Divergence Analysis with Affine Constraints

Diogo Sampaio, Rafael Martins, Fernando Magno Quintão Pereira, Caroline Collange, Fernando Magno, Quintão Pereira

Divergence Analysis with Affine Constraints *

Diogo Sampaio Rafael Martins Caroline Collange Fernando Magno Quintão Pereira
Departamento de Ciência da Computação Universidade Federal de Minas Gerais, Brazil
{sampaio,rafaelms,fernando}@dcc.ufmg.br

June 26, 2012

Abstract

The rising popularity of graphics processing units is bringing renewed interest in code optimization techniques for SIMD processors. Many of these optimizations rely on divergence analyses, which classify variables as uniform, if they have the same value on every thread, or divergent, if they might not. This paper introduces a new kind of divergence analysis, that is able to represent variables as affine functions of thread identifiers. We have implemented this analysis in Ocelot, an open source compiler, and use it to analyze a suite of 177 CUDA kernels from well-known benchmarks. We can mark about one fourth of all program variables as affine functions of thread identifiers. In addition to the novel divergence analysis, we also introduce the notion of a divergence aware register allocator. This allocator uses information from our analysis to either rematerialize affine variables, or to move uniform variables to shared memory. As a testimony of its effectiveness, our divergence aware allocator produces GPU code that is 29.70% faster than the code produced by Ocelot's register allocator. Divergence analysis with affine constraints is publicly available in the Ocelot compiler since June/2012.

1 Introduction

Increasing programmability and low hardware cost are boosting the use of graphical processing units (GPU) as tools to run high-performance applications. In these processors, threads are organized in groups, called warps, that execute in lock-step. To better understand the rules that govern threads in the same warp, we can imagine that each warp has simultaneous access to a number of processing units, but uses only one instruction fetcher. As an example, if a warp groups 32 threads together, then it can process simultaneously 32 instances of the same instruction. Regular applications, such as scalar vector multiplication, fare very well in GPUs, because we have the same operation being independently performed on different data. However, divergences may happen in less regular applications whenever threads inside the same warp follow different paths after conditional branches. The branching condition might be true to some threads, and false to others. Given that each warp has access to only one instruction at each time, in face of a divergence some threads will be idle while others execute. Hence, divergences may be a major source of performance degradation – a loss that is hard to overcome. Difficulties happen because finding highly divergent branches burdens the application developer with a tedious task, which requires understanding code that might be large and complex.

In this paper we introduce a new *divergence analysis*, i.e., a technique that identifies variable names holding the same value for all the threads in a warp. We call these variables *uniform*. We improve on previous divergence analyses [1, 2, 3, 4] in non-trivial ways. In Section 3, we show that our analysis finds not only uniform variables, but also variables that are affine functions of thread identifiers. Contrary to Aiken's approach [1], we work on SIMD machines; thus, we handle CUDA and OpenCL programs. By taking affine relations between variables into consideration, we also improve on Stratton's [4], Karrenberg's [3] and Coutinho's [2] techniques.

The problem of discovering uniform variables is important in different ways. Firstly, it helps the compiler to optimize the translation of SIMD languages, such as C for CUDA and OpenCL, to ordinary CPUs. Currently there exist many attempts to compile such languages to ordinary CPUs [5, 3, 4]. Vector instruction sets such as the x86's SSE exten-

*This work was partially supported by CNPq, CAPES and FAPEMIG.

sion do not support divergences natively. Thus, compilers might produce very inefficient code to handle this phenomenon at the software level. This burden can be safely removed from non-divergent branches. Secondly, our analysis enables *divergence aware* code optimizations, such as Coutinho *et al.*'s [2] *branch fusion*, and Zhang *et al.*'s [6] thread reallocation. In this paper, we augment this family of techniques with a *divergence aware register allocator*. 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 optimization, because it allows us to perform register *rematerialization* [7] in SIMD processing elements.

