent variables. Karrenberg et al.'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). As we explain in Section 3.4, the symbol $\bot$ denotes unknown values.

factor out, for example, $\mathtt{N} = \mathtt{T}_{id} + c_1$ and $\mathtt{i} = \mathtt{T}_{id} + c_2$; thus, $\mathtt{N - i} = (1-1)\mathtt{T}_{id} + c_1 - c_2$. It is worth pointing out that none of the other divergent analyses that we have discussed here (not even that in Section 3.3) is expressive enough to reach this conclusion.

3. DIVERGENCE ANALYSES

In this section, we describe two divergence analyses. The first, which we present in Section 3.3, has a very simple and fast implementation. This initial analysis helps us formalize the second slower, yet more precise, algorithm, which we present in Section 3.4. This formalization uses a simple SIMD language introduced in Section 3.1, which we call μ-SIMD. Our divergence analyses work on a preprocessed version of μ-SIMD programs. Preprocessing, in our case, consists of converting the μ-SIMD programs to an intermediate representation called Gated Static Single Assignment (GSA) form, that we describe in Section 3.2.

3.1. The Core Language

In order to formalize the theory that we develop in this article, 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 *processing elements* (PEs) executing instructions in lock-step, yet subject to *partial execution*. In the words of Farrel and Kieronska, "All PEs execute the same statement at the same time with the internal state of each PE being either active or inactive" [1996, p. 40]. The archetype of an SIMD machine is the ILLIAC IV Computer [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; Keryell et al. 1991; Kung et al. 1982; Lawrie et al. 1975; Perrot 1979]. The recent developments in graphics cards have brought new members to this family. The Single Instruction Multiple Threads (SIMT) [Garland and Kirk 2010; Nickolls and Kirk 2009; Nickolls and Dally 2010] execution model, a term made popular by Nvidia's GPUs, is currently implemented as a multicore SIMD machine—CUDA being a programming language 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 Figure 3. We do not reuse the formal semantics of Bougé and leraire or Farrell

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`

Fig. 3. The syntax of μ-SIMD instructions.

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

Fig. 4. Elements that constitute the state of a μ-SIMD program.

and kierarska, 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 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 on our webpage [Pereira 2011].

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 Figure 4. A processing element is a pair (t, σ) uniquely identified by the natural t, referred to 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 labels to instructions. The result of executing a μ-SIMD abstract machine is a pair (Θ, Σ). 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 have 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.

Figures 5, 6, and 7 describe the big-step semantics of μ-SIMD. We use the auxiliary functions in Figure 5 plus the rules in Figure 6 to determine the semantics of instructions that change the program's control flow. According to Rule SP, 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 $\sigma(v) \neq 0$ for

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

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

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

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

Fig. 5. The auxiliary functions used in the definition of μ-SIMD.

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

(BT) $$\frac{P[pc] = \texttt{bz}\ v, l \qquad \textbf{split}(\Theta, v) = (\Theta, \emptyset) \qquad \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \Sigma, \Pi', P, l) \to (\Theta', \Sigma')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

(BF) $$\frac{P[pc] = \texttt{bz}\ v, l \qquad \textbf{split}(\Theta, v) = (\emptyset, \Theta) \qquad \textbf{push}(\Pi, \emptyset, pc, l) = \Pi' \qquad (\Theta, \Sigma, \Pi', P, pc+1) \to (\Theta', \Sigma')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

(BD) $$\frac{P[pc] = \texttt{bz}\ v, l \qquad \textbf{split}(\Theta, v) = (\Theta_0, \Theta_n) \qquad \textbf{push}(\Pi, \Theta_n, pc, l) = \Pi' \qquad (\Theta_0, \Sigma, \Pi', P, pc+1) \to (\Theta', \Sigma')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

(SS) $$\frac{P[pc] = \texttt{sync} \qquad \Theta_n \neq \emptyset \qquad (\Theta_n, \Sigma, (pc', \Theta_0, l, \emptyset) : \Pi, P, l) \to (\Theta', \Sigma')}{(\Theta, \Sigma, (pc', \emptyset, l, \Theta_n) : \Pi, P, pc) \to (\Theta', \Sigma')}$$

(SP) $$\frac{P[pc] = \texttt{sync} \qquad (\Theta_n, \Sigma, (\_, \emptyset, \_, \Theta_0) : \Pi, P, pc+1) \to (\Theta', \Sigma')}{(\Theta_0 \cup \Theta_n, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

(JP) $$\frac{P[pc] = \texttt{jump}\ l \qquad (\Theta, \Sigma, \Pi, P, l) \to (\Theta', \Sigma')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta', \Sigma')}$$

(IT) $$\frac{P[pc] = \iota \qquad \iota \notin \{\texttt{stop}, \texttt{bnz}, \texttt{bz}, \texttt{sync}, \texttt{jump}\} \qquad (\Theta, \Sigma, \iota) \to (\Theta', \Sigma') \qquad (\Theta', \Sigma', \Pi, pc+1, \Theta'', \Sigma'')}{(\Theta, \Sigma, \Pi, P, pc) \to (\Theta'', \Sigma'')}$$

Fig. 6. 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.

every PE, then Rule BF moves the flow to the next instruction, that is, $pc+1$. Similarly, if $\sigma(v) = 0$ 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*. In this case, in Rule BD, we execute the PEs in the "else" side of the branch, keeping the other PEs in the synchronization stack to execute them later. The **push** function in Figure 5 updates this stack. Even the nondivergent branch rules update the synchronization stack so that upon reaching a barrier, that is, 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 "then" branch, keeping the previously active PEs into hold. Finally, if we reach the barrier without any PE waiting

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

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

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

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

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

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

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

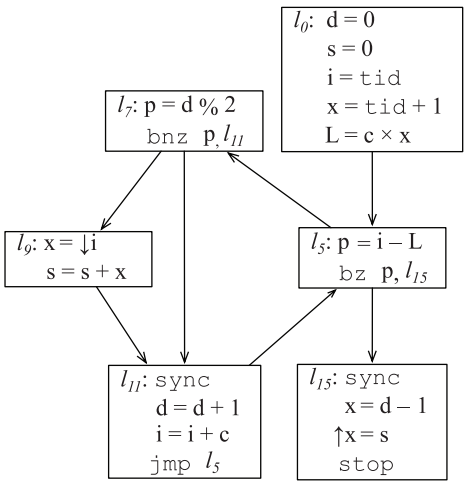
$$\text{(BP)}\quad \frac{t, \sigma \vdash v_2 = c_2 \qquad t, \sigma \vdash v_3 = c_3 \qquad c_1 = c_2 \oplus c_3}{(t, \sigma, \Sigma, v_1 = v_2 \oplus v_3) \rightarrow (\sigma \setminus [v_1 \mapsto c_1], \Sigma)}$$

Fig. 7. The operational semantics of μ-SIMD: data and arithmetic operations.

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].

Figure 7 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 behavior 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. The rest of the rules in Figure 7 are oblivious to the multithreaded nature of μ-SIMD. In other words, they determine the semantics of each instruction executed by a single PE. We use the notation $f[a \mapsto b]$ to denote the updating of function f, that is, $\lambda x.x = a \,?\, b : f(x)$. Rule CT describes the assignment of a constant to a variable. Similarly, Rule AS describes the copy of data from a variable v' to a variable v. Rule LD shows the loading of data from the common memory Σ into a PE's local variable v. In this rule, the contents of variable v_x are used to index Σ. Stores are defined by Rule ST. An instruction such as $\uparrow v_x = v$ copies the contents of v into the cell of Σ indexed by the contents of v_x. The store instruction might lead to a data race, that is, two PEs trying to write different data 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. Rule BP defines the execution of typical binary operations, such as addition and multiplication. The symbol $\oplus$ denotes different operators, which we interpret according to the semantics usually seen in arithmetics.

Figure 8 (left) shows the kernel `sumTriangle` from Figure 1 written in μ-SIMD. To keep the figure clean, we only show the label of the first instruction present in each



Cycle	Instruction	t_0	t_1	t_2	t_3
16	$l_5 : p = i - L$	✓	✓	✓	✓
17	$l_6 :$ bz p, l_{15}	✓	✓	✓	✓
18	$l_7 : p = d \,\%\, 2$	•	✓	✓	✓
19	$l_8 :$ bnz p, l_{11}	•	✓	✓	✓
20	$l_9 : x = \downarrow i$	•	✓	✓	✓
21	$l_{10} : s = s + x$	•	✓	✓	✓
22	$l_{11} :$ sync	•	✓	✓	✓
23	$l_{12} : d = d + 1$	•	✓	✓	✓
24	$l_{13} : i = i + c$	•	✓	✓	✓
25	$l_{14} :$ jmp l_5	•	✓	✓	✓
26	$l_5 : p = i - L$	•	✓	✓	✓
27	$l_6 :$ bz p, l_{15}	•	✓	✓	✓
28	$l_7 : p = d \,\%\, 2$	•	•	✓	✓
29	$l_8 :$ bnz p, l_{11}	•	•	✓	✓
	...				
43	$l_5 :$ bz p, l_{15}	•	•	•	✓
44	$l_{15} :$ sync	✓	✓	✓	✓

Fig. 8. (Left) Example of a μ-SIMD program. (Right) Snapshot of the execution trace of the μ-SIMD program on the left. 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 •.

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 8 (right) illustrates divergences 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, in cycle 17, the predicate p is 0 for thread t_0, and 1 for the other PEs. In the 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 takes place in Cycle 27, 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 8. Once t_3 leaves the loop, all the threads synchronize via the sync instruction at label l_{15} and resume lock-step execution.

3.2. Gated Static Single Assignment Form

To better handle control dependences between program variables, we work with μ-SIMD programs in Gated Static Single Assignment form [Ottenstein et al. 1990; Tu and Padua 1995] (GSA). Figure 9 shows the program in Figure 8 converted to GSA form. This intermediate program representation differs from the well-known Static Single Assignment [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 [Ottenstein et al. 1990].

—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 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.

