Divergence Analysis with Affine Constraints
Diogo Sampaio, Rafael Martins, Sylvain Collange, Fernando Magno Quintão Pereira

To cite this version:
Diogo Sampaio, Rafael Martins, Sylvain Collange, Fernando Magno Quintão Pereira. Divergence Analysis with Affine Constraints. 2011. <hal-00650235v1>

HAL Id: hal-00650235
https://hal.archives-ouvertes.fr/hal-00650235v1
Submitted on 9 Dec 2011 (v1), last revised 20 Nov 2012 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Divergence Analysis with Affine Constraints

Diogo Sampaio  Rafael Martins
Fernando Magno Quintão Pereira
UFMG – 6627 António Carlos Av, 31.270-010, Belo Horizonte, Brazil
{sampaio,rafaelms,fperera}@dcc.ufmg.br

Sylvain Collange
ENS Lyon – 46 allée d’Italie, 69364 Lyon, France
{sylvain.collange}@ens-lyon.fr

Abstract
The rise of graphics processing units in high-performance computing is bringing renewed interest in code optimization techniques that target 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 our divergence analysis with affine constraints on top of Ocelot, an open source compiler, and use it to analyze a suite of 177 CUDA kernels from well-known benchmarks. These experiments show that our algorithm reports 4% less divergent variables than the previous state-of-the-art algorithm of Coutinho et al. Furthermore, we can mark about one fourth of all divergent 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.

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 CUDA¹, AMD APP² and OpenCL³. Running general purpose programs in GPUs is attractive because these processors are 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. Such a hardware has allowed the development of high performance algorithms to solve problems as diverse as sorting [10], gene sequencing [37], IP routing [28] and program analysis [34]. In many cases, these applications out-perform the equivalent CPU program by factors of over 100x [35]. This trend is likely to continue, as upcoming hardware more closely integrates GPUs and CPUs [38], and new models of heterogeneous hardware are introduced [36].

GPUs are highly parallel; however, due to its restrictive programming model, not every application can benefit from all their processing power. In these processors, threads are organized in groups that execute in lock-step. Such groups are called warps in the NVIDIA jargon, or wavefronts in ATI’s. 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, the GeForce GTX 590 has 32 Streaming Multiprocessors, and each of them can run 48 warps of 32 threads. Thus, each warp might perform 32 instances of the same instruction in two cycles of the hardware pipeline. 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.

These divergences happen when threads inside the same warp follow different paths after processing the same branch. 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 have to wait, idly, while others execute. Hence, divergences may be a major source of performance degradation. As an example, Baghsorkhi et al. [4] have analytically showed that approximately one third of the execution time of the prefix scan benchmark [25], included in the CUDA software development kit (SDK), is lost due to divergences. Optimizing an application to avoid divergences is problematic for a number of reasons. First, some parallel algorithms are inherently divergent; thus, threads will naturally disagree on the outcome of branches. Second, finding highly divergent branches burdens the application developer with a tedious task, which requires a deep understanding of code that might be large and complex.

In this paper we present a static program analysis that identifies variables that hold the same value for all the threads in the same warp. We call these variables uniform. We know about a number of previous attempts to develop this kind of divergence analysis [2, 15, 26, 39]; however, we improve on all of them in non-trivial ways. As we will show in Section 4, we have created a static analysis that finds not only uniform variables, but also variables which are affine functions [8] of the identifier of a processing element. Contrary to Aiken’s approach [2], we work on SIMD machines; thus, we handle CUDA and OpenCL programs. By taking control dependences into consideration, we also improve on Stratton et al.’s [39] and Karrenberg’s [26] techniques.

The problem of discovering uniform variables is important in different ways. Firstly, it helps the compiler to optimize the translation of “SIMD” languages to ordinary CPUs. We call SIMD lan-

¹ See The CUDA Programming Guide, 1.1.1
² See AMD APP Guide
³ See The OpenCL Specification, 1.0
languages those programming languages, such as C for CUDA and OpenCL, that are equipped with abstractions to handle diversences. Currently there exist many attempts to compile such languages to ordinary CPUs [18, 26, 39]. Vectorial operations found in traditional architectures, such as the x86’s SSE extension, do not support diversences 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 non-divergent branches that we identify. Furthermore, the divergence analysis provides insights about memory access patterns [8]. 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 generate efficient code for vectorial instruction sets that do not support fast memory gather and scatter [18].