Our novel divergence analysis and register allocator are, since June 2012, distributed under GPL license as part of the Ocelot compiler [5]. We have compiled 177 CUDA kernels from 46 applications taken from the Rodinia [8] and the NVIDIA SDK publicly available benchmarks. In practice our divergence analysis runs in linear time on the number of variables in the source program. The experiments also show that our analysis is more precise than Ocelot's previous divergence analysis, which was based on Coutinho *et al.*'s work [2]. We not only point that about one fourth of the divergent variables are affine functions of thread IDs, but also find 4% more uniform variables. Finally, our divergence aware register allocator is effective: we speedup the code produced by Ocelot's linear scan register allocator by almost 30%.

2 Divergences in one Example

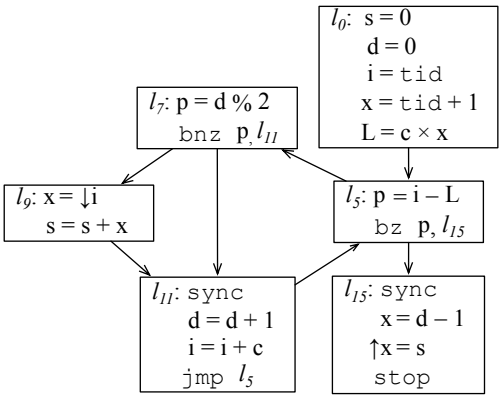
In order to describe our analysis and optimizations, we will be working on top of μ-SIMD, a core SIMD language whose operational semantics has been defined by Coutinho *et al.* [2]. Figure 1 gives the syntax of this language. The execution of a μ-SIMD program consists of a number of processing elements (PE) which execute in lock-step. A program P contains a set of *variable names* V, and each PE has access to a mapping $\theta : V \mapsto \mathbb{N}$. Each PE sees the same set of variable names, yet these names are mapped into different address spaces. The special variable $\mathtt{T}_{id}$, the *thread identifier*, holds a unique value for each PE. An assignment such as $v_1 = v_2 + c$ causes each active PE to compute – simultaneously – the value of $\theta[v_2] + c$, and to use this result to update $\theta[v_1]$. Threads communicate through a *shared memory* Σ, accessed via load and store instructions. For instance, when processing $\uparrow v_x = v$, the active PEs performs the assignment $\Sigma[\theta[v_x]] = v$ simultaneously. Inversely, $v =\downarrow v_x$ updates $\theta[v]$ with the value in $\Sigma[\theta[v_x]]$. The language provides mutual exclusion via the atomic increment $v \overset{a}{\leftarrow} v_x + c$, which, for some arbitrary serialization of the active PEs, reads $\Sigma[\theta[v_x]]$, increments it by c, stores the incremented value back at $\Sigma[\theta[v_x]]$ and uses the modified value to update $\theta[v]$.

Labels	::=	$l \subset \mathbb{N}$
Constants (C)	::=	$c \subset \mathbb{N}$
Variables (V)	::=	$\mathtt{tid} \cup \{v_1, v_2, \ldots\}$
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)	|	$v \overset{a}{\leftarrow} v_x + c$
– (binary addition)	|	$v_1 = o_1 + o_2$
– (binary multiplication)	|	$v_1 = o_1 \times o_2$
– (general binary operation)	|	$v_1 = o_1 \oplus o_2$
– (general unary operation)	|	$v = \oplus o$
– (simple copy)	|	$v = o$
– (synchronization barrier)	|	$\mathtt{sync}$
– (halt execution)	|	$\mathtt{stop}$

Figure 1: The syntax of μ-SIMD instructions.

Figure 2(Top) shows an example of a program written in μ-SIMD that sums up the columns of a triangular matrix. However, only the odd indices in each column contribute to the sum, as we ensure with the test at labels l_7 and l_8. In this program threads perform different amounts of work: the PE that has $\mathtt{T}_{id} = n$ visits $n + 1$ cells. 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 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. Consequently, the threads active inside the loop take the same path at the branch in label l_8.