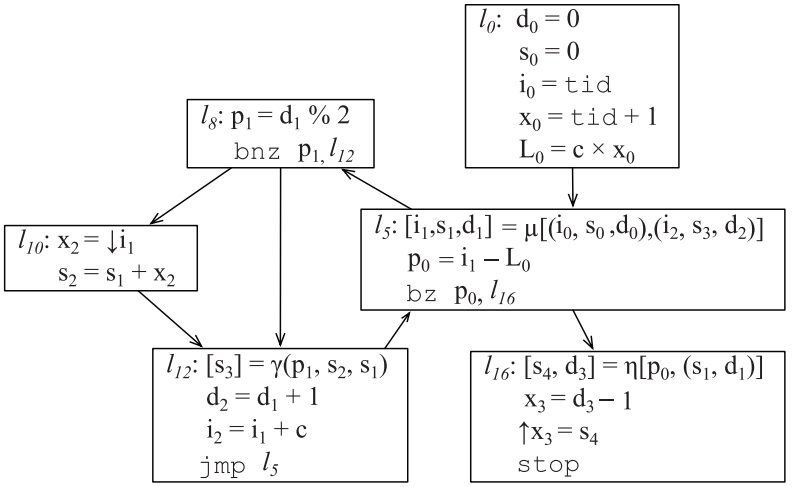


Fig. 9. The program from Figure 8 converted into GSA form.

—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 Tu and Padua's [1995] almost linear time algorithm to convert a program into GSA form. According to this algorithm, γ and η functions exist at the postdominator of the branch that controls them. A label l_p postdominates 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] have shown that reconverging divergent PEs at the immediate postdominator of the divergent branch is nearly optimal with respect to maximizing hardware utilization. Although Fung et al. have discovered situations in which it is better to do this reconvergence 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 [Appel 1998] to evaluate these functions, and we denote these parallel copies using Hack's matrix notation [Hack and Goos 2006]. For instance, the μ assignment at l_5 in Figure 9 denotes two parallel copies: either we perform $[i_1, s_1, d_1] = (i_0, s_0, d_0)$ in the case that we are entering the loop for the first time, or we do $[i_1, s_1, d_1] = (i_2, s_3, d_2)$.

We work on GSA-form programs because this intermediate representation allows us to transform *control dependences* into *data dependences* 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 9 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 8, 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 9, 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 $[s_3] = \gamma(p_1, s_2, s_1)$ creates data dependences from s_1 to s_2, s_1 and also p_1, the predicate controlling the branch at l_9. Hence, the GSA format transforms control in data dependences.

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

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

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

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

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

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

Fig. 10. Constraint system used to solve the simple divergence analysis.

3.3. The Simple Divergence Analysis

The simple divergence analysis reports whether a variable v is *uniform* or *divergent*. We say that a variable is uniform if it meets the condition in Definition 3.1, otherwise, it is 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 10. In Figure 10, we let $[\![v]\!]$ denote the abstract state associated with variable v. This abstract state is an element of the lattice $U > D$. This lattice is equipped with a meet operator $\wedge$ such that $a \wedge a = a$ and $U \wedge D = D \wedge U = D$. We optimistically initialize the abstract state of every variable with U. In Figure 10, we use $o_1 \oplus o_2$ for any binary operation, including addition and multiplication. We also use $\oplus o$ for any unary operation, including loads.

Definition 3.1 (*Uniform Variables*). 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 10 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 10 reveals. Therefore, our analysis admits a *sparse* implementation, as defined by Choi et al. [1991]. In the words of Choi et al., sparse dataflow 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 10 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.

Moving on with our example, Figure 11 shows the data-dependence graph created for the program in Figure 9. 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

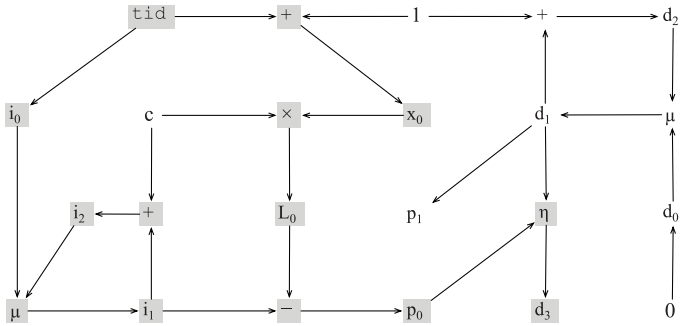


Fig. 11. The dependence graph created for the program in Figure 9. We only show the *program slice* [Weiser 1981] that creates variables p_1 and d_3. Divergent variables are colored gray.

the variable p_0 that controls the loop is. We prove the nondivergence 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 nth 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 redefined d_1 a different number of times. Although this fact cannot cause a divergence inside the loop, divergences might still happen outside it.

THEOREM 3.2. *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 Figure 7, 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 Figure 7, 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 Figure 7, 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 Figure 7, 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. □

3.4. Divergence Analysis with Affine Constraints

The previous analysis is not precise enough to point out that the loop in the kernel `avgSquare` (Figure 1) is nondivergent. In this section, we fix this 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 12 defines the meet operator and the abstract semantics of μ-SIMD's multiplication and addition. Notice that in Figure 12, we do not consider $\top \times a$ or $\top + a$ for any $a \in C$. This is safe because (i) we are working only with strict programs, that is, programs in SSA form in which every variable is defined before being used, (ii) we process the instructions in a pre-order traversal of the program's dominance tree, (iii) in an SSA form program, the definition of a variable always dominates every use of it [Budimlic

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

Fig. 12. Abstract semantics of the meet, multiplication, and addition operators used in the divergence analysis with affine constraints. We let $c_i \in \mathbb{Z}$.

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 13. Constraint system used to solve the divergence analysis with affine constraints of degree one.

et al. 2002], and (iv) upon definition, as we shall see in Figure 13, every variable receives an abstract value different from $\top$.

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 13. Initially, we let $[\![v]\!] = (\top, \top)$ for every v defined in the text of P, and $[\![c]\!] = (0, c)$ for each $c \in \mathbb{Z}$.

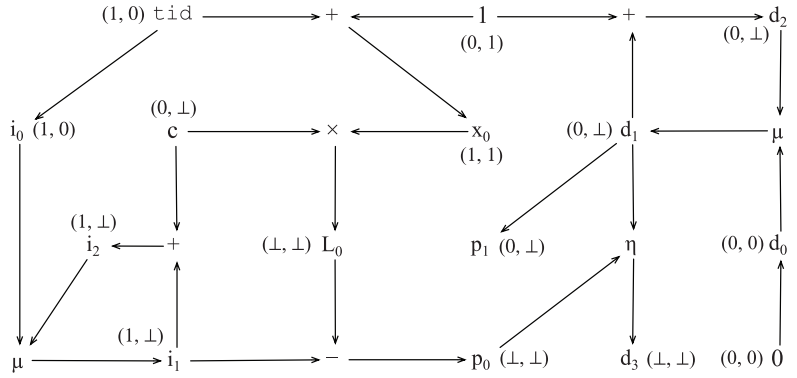


Fig. 14. Results of the divergence analysis with affine constraints for the program slice seen in Figure 11.

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 to terminate in at most five iterations [Nielson et al. 2005]. 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 Section 5 that our analysis is linear in practice. As an example, Figure 14 illustrates the application of the new analysis on the dependence graph first seen in Figure 11. Each node has been augmented with its abstract state, that is, 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 3.3. Furthermore, if the processing elements see v as the same affine function of their thread identifiers, for example, $v = c_1\mathrm{T}_{id} + c_2, c_1, c_2 \in \mathbb{Z}$, then we say that v is *affine*.

THEOREM 3.3. *If* $[\![v]\!] = 0\mathrm{T}_{id} + a, a \in C$, *then* v *is uniform. If* $[\![v]\!] = c\,\mathrm{T}_{id} + a, a \in C, c \in \mathbb{Z}, c \neq 0$, *then* v *is affine.*

PROOF. The proof is by structural induction on the rules in Figure 13. We will show a few cases.

—*CntA*. A variable initialized with a constant is uniform, given Rule Ct in Figure 7. 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 $[\![v_1]\!] = 0\mathrm{T}_{id} + a_1$ and $[\![v_2]\!] = 0\mathrm{T}_{id} + a_2$, where $a_1, a_2 \in C$. Thus, $[\![v]\!] = (0 + 0)\mathrm{T}_{id} + (a_1 + a_2)$. 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 $[\![v_1]\!] = c_1\mathrm{T}_{id} + a_1$ and $[\![v_2]\!] = c_2\mathrm{T}_{id} + a_2$, where $c_1, c_2 \in \mathbb{Z}$ and $a_1, a_2 \in C$. Thus, $[\![v]\!] = (c_1 + c_2)\mathrm{T}_{id} + (a_1 + a_2)$, and the result holds for the same 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 T_{id} plus a constant parcel a for every PE. The sum of these constant parcels, for example, $a_1 + a_2$ 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 12. These four cases abide by the semantics of addition if we replace $\oplus$ by $+$ in Rule Bp of Figure 7.
—*EtaA*. We know that p is uniform; hence, by either Rule Bt or Bf in Figure 7, PEs reach 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 T_{id} coefficient at that moment. Thus, v is either uniform or affine, by Rule As from Figure 7.

—*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 Figure 7. 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 $\perp$ 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 3.3, as Corollary 3.4 shows.