Secondly, in order to more precisely identify diversences, a common strategy is to use instrumentation based profilers. However, this approach may slowdown the target program by factors of over 1500x [14]! Our divergence analysis reduces the amount of branches that the profiler must instrument; hence, decreasing its overhead. Thirdly, the divergence analysis improves static performance prediction techniques used in SIMD architectures [4, 44]. 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 [15] branch fusion, and Zhang et al.’s [43] thread reallocation strategy. In this paper, we augment this family of techniques with a divergence aware register allocator. As we will show in Section 5, 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 enables us to perform a form of rematerialization [6] of values among SIMD processing elements.

We have implemented our analysis and optimizations in the Ocelot [18] open source optimizer, which optimizes PTX, the intermediate program representation used by NVIDIA’s GPUs. We have compiled 177 CUDA kernels from 46 applications taken from the Rodinia [11] and the NVIDIA SDK publicly available benchmarks. The experimental results given in Section 6 show that our implementation of the divergence analysis runs in linear time on the number of variables in the source program. Furthermore, these experiments indicate that our analysis is more precise than Coutinho et al.’s previous state-of-the-art algorithm. We not only find 4% less divergent variables, but also point that about one fourth of the divergent variables are affine functions of some thread identifier. Finally, our divergence aware register allocator is effective: by rematerializing affine values, or moving uniform values to the GPU’s shared memory, we have been able to speedup the code produced by Ocelot’s linear scan register allocator by almost 30%.

2. Background

A modern graphics processing unit usually provides to developers a large number of threads, which are arranged in small groups that we call warps, following NVIDIA’s nomenclature. Different warps execute independently of each other, following Deraea’s Single Program Multiple Data (SPMD) execution model [17]. On the other hand, the threads inside the same warp execute in lock-step, fitting Flynn’s Single Instruction Multiple Data (SIMD) machines [20]. This combination of SPMD and SIMD semantics is one of the characteristics of the so called Single Instruction Multiple Threads (SIMT) execution model [22, 29, 30]. In this paper we will focus on the SIMD characteristics of a typical GPU, because diversences are relevant only at this level.

We will use the two artificial programs in Figure 1 to explain the notion of diversences. 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 (PE), according to the SIMD semantics. All the processing elements see the same set of variables 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, e.g., 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 instance. For this discussion, just the understanding that a thread has a unique identifier is enough. In the rest of this paper we will denote this unique thread identifier by T_{id}.

Each processing element uses its unique identifier to find the data that it must process. Thus, in the kernel avgSquare each thread T_{id} is in charge of summing up the elements of the T_{id}th column of m. Once leaving the loop, this PE will store the average of the sum in v[T_{id}]. Notice that, following usual coding practices we represent the matrix in a linear format. In this program, diversences will not happen. That is, if we assume a SIMD semantics, then each thread will iterate through the loop the same number of times. Consequently, upon leaving the loop every thread sees the same value at its image of variable d.

Kernel sumTriangle presents a very different behavior. This 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 T_{id} = n will visit n + 1 cells of m. After a thread leaves

---

\[1\] _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 L = T_{id} + c - 1;
\[6\] for (int i = T_{id}; i < L; i += c) {
\[7\] sum += m[i];
\[8\] d += 1;
\[9\] }
\[10\] v[tid] = sum / d;
\[11\] }
\[12\]}

\[2\] _global_ void sumTriangle(float* n, float* v, int c) {
\[3\] if (T_{id} < c) {
\[4\] int d = 0;
\[5\] float sum = 0.0F;
\[6\] int L = (T_{id} + 1) * c;
\[7\] for (int i = T_{id}; i < L; i += c) {
\[8\] sum += m[i];
\[9\] }
\[10\] v[d] = sum;
\[11\] }
\[12\]}

**Figure 1.** Two kernels written in C for CUDA. The gray lines in the right show the parts of matrix m processed by each thread. Dots mark the cells that add up to the sum in line 8 of sumTriangle.
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 \( T_{id} + 1 \) 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. Hence, 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.

**Previous Divergence Analyses.** Several algorithms have been proposed before to find uniform variables. The first technique that we are aware of is the barrier inference of Aiken and Gay [2]. This method, designed to the SPMD execution model, finds a conservative set of uniform \(^5\) variables via static analysis. However, because it is tied to a SPMD machine, Aiken and Gay’s algorithm can only report uniform variables at global synchronization points.

The recent interest on graphics processing units has given a renewed impulse to this type of analysis, in particular with a focus on SIMD machines. For instance, Stratton et al.’s [39] variance analysis, and Karrenberg et al.’s vectorization analysis distinguish uniform and divergent variables. However, these analysis are tools used in the compilation of SPMD programs to CPUs with explicit SIMD instructions, and both compilers generate specific instructions to manage divergences at runtime. On the other hand, we focus on architectures in which divergences are managed implicitly by the hardware. A naive application of Stratton’s and Karrenberg’s analyses in our static context may yield false negatives due to control dependences, although they work well in the dynamic scenario for which they were designed. For instance, Karrenberg’s select and loop-blending functions are similar to the GSA-form \( \gamma \) and \( \eta \) functions that we use. However, an important difference is that select and blend are concrete instructions emitted during code generation, while our GSA functions are abstractions used statically.