A conditional test $\mathtt{bnz}\ v, l'$ at label l causes all the threads to evaluate their $\theta[v]$. Those that find $\theta[v] \neq 0$ branch to l', whereas the others fall through the next instruction at $l + 1$. If two threads take different paths, i.e., v is divergent, then we say that the threads *diverge* at l. Figure 2(Bottom) illustrates this phenomenon via a snapshot of the execution of our example program. Our running program contains four threads: $t_0, \ldots, t_3$. When visiting the branch at label l_6 for the second time, the predicate p is 0 for thread t_0, and 1 for the other PEs. In face of this divergence, t_0 is pushed onto a stack of waiting threads,



Cycle	Instruction	t_0	t_1	t_2	t_3
14	$l_5 : p = i - L$	✓	✓	✓	✓
15	$l_6 :$ bz p, l_{15}	✓	✓	✓	✓
16	$l_7 : p = i \ \% \ 2$	•	✓	✓	✓
17	$l_8 :$ bz p, l_{11}	•	✓	✓	✓
	...				
25	$l_6 :$ bz p, l_{15}	•	✓	✓	✓
26	$l_7 : p = i \ \% \ 2$	•	•	✓	✓
27	$l_8 :$ bz p, l_{11}	•	•	✓	✓
	...				
44	$l_5 :$ bz p, l_{15}	•	•	•	✓
45	$l_{15} :$ sync	✓	✓	✓	✓

Figure 2: (Top) A μ-SIMD program. (Bottom) An execution trace of the program. If a thread t executes an instruction at cycle j, we mark the entry (t, j) with the symbol ✓. Otherwise, we mark it with •.

while the other threads continue executing the loop. When the branch is visited a third time, a new divergence will happen, this time causing t_1 to be stacked for later execution. This pattern will happen once again with thread t_2, although we do not show it in Figure 2. Once t_3 leaves the loop, all the threads synchronize via the `sync` instruction at label l_{15}, and resume lock-step execution.

3 The Analysis

Gated Single Static Assignment Form. To increase the precision of our analysis, we convert programs to the *Gated Static Single Assignment* form [9, 10] (GSA). This program representation uses three special instructions: μ, γ and η functions, defined as follows: γ functions represent the joining point of different paths created by an "if-then-else" branch in the source program. The instruction $v = \gamma(p, o_1, o_2)$ denotes $v = o_1$ if p, and $v = o_2$ if $\neg p$; μ functions, which only exist at loop headers, merge initial and loop-carried values. The instruction $v = \mu(o_1, o_2)$ represents the assignment $v = o_1$ in the first iteration of the loop, and $v = o_2$ in the others; η functions represent values that leave a loop. The instruction $v = \eta(p, o)$ denotes the value of o assigned in the last iteration of this loop, which is controlled by the predicate p. Figure 3 shows the GSA version of the program in Figure 2. This format serves us well because it separates variables into different names, which we can classify independently as divergent or uniform, while taking control dependences into consideration.

We use Tu and Padua's [10] almost linear time algorithm to convert a program into GSA form. According to this algorithm, γ and η functions exist at the *post-dominator* of the branch that controls them. A label l_p post-dominates another label l if every path from l to the end of the program goes through l_p. Fung *et al.* [11] have shown that re-converging divergent PEs at the immediate post-dominator of the divergent branch is nearly optimal with respect to maximizing hardware utilization. Thus, we assume that each γ or η function encodes an implicit synchronization barrier, and omit the `sync` instruction from labels where these functions appear. We use Appel's parallel copy semantics [12] to evaluate these functions. For instance, the μ assignment at l_5, in Figure 3 denotes two parallel copies: either we perform $[i_1, \text{sum}_1, d_1] = (i_0, \text{sum}_0, d_0)$, in case we are entering the loop for the first time, or we do $[i_1, \text{sum}_1, d_1] = (i_2, \text{sum}_3, d_2)$ otherwise.