COROLLARY 3.4. *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 3.3's analysis, v is uniform if it is a function of only uniform variables, for example, $v = f(v_1, \ldots, v_n)$, and every v_i, $1 \leq i \leq n$ is uniform. From Theorem 3.2, 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 a specialization of a framework that we call *the divergence analysis with polynomial constraints*. In the general case, we let $[\![v_i]\!] = a_n\mathtt{T}_{id}^n + a_{n-1}\mathtt{T}_{id}^{n-1} + \cdots + a_1\mathtt{T}_{id} + a_0$, where $a_i \in C, 1 \leq i \leq n$. Addition and multiplication of polynomials follow the usual algebraic rules. The rules in Figure 13 use polynomials of degree one. Constant propagation uses polynomials of degree zero. Our polynomials of degree one are a special instance of Miné's octagons [Miné 2006]. The main difference between our abstract domain and Miné's is that while octagons can relate any two variables, we only relate variables with the thread identifier, hence obtaining a more efficient implementation. Similarly, analyses involving more than one different thread identifier can be seen as special cases of Cousot and Halbwachs polyhedrons [Cousot and Halbwachs 1978]. As an example, C for CUDA gives developers three dimensions along which to identify threads, for example, $\mathtt{T}_{id}(x)$, $\mathtt{T}_{id}(y)$, and $\mathtt{T}_{id}(z)$. In practice, we have to handle all these dimensions. In this article, we omit the simple, yet cumbersome, extra machinery that these identifiers require for the sake of simplicity.

There are situations in which polynomials of degree two let us find more affine variables. The extra precision comes out of Theorem 3.5. Consider, for instance, the program in Figure 15, 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-two analysis marked 39 more variables, out of almost 10,000, as affine, when compared to the degree-one analysis. We could not gain more precision from polynomials of degree three or higher.

THEOREM 3.5. *If $[\![v]\!] = 0T_{id}^2 + a_1 T_{id} + a_0, a_1, a_0 \in C$, then v is an affine function of T_{id}.*