Coutinho et al. [15] have proposed a version of the divergence analysis that works in programs in the Gated Static Single Assignment (GSA) format [32, 40]. Thus, Coutinho et al.’s analysis is precise enough to find out that variable \( d \) is uniform inside the loop in the kernel `sumTriangle`, and divergent outside. Nevertheless, their version is over-conservative because it does not consider affine relations between variables. For instance, in the kernel `avgSquare`, variables \( i \) and \( L \) are functions of \( T_{id} \). However, if we inspect the execution trace of this program, then we will notice that the comparison \( i < L \) always has the same value for every thread. This fact happens because both variables are functions of two affine expressions of \( T_{id} \), whose combination cancel the \( T_{id} \) factor out, e.g.: \( L = T_{id} + c_1 \) and \( i = T_{id} + c_2 \); thus, \( L - i = (1 - 1)T_{id} + c_1 - c_2 \). Therefore, a more precise divergence analysis requires some way to take affine relations between variables into consideration.

Figure 2 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 dependences into consideration, but not control dependences. 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 \( d \) as uniform. Notice that, because Coutinho et al.’s divergence analysis, as well as ours, use the GSA intermediate representation, we distinguish the live ranges of variable \( d \) inside (\( d_{\text{in}} \)) and outside (\( d_{\text{out}} \)) the loops.

**Divergence Optimizations.** We call divergence optimizations the compiler optimization techniques that use the results of divergence analysis to generate better code. Many recent works describe optimizations in this category. As an example, the PTX programmer’s manual \(^6\) recommends replacing ordinary branch instructions (bra) proved to be non-divergent by special instructions (bra.una), which are supposed to divert control to the same place for every active thread. Other examples of divergence optimizations include branch fusion, thread reallocation, branch distribution, iteration delaying and work unification. Work unification, for instance, consists in leaving to only one thread the task of computing non-divergent variables; hence, reducing memory accesses and hardware occupancy. In this line, Collange et al. [13] have designed a hardware mechanism that dynamically detects non-divergent variables and assigns their computation to a single thread.

Branch distribution [24] is a form of code hoisting that works both at the prolog and at the epilogue of a branch. The objective of this optimization is to merge code inside program paths that might be executed by potentially divergent threads. Branch fusion [15] is a generalization of branch distribution. This optimization finds the longest chains of common instructions between two divergent paths and joins them. 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, in such a way that only one, or just a few, warps will contain divergent threads. It has been implemented at the software level by Zhang et al. [42, 43], and simulated at the hardware level by Fung et al. [21].

A number of compiler optimizations try to rearrange loops in order to mitigate the impact of divergences. Carrillo et al. [9] have proposed *branch splitting*, a code transformation that splits a parallelizable loop enclosing a multi-path branch into multiple loops, each containing only one branch. Lee et al. [27] 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 et al. [24] have generalized Lee’s approach proposing *iteration delaying*, a method that regroups loop iterations, executing those that take the same branch direction together.

\(^5\) Aiken and Gay would call these variables *single-valued*

\(^6\) PTX programmer’s manual, 2008-10-17, SP-03483-001_v1.3, ISA 1.3

---

**Table 2.** A comparison between different versions of divergence analyses. We use U for uniform and D for Divergent variables. Karrenberg’s analysis can mark variables in the format \( 1 \times T_{id} + c, c \in \mathbb{N} \) as consecutive (c) or consecutive aligned (ca).