Affine Analysis. The objective of the divergence analysis with affine constraints is to associate with every program variable an *abstract state* which tells us if that variable is uniform, divergent or *affine*, a term that we shall define soon. This abstract state is a point in a lattice A, which is the product of two instances of a simpler lattice C, e.g., $A = C \times C$. We 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 $\wedge$, such that $c_1 \wedge c_2 = \bot$ if $c_1 \neq c_2$, and $c \wedge c = c$. Notice that C is the lattice used in the compiler optimization known as constant propagation; hence, for a proof of monotonicity, see Aho *et al* [13, p.633-635]. 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:

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

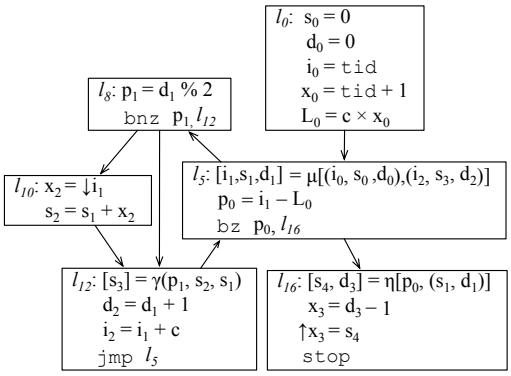


Figure 3: The program from Figure 2 converted into GSA form.

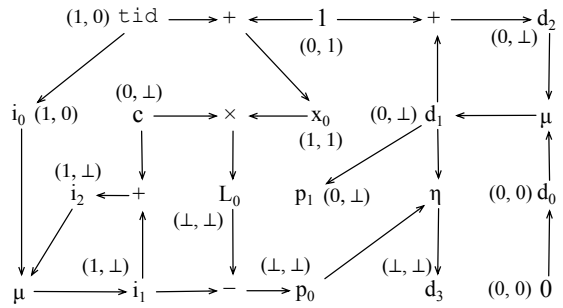


Figure 5: The dependence graph denoting the slice of the program in Figure 3 that produces variable d_3. The figure shows the results of our divergence analysis with affine constraints.

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

Complexity. The constraint system in Figure 4 can be solved in time linear on the size of the program's *dependence graph* [14]. The dependence graph has a node for each program variable. If the constraint that produces $[\![v]\!]$ uses $[\![v']\!]$ as a premise, then the dependence graph contains an edge from v' to v. As an example, Figure 5 shows the program dependence graph that we have extracted from Figure 3. We show only the dependence relations in the program slice that contributes to create variable d_3. Each node in Figure 5 has been augmented with the results of our divergence analysis with affine constraints.

Correctness. Given a μ-SIMD program P, plus a variable $v \in P$, we say that v is *uniform* if every processing element always sees v with the same value at simultaneous execution cycles. On the other hand, if these processing elements see v as the same affine function of their thread identifiers, e.g., $v = c_1 \mathtt{T}_{id} + c_2, c_1, c_2 \in \mathbb{Z}$, then we say that v is *affine*. Otherwise, if v is neither uniform nor affine, then we call it *divergent*. The abstract state of each variable tells us if the variable is *uniform*, *affine* or *divergent*.

Theorem 3.1 *If $[\![v]\!] = 0\mathtt{T}_{id} + a, a \in C$, then v is uniform. If $[\![v]\!] = c\mathtt{T}_{id} + a, a \in C, c \in \mathbb{Z}, c \neq 0$, then v is affine.*

4 Register Allocation

Register allocation is the problem of finding storage location to the values manipulated in a program. Either we place these values in registers or in memory. Values mapped to memory are called *spills*. A modern GPU has many memory levels that the compiler must take into consideration when trying to decide where to place spills. Traditional register allocators, such as the one used in the NVIDIA compiler, or in Ocelot [5], map spills to the *local memory*. This memory is exclusive to each thread, and is located off-chip in all the architectures that we are aware off. We have observed that spilled values that our analysis classifies as uniform or affine can be shared among all the threads in the same warp. This observation is particularly useful in the context of graphics processing units, because they are equipped with a fast-access *shared* memory, which is visible to all the threads in execution. The main advantage of mapping spills to the shared memory is speed. This memory is approximately 100x faster than the local memory [15].