PROOF. 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 3.3. □

```
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	$(\bot, \bot)$	$(0, \bot, \bot)$

Fig. 15. An example where a higher-degree polynomial improves the precision of the simple affine analysis. We let $a_2\mathrm{T}_{id}^2 + a_1\mathrm{T}_{id} + a_0 = (a_2, a_1, a_0)$.

4. DIVERGENCE AWARE REGISTER SPILLING

Similar to traditional register allocation, we are interested in finding storage area for 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 570 GPU has 32,768 registers. However, running 1,536 threads at the same time, this SM can afford at most 21 registers to each thread in order to achieve maximum hardware occupancy.
—*Shared Memory*. This fast storage space is addressable by each thread in flight and usually is used as a scratchpad memory. 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 three 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 570 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 spiller 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 [Nickolls and Dally 2010]. When moving nondivergent 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 spiller may provide up to a 32-fold improvement in cache locality.

Figure 16 shows the instance of the register allocation problem that we obtain from the kernel `avgSquare` in Figure 1. There are many ways to model register allocation. In this article, we use an approach called *linear scan* [Poletto and Sarkar 1999]. Thus,

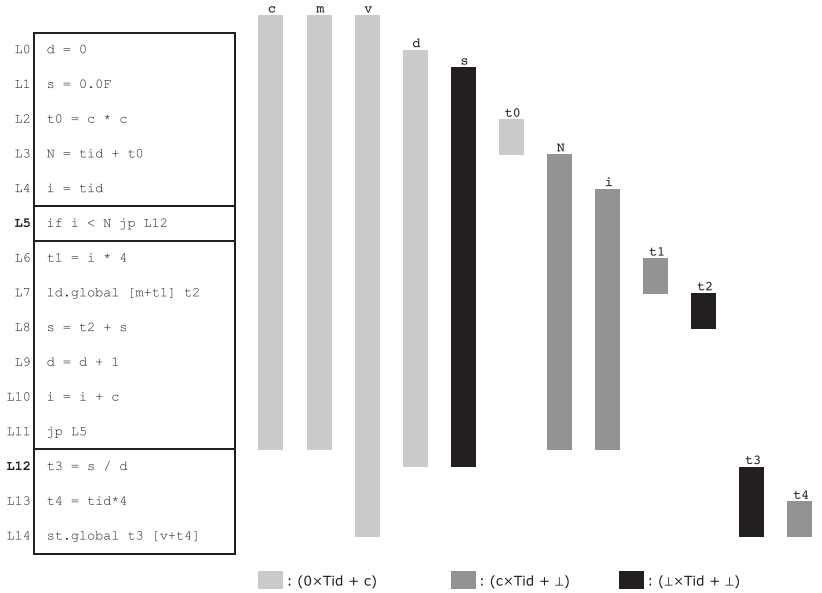


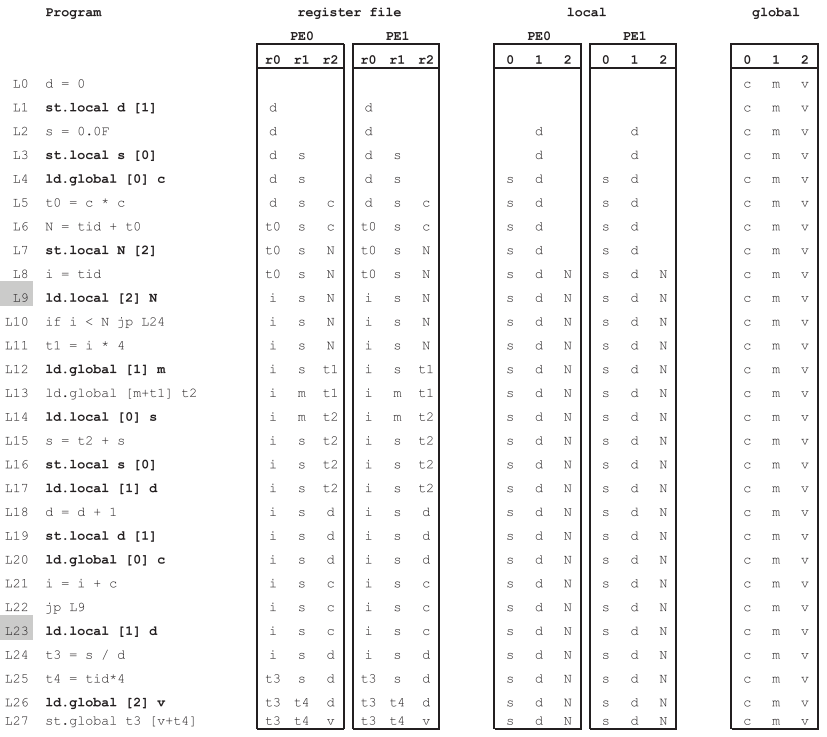
Fig. 16. The register allocation problem for the kernel `avgSquare` in Figure 1.

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.

If the register pressure exceeds the number of available registers at a given program point p, then our linear scan chooses one of the live variables and maps it into memory, a process called *spilling*. We spill the variable that has the farthest use from p, following Belady's heuristics [1966]. Current register allocators for graphics processing units place spilled values in the local memory. Figure 17 illustrates this approach. In this example, we assume a warp with two processing elements, 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. The cache mitigates this problem, but it is not a definitive solution. The number of threads using the cache is large—in the order of thousands—and the cache itself is small, for example, 16 KB, therefore, 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.

4.1. 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 18 shows how we replace loads and stores to the local memory by more efficient instructions. The



	Program	register file PE0 r0	r1	r2	register file PE1 r0	r1	r2	local PE0 0	1	2	local PE1 0	1	2	global 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

Fig. 17. Traditional register allocation, with spilled values placed in 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)	$v = c_1 \mathsf{T}_{id} + c_2$	$\emptyset$
(iv)	$(c, \bot)$	$t =\downarrow v_x; v = c\mathsf{T}_{id} + t$	$t = v_x - c\mathsf{T}_{id}; \uparrow v_x = v$

Fig. 18. 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 describes a rewriting system: we replace loads-to and stores-from local memory 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, in this article, we propose a form of Briggs's style rematerialization [Briggs et al. 1992] that suits SIMD machines. The lattice that we use in Figure 13 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, that is, $[\![v_x]\!] = (0\mathsf{T}_{id}, c)$, as in Line (i) of Figure 18 or $[\![v_x]\!] = (c_1\mathsf{T}_{id}, c_2)$, 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 19 shows the code that we generate for the program in Figure 16. The most apparent departure from the allocation given in Figure 17 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

	Program	register file PE0 r0	r1	r2	register file PE1 r0	r1	r2	local PE0 0	local PE1 0	shared 0	1	global 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

Fig. 19. Register allocation with variable sharing.

L1 and L19 in Figure 17 have been replaced by stores to shared memory in labels L1 and L20 of Figure 19. Similar changes happened to the instructions that load d from local memory in Figure 17. 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 $[\![v]\!] = c\mathrm{T}_{id} + x$, 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 rewrites the load from local memory seen at Line L9 of Figure 17.

Implementation Details. There are two technical details that we had to take into consideration when implementing our register spiller: which thread writes and reads share spills, and how shared memory is divided between multiple SIMD units. Concerning the first issue, we let all the threads in a warp to access the uniform data. Neither race conditions nor bank conflicts [Gou and Gaydadjiev 2013, Section 2.3] are issues, because all the threads write the same value. The alternative would be to choose only one of them to manipulate uniform spills. However, this solution is hard to implement and yields slower code. The difficulty of choosing valid writer stems from the fact that not all the threads may be active at a given program point. Furthermore, even if we assume that we could choose a writer using code like `if(0 == T`$_{id}$`)` then $\uparrow x$, we still suffer a performance penalty. We have observed empirically, on two different Nvidia GPUs—GTX 560 and GTX 670—that this predication is almost three times slower than simply letting all the threads perform the uniform store of variable x.

The second issue that we had to take into account in our implementation of the divergence-aware spiller is how the shared memory is partitioned among *warps*. Graphics processing units are not exclusively SIMD machines. Rather, they run several SIMD threads, or warps, inside a single SM. Our divergence analysis finds uniform variables per warp. Therefore, in order to implement the divergence-aware register spiller, 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 spiller 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.

5. EXPERIMENTS

This section presents numbers that we produced with the divergence analyses and register allocators available in Ocelot [Diamos et al. 2010] revision 2,233, released on May 2013. We ran Ocelot on a quad-core Intel Core-i7 930 processor at 2.8GHz. This computer also hosts the GPU that we use to execute the kernels: a NVIDIA GTX 570 (Fermi) graphics processing unit that contains 14 stream multiprocessors clocked at 1,464 MHz and 1,280 MB of memory. To avoid performance discrepancy, we disabled CPU and GPU frequency scaling. This GPU allows us to run up to 1,536 threads per SM, each one using 21 registers for maximum occupancy. In our experiments, we have artificially reduced the number of available registers to eight in order to provoke more spills when comparing the different register allocators. Each kernel has access to 48 KB of shared memory and 16 KB of cache for the local memory. In these experiments, we are reserving the 16KB cache to local memory only, that is, the kernels have been compiled with the option `-dlcm=cg`; thus, loads from global memory are not cached. In this way, we have more space in the cache to place spilled code. This setup tends to improve the results of register allocators that only spill into local memory.

Benchmarks. We have tested our divergence analysis in all 395 different CUDA kernels that we took from the 68 applications present in the Rodinia 2.0.1 [Che et al. 2009], Parboil 2.5 [Stratton et al. 2012], and NVIDIA SDK 5.0 benchmark suites. If the kernel already uses too much shared memory, our allocator has no room left to place spilled values in that region, and it does not perform any change in the program. We have observed this situation in 39 kernels. We could compile them with 21 registers but did not find enough storage space available when using only eight. The culprit, in this case, is the excessive number of spills into shared memory due to the high register pressure. A typical example of this kind of kernel is `parboil::mri-gridding::gridding_GPU`, which has a maximum pressure of 68 registers and already uses shared memory liberally. When reporting runtime numbers, we will use only the 40 kernels in our test suite that take the longest time to execute when compiled with eight registers. These kernels are listed in Figure 20. Together, these benchmarks gives us 6,142 PTX instructions. We will use short names, given in that table, to indicate each kernel in the charts that we will show in the rest of this section.

