GPU Divergence: Analysis and Register Allocation

Diogo Nunes Sampaio
Adviser: **Fernando Magno Quintão Pereira**

[1]Universidade Federal de Minas Gerais

sampaio@dcc.ufmg.br

Abstract. *The 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 a phenomenon called divergences. Divergences happen whenever two processing elements (PEs) see the same variable name holding different values. In this dissertation we have introduced divergence analysis, a static analysis that discovers data divergences. This analysis, currently deployed in Ocelot, 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 have demonstrated this last point by introducing the notion of a divergence aware register allocator. This allocator 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 allocator produces GPU code that is 29.70% faster than the code produced by the register allocator originally used in Ocelot. These results have received substantial acknowledgment from the research community. In particular, we got the best paper in the Brazilian Symposium on Programming Languages, and published our work in the ACM Transactions on Programming Languages and Systems (TOPLAS), the leading journal in the field.*

1. Introduction

Increasing programmability and low hardware cost are boosting the use of graphical processing units (GPU) 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.

This dissertation **solved** this problem by introducing the *Divergence Analysis*. This analysis identifies variable names holding the same value for all the threads in a warp. The analysis precision is augmented by the *Affine Divergence Analysis*, where the value of every variable is described as a linear equation to the special variable $\mathtt{T}_{id}$ (*thread identifier*), that always holds a unique value for each processing element.

Discovering divergent 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. There exist many attempts to compile such languages to ordinary CPUs [Diamos et al. 2010, Karrenberg and Hack 2011, Stratton et al. 2010] vector instruction sets that 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 analyses enables *divergence aware* code optimizations, such as Coutinho *et al.*'s [Coutinho et al. 2011] *branch fusion*, and Zhang *et al.*'s [Zhang et al. 2011] thread reallocation. We have augmented this family of techniques with a *divergence aware register allocators*. These allocators are a **second contribution** of this work. As we will show in Section 3, we use divergence information to decide the best location of variables that have been spilled during register allocation. The concrete result of our ideas is a register allocator that produces code 29.70% better than the original algorithm. To emphasize that we are not comparing against a straw-man, we have tested our ideas against the allocator of Ocelot, a GPU compiler. *Our results were good enough to convince the Ocelot maintainers to replace their original allocator with our new one.*

Products of this Dissertation: This dissertation gave origin to a large number of publications, and a substantial body of software nowadays publicly available in an industrial-quality compiler. We have published two journal papers [Coutinho et al. 2013, Sampaio et al. 2013], one paper in an international conference [Coutinho et al. 2011], and three papers in national symposia [Sampaio et al. 2010, Sampaio et al. 2012a, Sampaio et al. 2012b]. The dissertation author is the first author of four of these papers and second on the others. We would like to emphasize three achievements of this work:

- [Sampaio et al. 2012b] won the **Best Paper** of the Brazilian Symposium on Programming Languages (SBLP) of 2012.
- [Sampaio et al. 2013] was published in the ACM Transactions on Programming Languages and Systems (TOPLAS), the leading journal in the field of programming languages.
- Contributions to [Coutinho et al. 2011] that was published in PACT, a conference with an acceptance rate of under 17%.

In addition to the publications, we have contributed a long string of patches to Ocelot [Diamos et al. 2010], an open source compiler for GPUs, used in NVIDIA[1]. Therefore, instead of being restricted to the digital libraries of academic research, our work is currently available in an open source tool, is effectively used in the industry, and

[1]Ocelot is available at `http://code.google.com/p/gpuocelot/`

is part of a vibrant and enthusiastic community of developers. To illustrate the importance of this contribution, we quote Gregory Diamos, research engineer of NVIDIA[2]. Gregory's statement also pinpoints our contributions, in terms of software designed and released:

> "Revision 1943 includes a few major updates to the analysis framework in Ocelot. It adds in new SSA forms, a framework for analyzing uniform and affine values, and enhanced utilities for register allocation. Two new SSA forms (gated and minimal) improve the precision of divergence analysis. The uniform and affine variable analysis can identify values that are computed identically across all threads, or as simple functions of the thread id. Finally, the new register allocation code is able to spill uniform and affine registers to shared memory, potentially improving the performance of programs with high register pressure. The new framework includes helper code for managing live ranges, spilling registers, and maintaining stack space for spilled registers. I want to thank Diogo Sampaio, Sylvain Collange, Rafael Martins, and Fernando M. Q. Pereira for this contribution of over 10,000 lines of code."