We have developed a set of rewriting rules that can be applied after register allocation, mapping some of the spills to the shared memory. To accommodate the notion of local memory in μ-SIMD, we have augmented its syntax with two new instructions: $v = \Downarrow v_x$ loads the value stored at local memory address v_x into v; $\Uparrow v_x = v$ stores v into the local memory address v_x. The table in Figure 6 shows our rewriting rules. In row (ii) we are simply remapping uniform variables from the local to the shared memory. The affine analysis sometimes lets us perform *constant*

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4: Constraint system used to solve our divergence analysis with affine constraints.

propagation and *rematerialization*. Constant propagation, possibly the most well-known compiler optimization, replaces a variable by the constant that it holds. Row (i) replaces loads of constants by the constant itself, and removes stores of constants. Rematerialization recomputes the value of a spilled variable, whenever possible, instead of moving it to and from memory [7]. Row (iii) shows the rewriting rules that do rematerialization. In this case each thread recalculates a spilled value based on its $\mathtt{T}_{id}$, plus the coefficients of the value, as determined by the affine analysis. Finally, row (iv) combines rematerialization with shared storage. If we spill an affine variable whose highest coefficient we cannot determine, e.g., $[\![v]\!] = c_1\mathtt{T}_{id} + t$, then we move only its unknown component t to shared memory. Given Theorem 3.1, this value is guaranteed to be the same for all the threads. Once we load it back from shared memory, we can combine it with c_1, which the affine analysis determines statically, to recompute the value of v.

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

Figure 6: 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.

Implementation details: Graphics processing units are not exclusively SIMD machines. Rather, a single GPU executes many independant SIMD groups of threads, or warps. Our divergence analysis finds uniform variables per warp. Hence, to implement the divergence aware register allocator, we partition the shared memory among all the warps that might run simultaneously. Due to this partitioning we do

not need to synchronize accesses to the shared memory among different warps. On the other hand, the register allocator requires more space in the shared memory. That is, if the allocator finds out that a given program demands N bytes to accommodate the spilled values, and the target GPU runs up to M warps simultaneously, then this allocator will need $M \times N$ bytes in shared memory.

5 Experimental Evaluation

Every technique described in this paper has been implemented in the Ocelot compiler, and as of June/2012, is part of its official distribution. We run Ocelot on a quad-core AMD Phenom II 925 processor with a 2.8 GHz clock. The same workstation also hosts the GPU that we use to run the kernels: a NVIDIA GTX 570 (Fermi) graphics processing unit.

Benchmarks: We have successfully tested our divergence analysis in all the 177 different CUDA kernels from the Rodinia [8] and NVIDIA SDK 3.1 benchmark suites. These benchmarks give us 31,487 PTX instructions. In this paper, we report numbers to the 40 kernels with the longest running times that our divergence aware register allocator produces. If the kernel already uses too much shared memory, our allocator has no room to place spilled values in that region, and performs no optimization. We have observed this situation in two of the Rodinia benchmarks: `leukocite` and `lud`. The 40 kernels that we show in this paper give us 7,646 PTX instructions and 9,858 variables – in the GSA-form programs – to analyze. The larger number of variables is due to the definitions produced by the η, γ and μ functions used to create the GSA intermediate program representation. We name each kernel with four letters. The first two identify the application name, and the others identify the name of the kernel. The full names are available in our webpage.