Runtime of the Divergence Analysis with Affine Constraints. Figure 21 compares the runtime of the two divergence analyses seen in Sections 3.3 and 3.4. We are showing results for the 100 largest benchmarks that we have in our test suite. The affine analysis of Section 3.4 took 58.6 msecs to go over all these kernels. On average, the divergence analysis with affine constraints of degree two is 1.39× slower than the simple divergence analysis of Section 3.3. This slowdown is expected, because the affine analysis uses a lattice of height nine, whereas the simple analysis

B	A	K	N	Insts	Vars	Pres.
r	nn	euclid	r.nn	27	24	20
r	kmeans	invert_mapping	r.ks.ig	30	30	12
s	HSOpticalFlow	JacobiIteration	s.hw.jn	33	33	24
r	gaussian	Fan1	r.gn.f1	34	31	20
s	scalarProd	scalarProdGPU	s.sp	37	36	24
s	HSOpticalFlow	UpscaleKernel	s.hw.up	41	41	26
s	reduction	reduce1	s.re.r1	43	41	16
s	matrixMul	matrixMulCUDA	s.mm	45	43	16
s	HSOpticalFlow	ComputeDerivativesKernel	s.hw.cd	46	48	34
r	cfd	cuda_compute_step_factor	r.cd.sf	47	45	29
s	reduction	reduce0	s.re.r0	47	45	18
s	reduction	reduce2	s.re.r2	51	44	16
r	gaussian	Fan2	r.gn.f2	54	51	14
s	reduction	reduce4	s.re.r4	58	55	14
r	cfd	cuda_time_step	r.cd.ts	61	59	34
r	kmeans	kmeansPoint	r.ks.kt	62	55	21
s	clock	timedReduction	s.ck	64	62	16
s	reduction	reduce5	s.re.r5	67	64	16
s	reduction	reduce3	s.re.r3	69	96	17
s	reduction	reduce6	s.re.r6	79	100	25
s	bicubicTexture	d_renderCatRom	s.bt	94	124	53
s	simpleGL	simple_vbo_kernel	s.gl	95	116	38
s	SobolQRNG	sobolGPU_kernel	s.sq	96	113	21
r	streamcluster	pgain_kernel	r.sr	97	132	26
s	Mandelbrot	Mandelbrot0	s.md.M0	106	135	28
r	pathfinder	dynproc_kernel	r.pr.dc	110	136	29
s	BlackScholes	BlackScholesGPU	s.bs	121	144	63
s	bilateralFilter	d_bilateral_filter	s.bf	134	147	36
p	bfs	BFS_in_GPU_kernel	p.bs.gpu	139	175	28
s	HSOpticalFlow	DownscaleKernel	s.hw.dw	155	177	60
s	volumeRender	d_render	s.vr	163	196	38
p	bfs	BFS_kernel_multi_blk_inGPU	p.bs.blk	174	208	33
r	hotspot	calculate_temp	r.ht	205	210	60
p	cutcpp	cuda_cutoff_potential_lattice6overlap	p.ct	227	298	33
p	bfs	BFS_kernel	p.bs.ker	229	337	44
s	HSOpticalFlow	WarpingKernel	s.hw.wa	252	335	60
s	simpleTexture3D	d_render	s.s3	313	357	25
s	imageDenoising	KNN	s.id.KN	368	366	44
r	cfd	cuda_compute_flux	r.cd.cx	721	715	53
r	heartwall	kernel	r.hl	1,348	2,067	51

Fig. 20. The benchmarks that we have used in the experiments discussed in this section. **B**: repository (Nvidia SDK 5.0, Rodinia 2.0.1, Parboil 2.5). **A**: application name. **K**: kernel name. **N**: acronym in the charts. **Insts**: number of PTX instructions before conversion to the GSA format. **Vars**: variables in the GSA format. **Pres**: maximum register pressure.

uses a lattice of height two. We have observed a few extreme cases. As an example, in `sdk::concurrentKernels::mykernel`, a test case with 879 PTX instructions, the affine analysis is 19× slower than the simple divergence analysis. We measure time in CPU ticks, as given by the `rdtsc` ×86 instruction. There is a strong correlation between runtime and number of variables: the coefficient of determination for the simple analysis is 0.957, and for the affine analysis is 0.936. This linear behavior is visually apparent in Figure 21 (top), where we have plotted the time that our affine analysis spends per variable. We conclude from this experiment that in practice, both analyses are linear on the number of variables in the target program. Furthermore, they are cheap enough to be used as standard compilation passes, even in lower optimization levels.

Precision of the Divergence Analysis with Affine Constraints. Figure 22 (top) compares the precision of the simple divergence analysis from Section 3.3 and the analysis

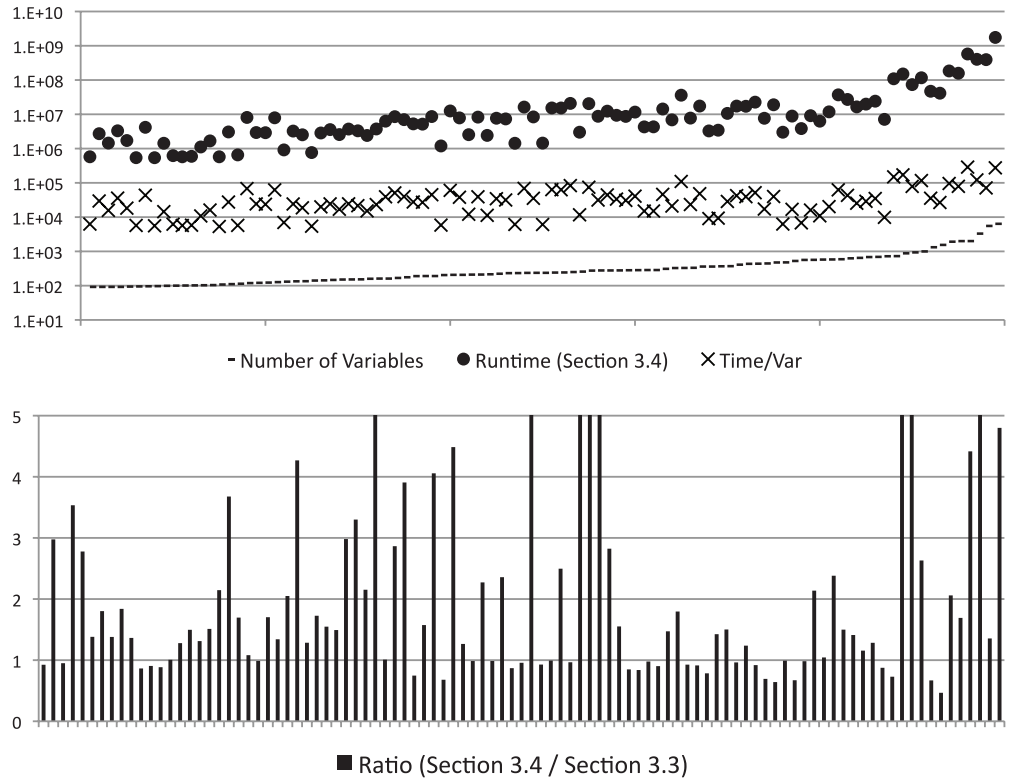


Fig. 21. (Top) Time, in CPU cycles, to run the divergent analyses compared with the number of variables per kernel in GSA-form. Points in the X-axis are kernels, sorted by the number of variables they contain. (Bottom) Comparison between times to run our two divergence analyses. Each bar gives time of affine analysis over time of simple analysis. We have cropped bars that exceed a slowdown of 5×. We report time of analyses only, excluding other compilation phases.

with affine constraints from Section 3.4. The simple analysis reports that 56.02% of the variables are divergent, while the affine analysis gives 54.42%. However, whereas the simple divergence analysis only marks a variable as uniform or not, the affine analysis can find that a nontrivial proportion of the divergent variables are affine functions of some thread identifier. Figure 22 (bottom) gives the distribution of the abstract states that we found with the divergence analysis with affine constraints of degree one. In that figure, we let $a_1\mathrm{T}_{id} + a_0 = (a_1, a_0)$, for $a_i \in \{\bot, c, 0\}$. Even though we reported that 56.02% of the variables are divergent, that is, have $a_1 \neq 0$, we found out that 20.70% of these variables are affine functions of some thread identifier. As we show later, this information is vital to register allocation.

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 6,142 variables. We have not found any situation in which higher degrees would improve on the second-degree analysis. Consequently, polynomials of degree one are today the default option in Ocelot.

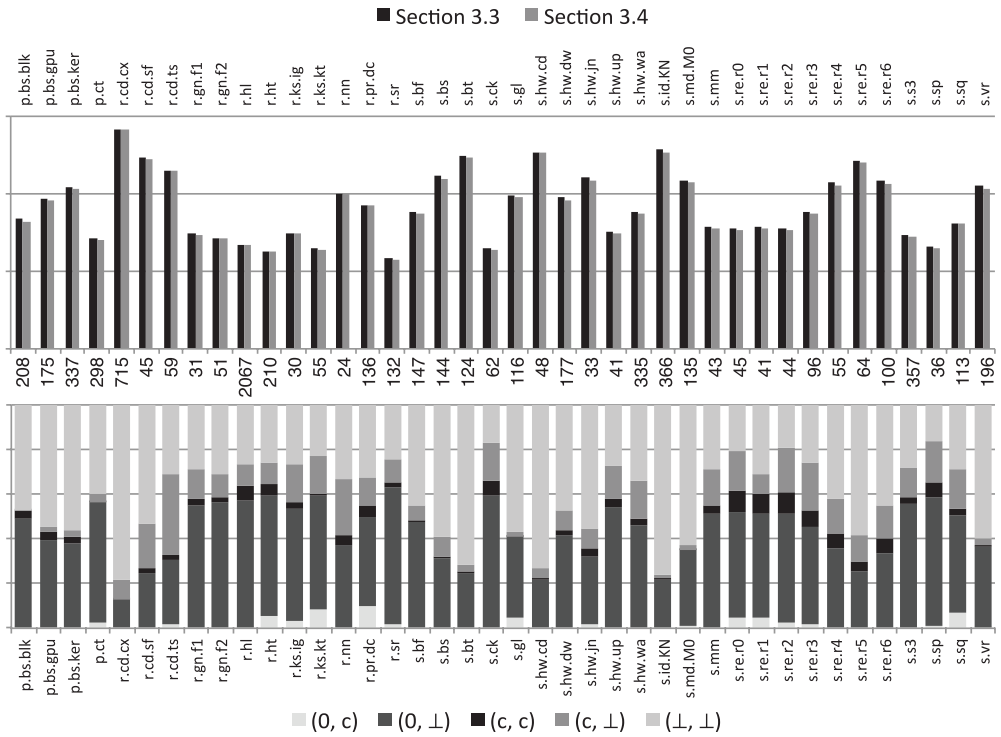


Fig. 22. (Top) Percentage of divergent variables (0–100%) reported by the simple divergence analysis of Section 3.3, and the divergence analysis with affine constraints of Section 3.4. (Bottom) Abstract states that the divergence analysis with affine constraints of Section 3.4 finds for the program variables. Values between charts give number of variables in each kernel in GSA form.

5.1. Register allocation

Figure 23 compares the runtime of code produced by three different implementations of register spillers. We use, as a baseline, the spiller present in the linear scan register allocator [Poletto and Sarkar 1999] that is publicly available in the Ocelot distribution. The two other allocators are implemented as rewriting patterns that change the spill code inserted by linear scan according to the rules in Figure 18. All these three allocators use the same policy to assign variables to registers and to compute spilling costs. The divergence aware spillers are DivRA, which moves to shared memory the variables that the simple divergence analysis of Section 3.3 marks as uniform, and AffRA, which uses all the four rules in Figure 18 guided by the analysis of Section 3.4 with polynomials of degree one. Notice that DivRA can only use row (ii) in Figure 18;

Figure 23 reports time for each kernel individually, instead of showing the runtime of an entire application made of several kernels. Although kernels run in the GPU, we measure their runtime in CPU ticks by synchronizing the start and end of each kernel call with the CPU. We have run each benchmark 15 times, and the variance is negligible: in all the experiments, the difference between the minimum and the maximum time observed was less than 1%; hence, we omit error bars for the sake of legibility. We take about one and a half hours to execute the 40 benchmarks 15 times on our GTX 570 GPU. In this experiment, we have reduced the quantity of registers available to each thread in order to increase the number of spills. In this way, we have more opportunities to compare the quality of the code produced by the different spillers under a situation of extreme stress. Linear scan uses nine registers, whereas