<table>
<thead>
<tr>
<th>Data Dep.</th>
<th>Aiken</th>
<th>Karr.</th>
<th>Cout.</th>
<th>Affine degree 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only</td>
<td>[2]</td>
<td>[26]</td>
<td>[15]</td>
<td>[Section 4]</td>
</tr>
<tr>
<td>c</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>( 0T_{id}^2 + 0T_{id} + \perp )</td>
</tr>
<tr>
<td>m</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>( 0T_{id}^2 + 0T_{id} + \perp )</td>
</tr>
<tr>
<td>v</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>( 0T_{id}^2 + 0T_{id} + \perp )</td>
</tr>
<tr>
<td>i</td>
<td>D</td>
<td>D</td>
<td>ca</td>
<td>D</td>
</tr>
<tr>
<td>avgSquare</td>
<td></td>
<td></td>
<td></td>
<td>( 0T_{id}^2 + T_{id} + \perp )</td>
</tr>
<tr>
<td>( d_{\text{in}} )</td>
<td></td>
<td></td>
<td></td>
<td>( 0T_{id}^2 + 0T_{id} + \perp )</td>
</tr>
<tr>
<td>( d_{\text{out}} )</td>
<td></td>
<td></td>
<td></td>
<td>( 0T_{id}^2 + 0T_{id} + \perp )</td>
</tr>
<tr>
<td>( \text{sumTriangle} )</td>
<td></td>
<td></td>
<td></td>
<td>( 0T_{id}^2 + \perp T_{id} + \perp )</td>
</tr>
<tr>
<td>( d_{\text{in}} )</td>
<td></td>
<td></td>
<td></td>
<td>( 0T_{id}^2 + 0T_{id} + \perp )</td>
</tr>
<tr>
<td>( d_{\text{out}} )</td>
<td></td>
<td></td>
<td></td>
<td>( 0T_{id}^2 + 0T_{id} + \perp )</td>
</tr>
</tbody>
</table>

---

Divergence Analysis with Affine Constraints

3 2011/12/9
3. The Intermediate Program Representation

In order to describe our analysis and optimizations, we will be working on top of µ-SIMD, a core SIMD language whose semantics has been defined by Coutinho et al. [15], and which we briefly review for the sake of completeness. Figure 3 gives the syntax of this language; for its operational semantics, see Coutinho et al. Informally, 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 possibly different — mapping $\theta : V \rightarrow \mathbb{N}$. The µ-SIMD instructions manipulate operands ($o$), which can be either variables ($v$) or constants ($c$). 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 $\theta[v_1]$. In µ-SIMD, different threads communicate through a shared memory $\Sigma$, accessed via load and store instructions. For instance, when processing $v_i = v$, each active PE performs the assignment $\Sigma[\theta[v_x]] = v$ simultaneously. Inversely, $v_i = v_x$ updates $\theta[v]$ with the value in $\Sigma[\theta[v_x]]$. The language provides mutual exclusion via the atomic increment $v \leftarrow v + c$, 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 $\theta[v]$. The semantics of branches and synchronization instructions is more elaborate, and we will describe it informally in the next example.

| Labels                                      | ::= | $l \in \mathbb{N}$ |
| Constants ($C$)                             | ::= | $c \in \mathbb{N}$ |
| Variables ($V$)                             | ::= | $\text{tid} \cup \{v_1, v_2, \ldots\}$ |
| Operands ($V \cup C$)                       | ::= | $\{o_1, o_2, \ldots\}$ |
| Instructions                                | ::= | \textbf{bz}/\textbf{bnz} \textbf{v}, \textbf{l} |
|                                           | ::= | \textbf{jump} \textbf{l} |
|                                           | ::= | $\uparrow v_x = v$ |
|                                           | ::= | $v = \downarrow v_x$ |
|                                           | ::= | $v = v + c$ |
|                                           | ::= | $v_1 = o_1 \times o_2$ |
|                                           | ::= | $v = \oplus o$ |
|                                           | ::= | $v = o$ |
|                                           | ::= | \textbf{sync} |
|                                           | ::= | \textbf{stop} |

**Figure 3.** The syntax of µ-SIMD instructions.

An Example of Divergent Behavior. A running instance of a µ-SIMD program consists of a set of threads $\{t_0, t_1, \ldots, t_{n-1}\}$ that execute in lock-step. A conditional test \textbf{bnz} $v$; $l'$ at label $l$ causes all the threads to evaluate their $\theta[v]$. Those that find $\theta[v] \neq 0$ will branch to $l'$, whereas the others will fall through the next instruction at label $l + 1$. If two threads take different control flow decisions, i.e., $v$ is not uniform, then we say that the threads diverge at $l$. Figure 5 illustrates this phenomenon by showing a snapshot of the execution of the kernel \texttt{sumTriangle}, seen in Figure 1. We have converted this function to the control flow graph given in Figure 4, which is written in µ-SIMD syntax. Each instruction has a label, but for readability, we only show the first label of each basic block. As we see in Figure 5, we assume that our running program contains the four threads: $t_0, \ldots, t_3$. When visiting the branch at label $l_9$ 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, 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.