Runtime of the divergence analysis with affine constraints: Our divergence analysis with affine constraints took 58.6 msecs to go over all the 177 kernels. Figure 7 compares the analysis runtime with the number of variables per program, considering the 40 chosen kernels only. The top chart measures time in CPU ticks, as given by the `rdtsc` x86 instruction. The coefficient of determination between these two quantities, 0.972, indicates that in practice our analysis is linear on the number of variables in the target program. Figure 7 also compares the runtime of our analysis with the divergent analysis originally present

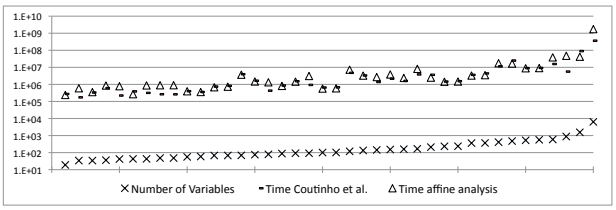


Figure 7: Time, in CPU cycles, to run the divergent analyses compared with the number of variables per kernel in GSA-form. Points in the X-axis are kernels, sorted by the number of variables they contain. We report time of analyses only, excluding other compilation phases.

in the Ocelot distribution. This analysis was implemented after Coutinho *et al.*'s [2], and was available by default in Ocelot until revision 1520, when it was replaced by our algorithm. On the average, our analysis is just 1.39x slower, even though Ocelot's old analysis only marks a variable as uniform or not.

Precision of the divergence analysis with affine constraints: Figure 8 compares the precision of our analysis with the precision of the old divergence analysis of Ocelot. Ocelot's analysis reports that 63.78% of the variables are divergent, while we report 58.81%. However, the old divergence analysis can only mark a variable as uniform or not [2]. On the other hand, our analysis can find that 24.84% of the divergent variables are affine functions of the thread identifier.

Register allocation: Figure 9 compares three different divergence aware register allocators. We use, as a baseline, the linear scan register allocator [16] that is publicly available in the Ocelot distribution, and is not divergence aware. All the four allocators use the same policy to assign variables to registers and to spill variables. The divergence aware allocators are: (DivRA) which moves to shared memory only the variables that Ocelot's old divergence analysis [2] marks as uniform. This allocator can only use the row (*ii*) in Figure 6; (RemRA), which does not use shared memory, but tries to eliminate stores and replace loads by rematerializations of spilled values that are affine functions of $\mathtt{T}_{id}$ with known constants. This allocator uses only rows (*i*) and (*iii*) in Figure 6; (AffRA), which uses all the four rows in Figure 6, and is enabled by this paper's analysis.

Times are taken from the average of 15 runs of each kernel. We take about one and a half hours to execute the 40 benchmarks 15 times on our GTX 570 GPU. Linear Scan and RemRA use nine registers,

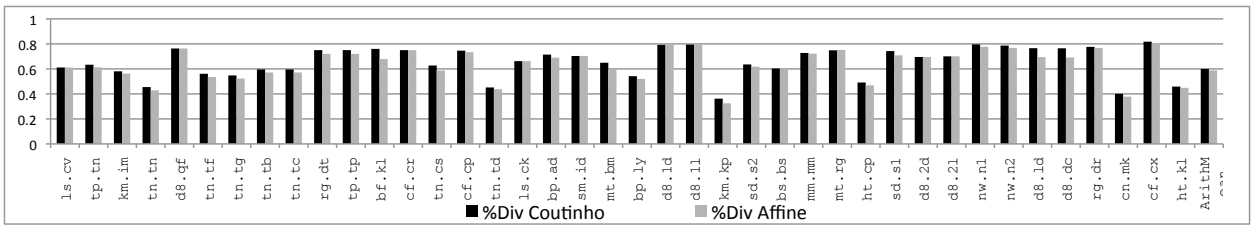


Figure 8: Percentage of divergent variables reported by our divergence analysis with affine constraints and the divergence analysis of Coutinho *et al.* [2]. Kernels are sorted by the number of variables.