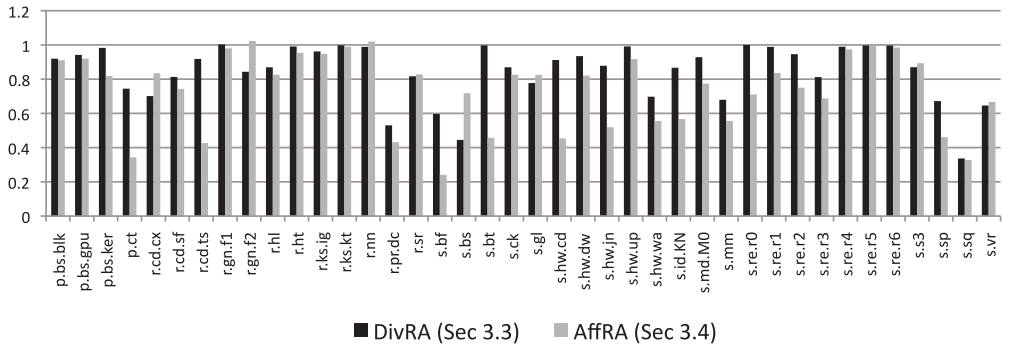


Fig. 23. Relative kernel execution time with different register allocators. Bars are normalized to the time given by Ocelot's linear scan allocator, which is divergence oblivious. The shorter the bar, the faster the kernel.

DivRA and AffRA 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. Additionally, AffRA uses one of its registers to load $\mathtt{T}_{id}$, as PTX's application binary interface requires this special variable to be in register when used as an operand.

On average, all the divergence-aware register spillers improve on Ocelot's original linear scan. DivRA yields a speedup of 15.37%, and AffRA gives a speedup of 26.21%. These numbers are the geometric mean over the results reported in Figure 23. There are situations when both DivRA and AffRA may produce code that is slower than the original linear scan algorithm. We have detected this behavior in `rodinia::nn::euclid`, for instance. This fact happens because (i) the local memory benefits from a 16KB cache that is as fast as shared memory; (ii) loads and stores to shared memory take three instructions each: a type conversion, a multiply add, and the memory access itself; and (iii) DivRA and AffRA insert into the kernel some setup code to delimit the storage area that is given to each warp. This code naturally demands some execution cycles. Nevertheless, this experiment lets us conclude that a divergence-aware register spiller produces code that is substantially faster than the binaries generated by an allocator that is oblivious to the idiosyncrasies of the SIMD world.

Affinity is Essential Information to Divergence-Aware Register Spilling. Figure 24 shows how the different divergence-aware register spillers target memory with load instructions. DivRA can either load values from the shared or the local memory. AffRA, in addition to these two alternatives, can also rematerialize values using either row (i) or (iii) of Figure 18. Rematerialization does not target any kind of memory. We call the code used to load or rematerialize values a *use reconstruction*. The main conclusion that we draw from this figure is the fact that affinity information is essential to reducing the amount of access to local memory. AffRA tends to insert more reconstruction code than DivRA. The former spiller deals with larger register pressure because it must load $\mathtt{T}_{id}$ in a register to operate with it. In our 40 benchmarks, AffRA had to reconstruct 2,287 uses of spilled variables, whereas DivRA had to reconstruct 2,058. Nevertheless, only 850 loads inserted by AffRA target the local memory. On the other hand, 1,572 loads inserted by DivRA read data from that memory.

The key to avoid going to local memory is affinity information. To illustrate this fact, Figure 25 explicitly separates the reconstruction code inserted by AffRA. We use $\Downarrow$ and $\downarrow$ to denote loads from local and shared memory, respectively. The tuples $(0, \perp)$, (c, c), and $(c, \perp)$ refer to the second, third, and fourth lines of Figure 18, respectively. Loads such as $\downarrow (c, c)$ and $\downarrow (c, \perp)$ would be considered divergent by the analysis of

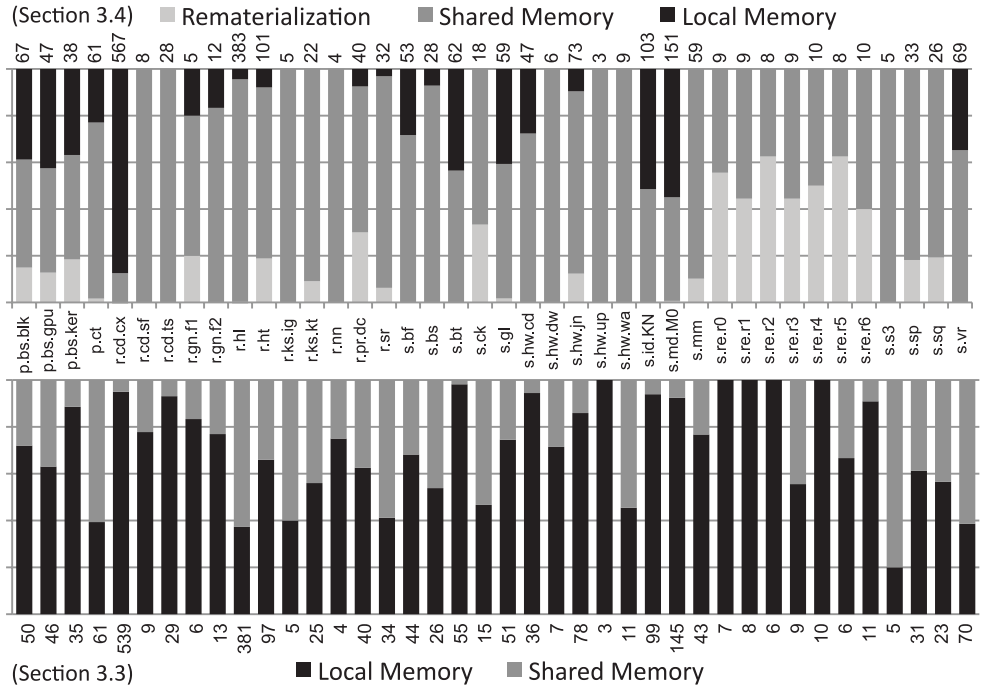


Fig. 24. Target location of instructions used to load the variables spilled by AffRA with eight registers and DivRA with nine. The numbers give the total quantity of load instructions.

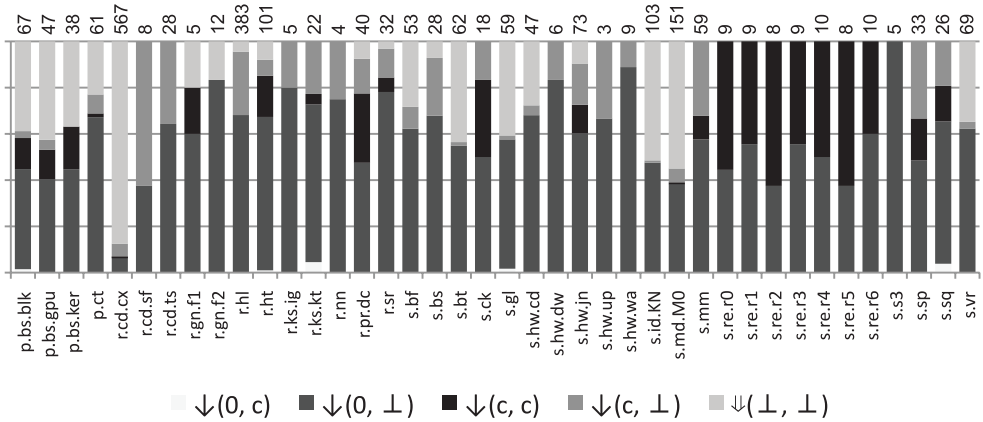


Fig. 25. Type of reconstruction code inserted by AffRA with polynomials of degree one. Values above bars give number of uses of spilled variables.

Section 3.3; hence, they are mapped onto the local memory by DivRA. On the other hand, the more precise analysis of Section 3.4 gives AffRA enough information to load these variables from the shared memory. Incidentally, by comparing Figures 25 and 23, we observe that the largest performance speedups of AffRA over DivRA have been obtained in benchmarks that contain a large proportion of instructions, such as $\downarrow (c, c)$ and $\downarrow (c, \bot)$. Examples of these benchmarks include `s.re.r1`, `s.re.r2` and `s.re.r3`. In `sdk::reduce::reduce3` (s.ra.r3), for instance, AffRA had to spill two variables of type

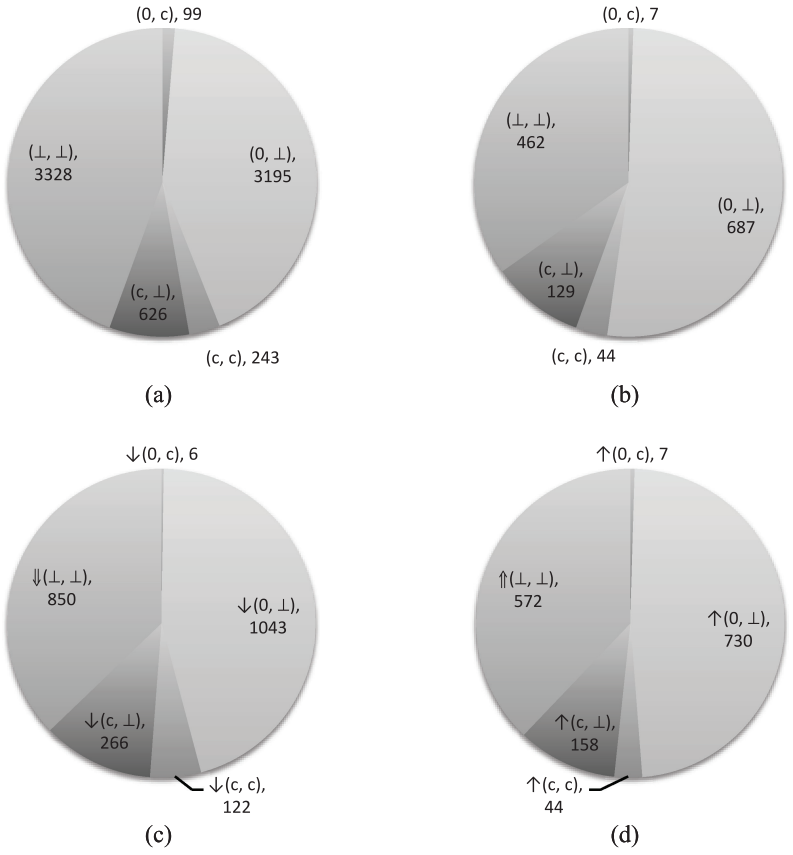


Fig. 26. (a) Distribution of abstract states among the variables in our test suite of 40 kernels. (b) Abstract state of the variables spilled by AffRA with eight registers. (c) Type of instructions used to load or rematerialize spilled values. (d) Type of instructions used to store spilled values.

(c, c), with four uses, and four variables of type $(0, \bot)$ with five uses. Whereas AffRA can rematerialize the four affine uses, DivRA would have to map them into local memory, as they would be considered divergent.

A Summary of Abstract States in the Context of Register Allocation. Figure 26 summarizes the information that the divergence analysis with affine constraints produces to our benchmarks using polynomials of degree one. By comparing Figures 26(a) and 26(b), we notice that the proportion of uniform variables that are spilled is larger than the total percentage of these variables in our test suite. Our divergence-aware register spillers do not assign a lower spilling cost to uniform variables. Thus, we speculate that this difference happens because uniform variables tend to have longer live ranges. A previous study by Collange et al. [2009] corroborates this hypothesis. Figures 26(c) and 26(d) show the proportion of spilling instructions, that is, loads and stores, that are inserted by AffRA. Ocelot's linear scan uses a "spill-everywhere" approach: each use of a spilled variable must be reconstructed, either via a load instruction or via rematerialization. Similarly, each definition of a spilled variable must be suffixed with code to save its value. We perform register allocation after the SSA elimination phase; thus, we may have more than one definition of each variable. This observation explains why

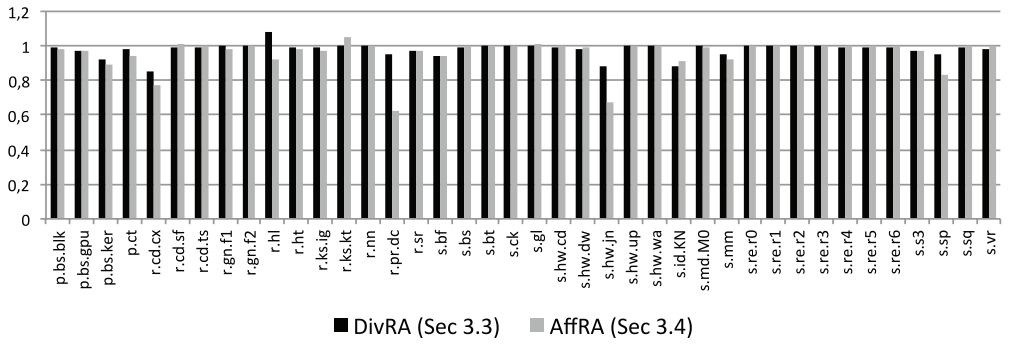


Fig. 27. Relative speedup obtained by different register spillers in a setup with 21 registers available for each thread. Like in Figure 23, bars are normalized to the runtime produced by Ocelot's linear scan register allocator. The shorter the bar, the faster the kernel.

we have more store instructions than spilled variables in our programs. Overall, we reconstruct 2,287 uses, as we have mentioned before, and had to save 1,511 definitions via store instructions. As we see in Figure 26(d), $44 + 7$ of these stores did not require any code, because we dematerialize variables bound to either $(0, c)$ or (c, c).

Runtime vs. Number of Available Registers. As we have mentioned before, we have limited our previous experiments to eight registers to exercise our spillers in a setting with high register pressure. However, our GTX 570 GPU provides us with a large register file with 32K registers. Hence, we can give each PE up to 21 registers and still use all the 1,536 potential physical threads. If the number of available registers is high enough, then spills are a rare event, and the performance gap between the different register spillers tends to decrease. Figure 27 makes this trend clear by showing the runtime of the binaries produced by the different register allocators that we have, assuming that we can give each thread up to 21 registers. In this setup, AffRA yielded code 3.84% faster than linear scan, and DivRA yielded code 1.47% faster.

To study scenarios with different register banks, we have produced histograms to four kernels, showing how the runtime of the binaries produced by AffRA varies with the increase in the number of available registers. These histograms are given in Figure 28. The first three kernels, `rodinia::dynproc_kernel`, `sdk::d_bilateral_filter`, `rodinia::cuda_compute_flux`, and `parboil::cuda_cutoff_potential_lattice` gave us the largest speedups obtained by the affine-aware spiller (AffRA) over linear scan, considering only eight registers available. The fourth, `rodinia::cuda_compute_flux` gave us the largest number of variables spilled by AffRA. We vary the number of registers from eight to 23. In `rodinia::cuda_compute_flux`, the kernel with the highest register pressure out of our four examples, the speedup of AffRA and DivRA over the traditional linear scan is still noticeable.