![Diagram of the Intermediate Program Representation](image)

**Figure 4.** The \texttt{sumTriangle} kernel written in µ-SIMD syntax.

The Gated Single Static Assignment Form. In order to better handle control dependences between program variables, we will be working with µ-SIMD programs in Gated Single Static Assignment form [32, 40] (GSA). This intermediate program representation differs from the well-known Static Single Assignment [16] form because it augments $\phi$-functions with the predicates that control them. The GSA form uses three special instructions: $\mu$, $\gamma$ and $\eta$ functions, which Ottenstein et al. [32] define as follows:

**Figure 5.** A snapshot of the execution trace of the µ-SIMD program given in Figure 4. If a thread $t$ executes an instruction at a cycle $j$, we mark the entry $(t, j)$ with the symbol $\checkmark$. Otherwise, we mark it with the symbol $\cdot$.

This pattern will happen once again with thread $t_2$, although we do not show it in Figure 5. Once $t_3$ leaves the loop, all the threads synchronize via the \texttt{sync} instruction at label $l_{15}$, and resume lock-step execution. Notice that the way in which this idle waiting is implemented is immaterial to our discussion. It can be formalized via a stack, as in Coutinho et al. [15], or by execution with no-write-back, as done by Bougé et al. [5] and Farrell et al. [19].
γ functions represent the joining point of different paths created by an “if-then-else” branch in the source program. The instruction \( v = γ(p, o_1, o_2) \) denotes \( v = o_1 \) if \( p \), and \( v = o_2 \) if \( ¬p \).

• \( μ \) functions, which only exist at loop headers, merge initial and loop-carried values. The instruction \( v = μ(o_1, o_2) \) 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 \( v = η(p, o) \) denotes the value of \( o \) assigned in the last iteration of this loop, which is controlled by the predicate \( p \).

Figure 6 shows the program in Figure 5 converted to GSA form. In our implementation we use Coutinho et al.’s [15] version of Tu and Padua’s [40] 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, and only if, every path from \( l \) to the end of the program goes through \( l_p \). Fung et al. [21] 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. Although Fung et al. have discovered situations in which it is better to do this re-convergence past \( l_p \), they are very rare. Thus, we assume that each \( γ \) or \( η \) function encodes an implicit synchronization barrier, and omit the sync instruction from labels where any of these functions is present. As we see in Figure 6, the special functions are placed at the beginning of basic blocks. We use Appel’s parallel copy semantics [3] to evaluate these functions, and we denote these parallel copies using Hack’s matrix notation [23]. For instance, the \( μ \) assignment at \( l_5 \), in Figure 6 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 \( \{ i_1, \text{sum}_1, d_1 \} = \{ i_2, \text{sum}_2, d_2 \} \) otherwise.

\[
\begin{array}{c|c|c|c}
\text{A} & \top & c_1 & \bot \\
\top & \top & c_1 & \bot \\
c_2 & c_2 & c_2 \land c_2 & \bot \\
\bot & \bot & \bot & \bot \\
\end{array}
\]

Figure 7. Abstract semantics of the meet and multiplication operators used in our divergence analysis. We let \( c_1 \in \mathbb{Z} \).

Upon definition, as we shall see in Figure 8, every variable receives an abstract value different from \( \top \).

We let \( c_1 \land c_2 = \bot \) if \( c_1 \neq c_2 \), and \( c \land c = c \) otherwise. Similarly, we let \( c + \bot = \bot + c = \bot \). Notice that \( C \) is the lattice normally used to implement constant propagation; hence, for a proof of monotonicity, see Aho et al. [1], 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 \( a_1 \, T_{id} + a_2 \). We define the meet operator of \( A \) as follows:

\[
(a_1 \, T_{id} + a_2) \land (a_1' \, T_{id} + a_2') = (a_1 \land a_1') \, T_{id} + (a_2 + a_2')
\]

We let the constraint variable \( [v] = a_1 \, T_{id} + a_2 \) denote the abstract state associated to variable \( v \). We determine the set of divergent variables in a \( \mu \)-SIMD program \( P \) via a forward-must dataflow analysis whose transfer functions are given by the inference rules in Figure 8. Initially we let \( [v] = (\top, \top) \) for every variable \( v \) defined in the text of \( P \), and \( [c] = (0, c) \) for each \( c \in \mathbb{Z} \).

Sparse Implementation. If we see the inference rules in Figure 8 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 8 reveals. Therefore, our analysis admits a sparse implementation, as defined by Choi et al. [12]. 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 our underline 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 [31]. Each iteration is linear on the size of the dependence graph, which might be quadratic on the size of program variables, if we allow \( γ \) and \( μ \) functions to have any number of parameters. Nevertheless, as we show in Section 6, our analysis is linear in practice. As an illustration, Figure 9 shows the program dependence graph that we have extracted from Figure 6. We show only the program slice [41] that creates variable \( d_4 \). Each node has been augmented with the results of our divergence analysis with affine constraints.