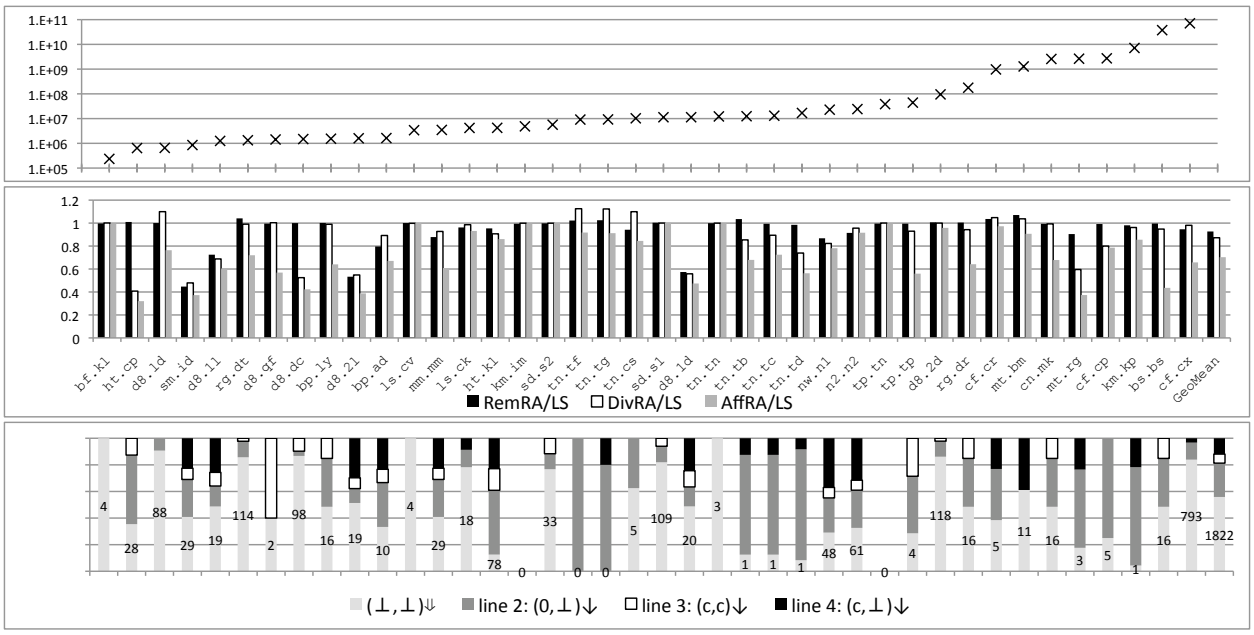


Figure 9: From top to bottom: (i) Runtime of the kernels after register allocation using AffRA. (ii) Relative speedup of different register allocators. Every bar is normalized to the time given by Ocelot's linear scan register allocator. The shorter the bar, the faster the kernel. (iii) Static number of instructions inserted to implement loads of spilled variables. Numbers count loads from local memory that have not been rewritten. SDK's `Kmeans::invert_mapping` (`km.im`) and `Transpose::transpose_naive` did not contain spill code.

whereas DivRA and AffRA use eight, because these two allocators require one register to load the base address that each warp receives in shared memory to place spill code. Each kernel has access to 48KB of shared memory, and 16KB of cache for the local memory. On the average, all the divergence aware register allocators improve on Ocelot's simple linear scan. The code produced by RemRA, which only does rematerialization, is 7.31% faster than the code produced by linear scan. DivRA gives a speedup of 12.75%, and AffRA gives a speedup of 29.70%. These numbers are the geometric mean over the results in Figure 9. There are situations when both DivRA and AffRA produce code that is slower than the Ocelot's linear scan. This fact happens because (i) the local memory has access to a 16KB cache that is as fast as shared memory; (ii) loads and stores after rewriting take three instructions each, according to rule four of Figure 6: 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.