The Impact of Cache on Register Allocation. Our GPU, the Nvidia GTX 570, lets us use two different cache configurations: from a total of 64KB on-chip memory, we can separate either 48KB or 16KB to the L1 cache. The remaining space is used as shared memory. In the previous experiments, we have used the configuration that allocates 48 KB to the shared memory. This configuration is the default. In this section, we will analyze the behavior of our spillers with the larger cache. Changing the default setup requires the modification of the source code, which must contain a call to the necessary configuration routine. Because this task demands some familiarity with the internals of each benchmark, we have only applied the change in a few of them. Figure 29 summarizes some of our findings.

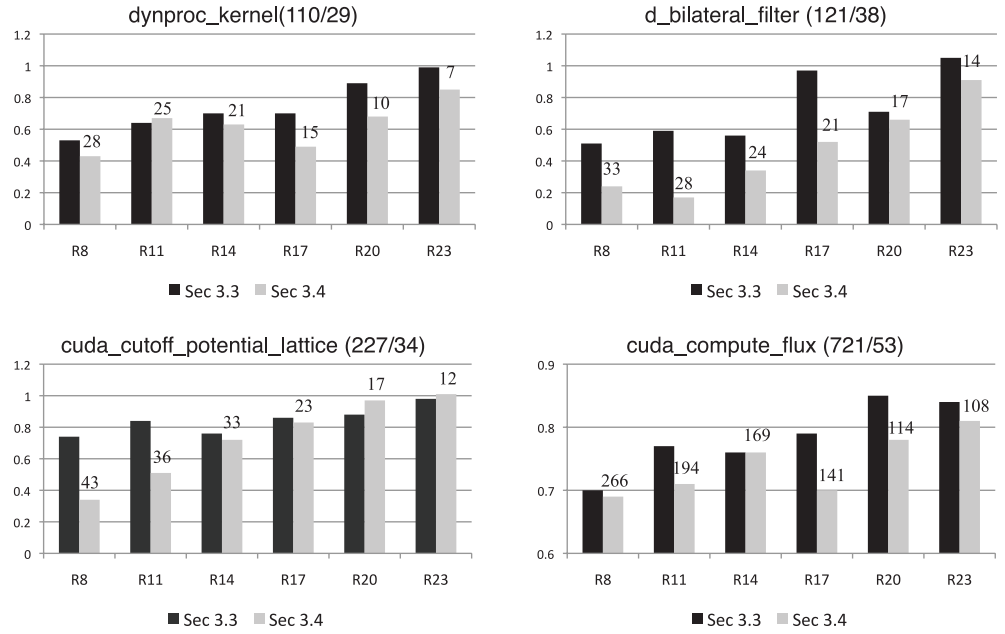


Fig. 28. Runtime of code produced by the different register spillers versus the number of available registers. Bars are normalized by the runtime of code produced by Ocelot's linear scan. Values above bars show number of variables spilled by AffRA. Values next to kernel names give number of PTX instructions in the program, and maximum register pressure. The shorter the bar, the faster the code.

Our divergence-aware spillers cannot generate code to some benchmarks if they only have 16KB of shared memory available. We have observed this behavior, for instance, in `rodinia::cuda_compute_flux`, when compiled with only eight registers. The spillers fail for lack of storage space: the 16 KB reserved to the shared memory is half the size of the register bank of the GTX 570! The configuration with the small shared memory might also reduce the occupancy of the processing units available. In a modern GPU, sets of SIMD threads, for example, warps, are further grouped into units called *blocks*. The shared memory is visible to all the threads in the same block. We can run multiple blocks, as long as the amount of shared storage that each of them requires fits together into the total space available for the shared memory. As an example, Figure 29 shows that in the setting with eight registers, for `dynproc_kernel` and `simple_vbo_kernel`, the execution time increases with the 48KB cache. A larger cache implies less shared memory space. The small number of registers forces too many uniform and affine variables into shared cells; hence, increasing the demands of each block. Unable to meet these demands, the GPU scheduler postpones the execution of one of the blocks, which leads to the slowdown. Notice that in these two cases, the slowdown is only observable in settings with few registers available. The availability of more registers reduces the pressure on the shared memory, thus leading to full hardware occupancy DivRA can also lead to lower hardware occupancy if it places too many uniform variables in the shared memory, but we have not detected this behavior.

The functioning of `dynproc_kernel` and `simple_vbo_kernel` in the register-rich environment is not a coincidence. In general, the larger cache speeds up code generated by any of our three allocators, as long as the hardware occupancy is not compromised. Allocators that use the local memory more are more sensitive to the size of the cache. Figure 29 shows that AffRA is much more stable than Ocelot's original allocator or

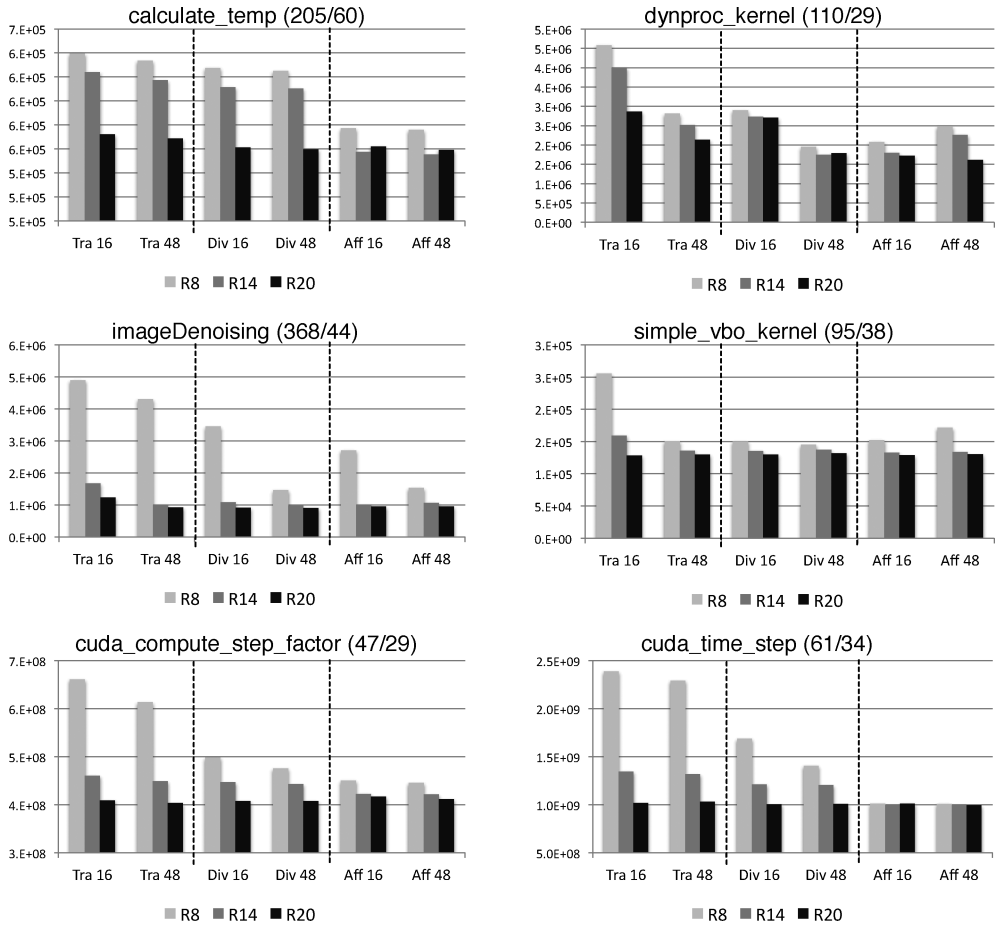


Fig. 29. Impact of the cache configuration on the runtime of six benchmarks. Bars shows absolute times in CPU ticks. Tra is the traditional (divergence oblivious) spiller used by Ocelot. We use Div for DivRA and Aff for AffRA. We write 16 to denote a 16KB L1 cache, and 48 to denote a 48KB L1 cache.

even DivRA, as it places less pressure on the L1 cache. We expect that if the number of physical threads in the upcoming GPUs increases faster than the size of their caches, then our divergence aware allocators will be even more relevant. We conclude this study by pointing out that the divergence-aware allocators tend to outperform the traditional algorithm in any setup. This result holds even when the latter is given a cache that is three times the size of the cache given to the divergence-aware allocators. The only exception to this general trend was `simple_vbo_kernel`. In the register-poor enviornment, Ocelot's original allocator with a 48KB cache produced faster code than AffRA with only 16 KB of shared memory. The villain, in this case, was the lower hardware occupancy.

6. CONCLUSION

This article has presented divergence analysis, a technique that helps developers and compilers to better understand the behavior of programs that execute on SIMD environments. We have discussed two different implementations of this analysis. The first, seen in Section 3.3, has a simple and very efficient implementation. The second, seen

in Section 3.4, is more elaborate but provides better precision. This article has also introduced the notion of a divergence-aware register spiller. We have tested our ideas on an NVIDIA GPU, but we believe that they will work in any SIMD-like environment.

While we claim that this work improves the quality of the code that compilers generate to graphics processing units, it is our understanding that there is still much work to be done in terms of software engineering. All the algorithms that we have presented in this article were implemented in a compiler back-end that optimizes code written in the PTX intermediate representation. As such, currently our ideas support the compiler, but not the programmer. On the other hand, the different flavors of divergence analyses can be very useful to software development as well. By automatically pointing out divergent branches, uncoalesced memory accesses, uniform data, and affine relations between variables, techniques similar to ours could be used to guide the code developer into obtaining maximum benefit from a SIMD-like programming language. Given that parallelism is one of the key pillars on which the high performance of the contemporary computer rests, we expect techniques like ours to be more important each day.

Reproducibility. All the algorithms discussed in this article are publicly available in the Ocelot compiler. Tables with experimental data plus further material discussing our techniques are publicly available at http://simdopt.wordpress.com/.

ACKNOWLEDGMENTS

We thank the Ocelot community for helping us implement all the techniques that we describe in this article. We thank the referees of prior manuscripts, as well as the reviewers of this work, for very helpful and constructive comments.

REFERENCES

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 AFIPS*. ACM, 57–73.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. 2006. *Compilers: Principles, Techniques, and Tools* 2nd Ed. Addison Wesley.

Aiken, A. and Gay, D. 1998. Barrier inference. In *Proceedings of POPL*. ACM, 342–354.

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. In *Proceedings of PPoPP*. ACM, 105–114.

Belady, L. A. 1966. A study of replacement algorithms for a virtual storage computer. *IBM Syst. J. 5,* 2, 78–101.

Blelloch, G. and Chatterjee, S. 1990. Vcode: A data-parallel intermediate language. In *Proceedings of FMPC*. ACM, 471–480.

Boudier, P. and Sellers, G. 2011. Memory system on Fusion APUs. In *Proceedings of the AMD Fusion Developer Summit*. AMD.

Bougé, L. and Levaire, J.-L. 1992. Control structures for data-parallel SIMD languages: Semantics and implementation. *Future Gen. Comput. Syst. 8,* 4, 363–378.

Bouknight, W., Denenberg, S. A., McIntyre, D. E., Randall, J. M., Sameh, A. H., and Slotnick, D. L. 1972. The Illiac IV system. *IEEE 60,* 4, 369–388.

Briggs, P., Cooper, K. D., and Torczon, L. 1992. Rematerialization. In *Proceedings of PLDI*. ACM, 311–321.