Correctness. Given a \( \mu \)-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., \( v = c_1 \, T_{id} + c_2, c_1, c_2 \in \mathbb{Z} \), 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 4.1.** If \( [v] = c \, T_{id} + a, a \in \mathbb{C}, \) then \( v \) is uniform. If \( [v] = \top \), then \( v \) is affine.

**proof:** The proof is by induction on the rules in Figure 8. We will show a few cases:
Divergence Analysis with Affine Constraints

\[ v = c \times T_{id} \quad [\text{Tid}] \quad \boxed{[v] = cT_{id} + 0} \]
\[ v \leftarrow v_x + c \quad [\text{ATM}] \quad \boxed{[v] = v_x + c} \]
\[ v = \oplus o \quad [\text{Guz}] \quad \frac{[o]}{[v]} = 0T_{id} + a \]
\[ v = \downarrow v_x \quad [\text{Ldu}] \quad \boxed{[v_x] = 0T_{id} + a} \]
\[ v = \gamma[p, o_1, o_2] \quad [\text{Gam}] \quad \boxed{[p] = 0T_{id} + a} \]
\[ v = o_1 + o_2 \quad [\text{Sum}] \]
\[ v = o_1 \times o_2 \quad [\text{Mlv}] \]
\[ v = o_1 \oplus o_2 \quad [\text{Mlc}] \]
\[ v = \gamma[p, o_1, o_2] \quad [\text{Pdv}] \]
\[ v = \mu[o_1, \ldots, o_n] \quad [\text{Rmu}] \]

- **CNT**: a variable initialized with a constant is uniform by definition, and the rule makes this case explicit.
- **SUM**: if the hypothesis holds by induction, then we have two cases to consider. (i) If \( v_1 \) and \( v_2 \) are uniform, then \( \boxed{[v_1] = 0T_{id} + a_1} \) and \( \boxed{[v_2] = 0T_{id} + a_2} \), where \( a_1, a_2 \in C \). Thus, \( \boxed{[v] = 0T_{id} + (a_1 + a_2)} \). 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 \( \boxed{[v_1] = c_1T_{id} + a_1} \), and \( \boxed{[v_2] = c_2T_{id} + a_2} \), where \( c_1, c_2 \in \mathbb{Z} \) and \( a_1, a_2 \in C \). Thus, \( \boxed{[v] = (c_1 + c_2)T_{id} + (a_1 + a_2)} \), and the result holds for the same reasons as in (i).

The other rules are similar. \( \square \)

Our divergence analysis with affine constraints subsumes the divergence analysis of Coutinho et al. [15], as Corollary 4.2 shows.

**Corollary 4.2.** If the divergence analysis of Coutinho et al. says that variable \( v \) is uniform, then our analysis says that \( v \) is uniform.

**Proof:** Because we use the same intermediate representation as Coutinho et al., both analyses work on the same program dependence graph. In Coutinho et al.’s analysis, \( v \) is non-divergent if it is a function of only non-divergent variables, e.g., \( v = f(v_1, \ldots, v_k) \), and every \( v_i \), \( 1 \leq i \leq n \) is non-divergent. From Theorem 4.1, we know that if \( \boxed{[v_i] = 0T_{id} + c_i} \) for every \( i, 1 \leq i \leq n \), then \( v \) is uniform. \( \square \)

Figure 8. Constraint system used to solve our divergence analysis with affine constraints of degree 1.

Figure 9. The dependence graph denoting the slice of the program in Figure 6 that produces variable \( d_2 \). The figure shows the results of our divergence analysis with affine constraints.
From Affine to Polynomial Functions. Our analysis, as well as constant propagation, are special cases of a more extensive framework which we call the divergence analysis with polynomial constraints. In the general case, we let \( \begin{bmatrix} \mathbb{v}\end{bmatrix} = a_2T_{id}^2 + a_1T_{id} + a_0 \), where \( a_i \in \mathbb{C}, 1 \leq i \leq n \). Addition and multiplication of polynomials follow the usual algebraic rules. The rules in Figure 8 use polynomials of degree one. Constant propagation uses polynomials of degree zero.

In our benchmarks we did not find situations where polynomials of degree three or higher could be useful. However, polynomials of degree two improve the way we handle parameters of functions. The extra precision comes out of Theorem 4.3. Consider, for instance, the program in Figure 10, which assigns to each processing element the task of initializing the rows of a matrix \( \mathbb{m} \) with one’s. The degree one divergence analysis would conclude that variables \( t_0, t_1 \) and \( t_2 \) 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 \( T_{id} \).

**Theorem 4.3.** If \( \begin{bmatrix} \mathbb{v}\end{bmatrix} = 0T_{id}^2 + a_1T_{id} + a_0 \), then \( \mathbb{v} \) is an affine function of \( T_{id} \).