Figure 9(Bottom) shows how often AffRA uses each rewriting pattern in Figure 6, column "Load". The tuples $(0, \bot)\downarrow$, $(c, c)\downarrow$ and $(c, \bot)\downarrow$ refer to lines *ii*, *iii* and *iv* of Figure 6, respectively. We did not find opportunities to rematerialize constants; thus, the rules in the first line of Figure 6 have not been used. $(\bot, \bot) \Downarrow$ represents loads from local memory that have not been rewritten. Overall we have been able to replace 46.8% of all the loads from local memory with more efficient instructions. The pattern $(0, \bot)\downarrow$ replaced 24.1% of the loads, and the pattern $(c, \bot)\downarrow$ replaced 15.3%. The remaining 7.4% modified loads were rematerialized, i.e., they were replaced by the pattern $(c, c)\downarrow$. Stores from local memory have been replaced similarly. For the exact numbers, see [18].

6 Conclusion

This paper has presented the divergence analysis with affine constraints. We believe that this is currently the most precise description of a divergence analysis in the literature. This paper has also introduced the notion of a divergence aware register allocator. We have tested our ideas on a NVIDIA GPU, but these techniques work in any SIMD-like environment. As future work, we plan to improve the reach of our analysis by augmenting it with symbolic constants. We also want to use it as an enabler of other automatic optimizations, such as Coutinho *et al.*'s branch fusion, or Carrillo *et al.*'s [17] branch splitting.

Reproducibility: our code is publicly available in Ocelot, revision 1521, June/2012. All the benchmarks used in this paper are publicly available. For more information about the experiments, see our website at `http://simdopt.wordpress.com`.

Extended version: a more extensive discussion about our work, including the proof of Theorem 3.1, is available in the extended version of this paper [18].

References

[1] A. Aiken and D. Gay, "Barrier inference," in *POPL*. ACM Press, 1998, pp. 342–354.

[2] B. Coutinho, D. Sampaio, F. M. Q. Pereira, and W. M. Jr., "Divergence analysis and optimizations," in *PACT*. IEEE, 2011.

[3] R. Karrenberg and S. Hack, "Whole-function vectorization," in *CGO*, 2011, pp. 141–150.

[4] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu, "Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs," in *CGO*. IEEE, 2010, pp. 111–119.

[5] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot, a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *PACT*, 2010, pp. 354–364.

[6] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, "On-the-fly elimination of dynamic irregularities for GPU computing," in *ASPLOS*. ACM, 2011, pp. 369–380.

[7] P. Briggs, K. D. Cooper, and L. Torczon, "Rematerialization," in *PLDI*. ACM, 1992, pp. 311–321.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*. IEEE, 2009, pp. 44–54.

[9] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe, "The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages," in *PLDI*. ACM, 1990, pp. 257–271.

[10] P. Tu and D. Padua, "Efficient building and placing of gating functions," in *PLDI*. ACM, 1995, pp. 47–55.

[11] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *MICRO*. IEEE, 2007, pp. 407–420.

[12] A. W. Appel, "SSA is functional programming," *SIGPLAN Notices*, vol. 33, no. 4, pp. 17–20, 1998.

[13] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[14] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 1999.

[15] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using cuda," in *PPoPP*. ACM, 2008, pp. 73–82.

[16] M. Poletto and V. Sarkar, "Linear scan register allocation," *TOPLAS*, vol. 21, no. 5, pp. 895–913, 1999.

[17] S. Carrillo, J. Siegel, and X. Li, "A control-structure splitting optimization for GPGPU," in *Computing frontiers*. ACM, 2009, pp. 147–150.

[18] D. Sampaio, R. Martins, S. Collange, and F. M. Q. Pereira, "Divergence analysis with affine constraints," École normale supérieure de Lyon, Tech. Rep., 2011.