Brockmann, K. and Wanka, R. 1997. Efficient oblivious parallel sorting on the MasPar MP-1. Vol. *ICSS. 1*, 200.

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. In *Proceedings of PLDI*. ACM, 25–32.

Carrillo, S., Siegel, J., and Li, X. 2009. A control-structure splitting optimization for GPGPU. In *Proceedings of the 6th ACM Conference Computing Frontiers* . ACM, 147–150.

Cederman, D. and Tsigas, P. 2009. GPU-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics 14,* 1, 4–24.

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 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 44–54.

CHOI, J.-D., CYTRON, R., AND FERRANTE, J. 1991. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of POPL*. ACM, 55–66.

COLLANGE, C ., DEFOUR, D., AND ZHANG, Y. 2009. Dynamic detection of uniform and affine vectors in GPGPU computations. In *Proceedings of the Parallel Processing Workshop (HPPC)*. Lecture Notes in computer science, vol. 6043, Springer, 46–55.

COUSOT, P. AND HALBWACHS, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of POPL*. ACM, 84–96.

COUTINHO, B., SAMPAIO, D., PEREIRA, F. M. Q., AND MEIRA, W. 2011. Divergence analysis and optimizations. In *Proceedings of International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 320–329.

COUTINHO, B., SAMPAIO, D., PEREIRA, F. M. Q., AND MEIRA, W. 2013. Profiling divergences in GPU applications. *Concur. Comput. Pract. Exp. 25,* 6, 775–789.

CUNNINGHAM, D., BORDAWEKAR, R., AND SARASWAT, V. 2011. GPU programming in a high level language: Compiling ×10 to CUDA. In *Proceedings of the ACM SIGPLAN X10 Workshop*. ACM, 8:1–8:10.

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 Comput. 7,* 1, 11–24.

DIAMOS, G., KERR, A., YALAMANCHILI, S., AND CLARK, N. 2010. Ocelot, a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 354–364.

DUBACH, C., CHENG, P., RABBAH, R., BACON, D. F., AND FINK, S. J. 2012. Compiling a high-level language for GPUS: (via language support for architectures and compilers). In *Proceedings of ACM SIGPLAN PLDI*. ACM, 1–12.

FARRELL, C. A. AND KIERONSKA, D. H. 1996. Formal specification of parallel SIMD execution. *Theo. Comp. Science 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. 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 IEEE/ACM MICRO*. IEEE, 407–420.

GARLAND, M. AND KIRK, D. B. 2010. Understanding throughput-oriented architectures. *Comm. ACM 53*, 58–66.

GOU, C. AND GAYDADJIEV, G. 2013. Addressing GPU on-chip shared memory bank conflicts using elastic pipeline. *Int. J. Parallel Program. 41,* 3, 400–429.

GROVER, V., JOANNES, B., AARTS, M., AND MURPHY, M. 2009. Variance analysis for translating CUDA code for execution by a general purpose processor. U.S. Patent 2009/0259997, Filed March 31, 2009, and issued October 15, 2009.

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 GPGPU-4*. ACM, 3:1–3:8.

JANG, B., SCHAA, D., MISTRY, P., AND KAELI, D. 2010. Static memory access pattern analysis on a massively parallel GPU. In *Proceedings of PLDI*. ACM.

KARRENBERG, R. AND HACK, S. 2011. Whole-function vectorization. In *Proceedings of the 9th IEEE/ACM CGO*. IEEE, 141–150.

KARRENBERG, R. AND HACK, S. 2012. Improving performance of OpenCL on CPUS. In *Proceedings of CC*. Springer, 1–20.

KERYELL, R., MATERAT, P., AND PARIS, N. 1991. POMP, or how to design a massively parallel machine with small developments. In *Proceedings of Parallel Architectures and Languages Europe (PARLE)*. Lecture Note in Computer Science, vol. 505, Springer, 83–100.

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*, 1054–1066.

LASHGAR, A. AND BANIASADI, A. 2011. Performance in GPU architectures: Potentials and distances. In *Proceedings of the 9th Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*. IEEE, 75–81.

LAWRIE, D. H., LAYMAN, T., BAER, D., AND RANDAL, J. M. 1975. GLYPNIR-a programming language for ILLIAC IV. *Comm. 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. In *Proceedings of ACM SIGPLAN PPoPP*. ACM, 101–110.

LEE, Y., AVIZIENIS, R., BISHARA, A., XIA, R., LOCKHART, D., BATTEN, C., AND ASANOVIC, K. 2011. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*. ACM, 129–140.

LEE, Y., KRASHINSKY, R., GROVER, V., KECKLER, S. W., AND ASANOVIC, K. 2013. Convergence and scalarization for data-parallel architectures. In *Proceedings of the IEEE/ACM CGO*. ACM, 1–11.

LEIBA, R., HACK, S., AND WALD, I. 2012. Extending a C-like language for portable SIMD programming. In *Proceedings of ACM SIGPLAN PPOPP*. ACM, 65–74.

MENG, J., TARJAN, D., AND SKADRON, K. 2010. Dynamic warp subdivision for integrated branch and memory divergence tolerance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*. ACM, 235–246.

MINÉ, A. 2006. The octagon abstract domain. *Higher Order Symbol. Comput. 19*, 31–100.

MU, S., ZHANG, X., ZHANG, N., LU, J., DENG, Y. S., AND ZHANG, S. 2010. IP routing processing with graphic processors. In *Proceedings of DATE*. IEEE, 93–98.

NICKOLLS, J. AND DALLY, W. J. 2010. The GPU computing era. *IEEE Micro 30,* 2, 56–69.

NICKOLLS, J. AND KIRK, D. 2009. *Graphics and Computing GPUs. Computer Organization and Design, (Patterson and Hennessy)* 4th Ed. Elsevier, Chapter A, A.1–A.77.

NIELSON, F., NIELSON, H. R., AND HANKIN, C. 2005. *Principles of Program Analysis*. Springer.

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. In *Proceedings of ACM SIGPLAN PLDI*. ACM, 257–271.

PEREIRA, F. M. Q. 2011. Dirmap's Blog. http://divmap.wordpress.com/.

PERROT, R. H. 1979. A language for array and vector processors. *ACM Trans. Program. Lang. Syst. 1*, 177–195.

PHARR, M. AND MARK, W. R. 2012. ISPC: A SPMD compiler for high-performance CPU programming. In *Proceedings of Innorative 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. In *Proceedings of POPL*. ACM.

RYOO, S., RODRIGUES, C. I., BAGHSORKHI, S. S., STONE, S. S., KIRK, D. B., AND MEI W. HWU, W. 2008. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of ACM SIGPLAN PPoPP*. ACM, 73–82.

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 $\times$86 platform. In *Proceedings of ACM SIGPLAN PLDI*. ACM, 431–440.

SAMADI, M., HORMATI, A., MEHRARA, M., AND MAHLKE, S. 2012. Adaptive input-aware compilation for graphics engines. In *Proceedings of ACM SIGPLAN PLDI*. ACM.

SAMPAIO, D., MARTINS, R., COLLANGE, C., AND PEREIRA, F. M. Q. 2012a. Divergence analysis with affine constraints. In *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 137–146.

SAMPAIO, D. N., GEDEON, E., PEREIRA, F. M. Q., AND COLLANGE, C. 2012b. Spill code placement for simd machines. In *Proceedings of the 16th Brazilian Symposium on Language Programmings (SBLP)*. Lecture Notes on Computer Science, vol. 7554, Springer, 12–26.

SANDES, E. F. O. AND DE MELO, A. C. M. 2010. CUDALIGN: Using GPU to accelerate the comparison of megabase genomic sequences. In *Proceedings of ACM SIGPLAN PPoPP*. ACM, 137–146.

SCHOLZ, B., ZHANG, C., AND CIFUENTES, C. 2008. User-input dependence analysis via graph reachability. Tech. rep., Sun, Inc.

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 IEEE/ACM CGO*. IEEE, 111–119.

STRATTON, J. A., RODRIGUES, C., SUN, I.-J., OBEID, N., CHANG, L.-W., ANSSARI, N., LIU, G. D., AND MEI W. HWU, W. 2012. The parboil report. Tech. rep., University of Illinois at Urbana-Champaign.

TU, P. AND PADUA, D. 1995. Efficient building and placing of gating functions. In *Proceedings of ACM SIGPLAN PLDI*. ACM, 47–55.

WEISER, M. 1981. Program slicing. In *Proceedings of ICSE*. IEEE, 439–449.

YANG, Y., XIANG, P., KONG, J., AND ZHOU, H. 2010. A GPGPU compiler for memory optimization and parallelism management. In *Proceedings of ACM SIGPLAN PLDI*. ACM, 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)*. ACM, 115–126.

ZHANG, E. Z., JIANG, Y., GUO, Z., TIAN, K., AND SHEN, X. 2011. On-the-fly elimination of dynamic irregularities for GPU computing. In *Proceedings of ASPLOS*. ACM, 369–380.

ZHANG, Y. AND OWENS, J. D. 2011. A quantitative performance analysis model for GPU architectures. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*. ACM, 382–393.

Received December 2012; revised June 2013; accepted August 2013