**proof:** Induction plus case analysis on the extended constraint rules for polynomials of degree two. \( \square \)

5. Divergence Aware Register Allocation

Register allocation is the problem of finding storage locations to the program variables. Usually these values might reside in registers or in memory. Keeping them in registers is preferable, because these are faster storage units. On the other hands, computer architectures tend to afford a few registers, whereas the main memory, normally much slower, is generally abundant. The values that the allocator sends to memory are called spills. In the case of SIMD machines, the register allocator has a third variable to play with: If processing elements share a common memory area – the shared memory – then the allocator has the opportunity to keep a single image of any uniform spilled value in that region. This optimization is particularly beneficial to graphics processing units, because in these architectures the shared memory usually has faster access times than the memory used for register spills. A typical example of this kind of memory hierarchy is found in the NVIDIA GeForce 8800 graphics processing units. According to Ryoo et al. [35], this hardware has, among others, two types of memory: shared and local. The shared memory has approximately the same latency as an on-chip register access, whereas the local memory is 200-300 times slower.

A divergence aware register allocator has a second advantage: it tends to improve cache utilization. The more recent Fermi GPU architecture provides on-chip caches that mitigate the cost of register spills to local memory. However, 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 [29]. When moving non-divergent variables to the shared memory, we only need to store one instance per warp, rather than one instance per thread. Thus, the divergence aware register allocator may provide up to a 32-fold improvement in cache locality.

A straightforward implementation of a register allocator will place spilled values into the local memory, which is exclusive to each processing element. In this way, there is no possibility of a thread to overwrite values of another. In order to accommodate the notion of local memory in \( \mu \)-SIMD programs, we augment the syntax of \( \mu \)-SIMD with two instructions to manipulate this memory. An instruction such as \( v = \llbracket v \rrbracket \) denotes a load of the value stored at local memory address \( v \) into \( v \). The instruction \( v \) stores a register memory address \( v \).

In addition to moving uniform values to shared memory, in this paper we propose a form of Briggs’s style rematerialization [6] that suits SIMD machines. The table in Figure 11 shows how we replace loads and stores to the local memory by more efficient instructions. The lattice that we use in Figure 8 is equivalent to the lattice used by Briggs et al. in their rematerialization algorithm.

**Figure 10.** An example where a higher degree polynomial improves the precision of the simple affine analysis. We let \( a_2T_{id}^2 + a_1T_{id} + a_0 = \{a_2, a_1, a_0\} \).

**Figure 11.** Rewriting rules that replace loads (\( \downarrow v = \llbracket v \rrbracket \)) and stores (\( \uparrow v = v \)) to the local memory with faster sequences of instructions.

### Implementation details:

As discussed in Section 2, graphics processing units are not exclusively SIMD machines. Rather, they combine into a single card many SIMD units, or warps. Our divergence analysis finds uniform variables per warp. Therefore, in order to implement the divergence aware register allocator, we must partition the shared memory among all the warps that might run simultaneously. The main advantage of this partitioning is that we do not need to synchronize accesses to the shared memory among different warps. On the other hand, the register allocator requires more space in the shared memory. That is, if the allocator finds out that a given program demands \( N \) bytes to accommodate the spilled values, and the target GPU runs up to \( M \) warps simultaneously, then this allocator will need \( M \times N \) bytes in shared memory.

6. Experiments

We have implemented our divergence analysis plus the divergence aware register allocator on top of the Ocelot [18] open source compiler, revision 1560 of November/2011. We run Ocelot on a quad-core AMD Phenom II 925 processor. Each core has a 2.8GHz
CPU clock. This CPU also hosts the GPU that we use to run the kernels: a NVIDIA GTX 570 (Fermi) graphics processing unit.

**Benchmarks:** We have successfully tested our divergence analysis in all the 177 different CUDA kernels that we took from the Rodinia [11] and NVIDIA SDK 3.1 benchmark suites. These benchmarks give us 31,487 PTX instructions. In this paper, we chose to report numbers to the 40 kernels with the longest running times that our divergence aware register allocator can optimize. 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 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 $\eta$, $\gamma$ and $\mu$ functions used to create the GSA intermediate program representation. Henceforth we will 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 anonymous repository.

**Runtime of the divergence analysis with affine constraints:** Our divergence analysis with affine constraints of degree 2 took 58.6 msecs to go over all the 177 kernels. Figure 12 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. We see a strong correlation between these two quantities, indicating that in practice our analysis is linear on the number of variables in the target program. Figure 12 also compares the runtime of our analysis with that of the divergence analysis in the Ocelot’s distribution, which is an implementation of Coutinho et al.’s algorithm. On the average, our analysis is 1.39x slower, even though our lattice of abstract values has height 9, whereas Coutinho et al.’s has height two.

**Precision of the divergence analysis with affine constraints:** Figure 14 compares the precision of our analysis with the precision of the divergence analysis of Coutinho et al., as taken from Ocelot. A direct comparison between the number of divergent variables is not possible, because these two implementations use slightly different program representations: while Ocelot’s version of Coutinho et al.’s algorithm partitions live ranges at every basic block boundary, we only do it at those basic blocks that leave out loops. Ocelot’s analysis reports that 63.78% of the variables are divergent, while we report 58.81%. However, notice that Ocelot’s simple divergence analysis can only mark a variable as uniform or not. On the other hand, our analysis can find that a non-trivial proportion of these divergent variables are affine functions of some thread identifier. Figure 13 (Left) gives the distribution of the abstract states that we found with the divergence analysis with affine constraints of degree 2. In that figure, we let $a_i T_{di}^2 + a_i T_{di} + a_0 = (a_3, a_4, a_5)$. Even though we report that 58.81% of the variables are divergent, i.e., have $a_1 \neq 0$, 24.84% of them are affine functions of some thread identifier. The affine analysis of degree 2 adds negligible improvement on the analysis of degree 1. The latter misses 39 divergent variables that the former captures in almost 10,000 variables.

**Register allocation:** Figure 15 compares three different implementations of divergence aware register allocators. We use, as a baseline, the linear scan register allocator [33] that is publicly available in the Ocelot distribution. All the other allocators are implemented as re-writing patterns that change the spill code inserted by linear scan according to the rules in Figure 11. All the four allocators use the same policy to assign variables to registers and to compute spilling costs. The divergence aware allocators are: (DivRA) which moves to shared memory only the variables that the simple divergence analysis [15] marks as uniform. This allocator can only use the second rule in Figure 11; (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 $T_{di}$ with known constants. This allocator uses only the first and third rules in Figure 11: (AffRA), which uses all the four rules in Figure 11 plus the results of our analysis with polynomials of degree 2.

We report time for each kernel individually. 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. 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, whereas DivRA and AffRA use eight, because these two allocators must reserve one register to load the base addresses that

![Figure 12](image1.png)  ![Figure 13](image2.png)
Affine
5
0
1
line
3:
(c,c)↓
DivRA/LS
19
1
0
19
line
2:
(0,
AffRA/LS
16
5
line
4:
(c,
61
naive
Divergence Analysis with Affine Constraints
9
Figure 15. 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. SDK’s allocators. Every bar is normalized to the time given by Ocelot’s linear scan register allocator. The shorter the bar, the faster the kernel. Numbers count loads from local memory. In this experiments we are reserving the 16KB cache to local memory, i.e., the kernels have been compiled with the option -d1cm=cg; thus, loads from global memory are not cached. On the average, all the divergence aware register allocators improve on Ocelot’s original linear scan. The code produced by RemRA, which only improves spilling via rematerialization, is 7.31% faster than the original linear scan algorithm. This fact happens because (i) the local memory has access to a 16KB cache that is as fast as shared memory; (ii) loads and stores according to rule four of Figure 11 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.

Figure 13 shows how many times each rewriting pattern in Figure 11 has been used during register allocation by AffRA. We use ↓ and ↑ to denote loads-from and stores-to local memory; similarly, ↓ and ↑ represent loads-from and stores-to shared memory. The tuples (0, ⊥), (c₁, c₂) and (c₁, ⊥) refer to the second, third and fourth lines of Figure 11, respectively. We did not find occasion to rematerialize constants; thus, the rules in the first line of Figure 11 have not been used. An entry such as (c₁, ⊥) ↓ indicates that 877 loads from local memory have been replaced by a load from shared memory plus a multiply-add instruction, according to the fourth line, first column of Figure 11. As we see in Figure 13, AffRA has been able to replace almost half of all the spilling code with faster instruction sequences, closely following the proportion of abstract states found by the divergence analysis. Interestingly, many loop limits have the abstract state (0, c₁, c₂), c₁, c₂ ∈ Z. These variables are good spill candidates, as they have long live ranges, and
tend to be only used once in the program code. Both RemRA and AffIRA could take benefit from the affine analysis to rematerialize these variables whenever necessary. Figure 15 (iii) shows the proportion of spill code inserted by AffIRA in each benchmark.

7. 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 we believe that these techniques will 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 [9] branch splitting.

Reproducibility: our code plus tables with experimental data are publicly available at http://simdopt.wordpress.com/

References


[38] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugarman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and...