In the rest of this document we provide a technical overview of our work. We omit describing the *Affine Divergence Analyses* as well as some references in this document. For a deeper understanding we recommend the original dissertation. In addition to it, our ideas are nowadays available as a set of class notes[3]. We also maintain two web pages where we explain our core ideas to the open software community [4] [5].

2. Background

Our compiler analyses and optimizations have been designed for graphics processing units (GPUs). These machines follow the Single Instruction, Multiple Data (SIMD) execution model. The execution of a 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 $\theta[v_1]$.

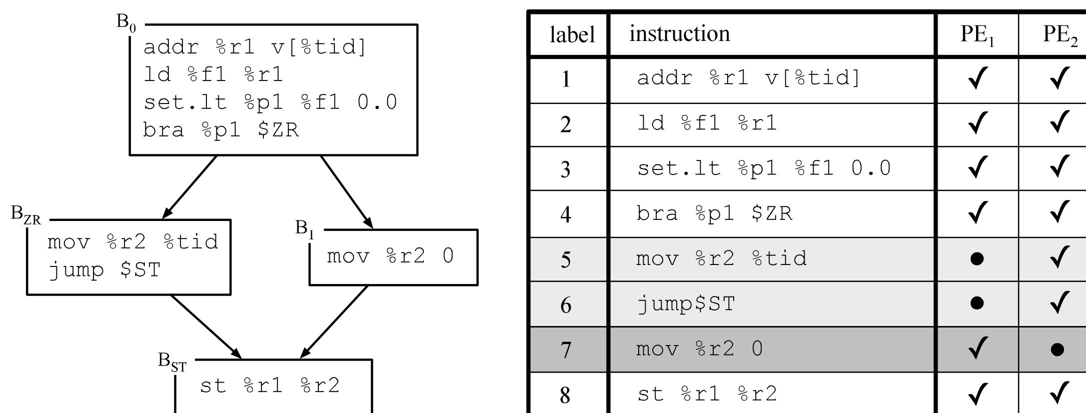
In the SIMD world, divergences happen due to conditional tests. A conditional test `bra` v, l' at label l causes all the threads to evaluate their $\theta[v]$. Those that find $\theta[v] \neq 0$ branch to l', whereas the others fall through the next instruction at $l + 1$. If two threads take different paths, i.e., v is divergent, then we say that the threads *diverge* at l. Figure 1 illustrates this phenomenon. Our example program of Figure 1 contains two processing elements, PE_1 and PE_2. When visiting the branch at the end of block B_0, predicate p is 0 for thread PE_1, and 1 for the other thread. In face of this divergence, PE_1 is pushed onto a stack of waiting threads, while PE_2 executes the instructions at block B_1. Once PE_2

[2]Message available at `https://groups.google.com/forum/#!searchin/gpuocelot/diogo$20sylvain$20thank/gpuocelot/HJzamEhq23w/Hz7rftjv4GYJ`

[3]The class notes about divergence analysis are available at `http://homepages.dcc.ufmg.br/ fernando/classes/dcc888/ementa/slides/DivergenceAnalysis.pdf`

[4]On-line description of divergence analysis available at `http://divmap.wordpress.com/`

[5]On-line description of register allocator available at `https://simdopt.wordpress.com/`



label	instruction	PE_1	PE_2
1	`addr %r1 v[%tid]`	✓	✓
2	`ld %f1 %r1`	✓	✓
3	`set.lt %p1 %f1 0.0`	✓	✓
4	`bra %p1 $ZR`	✓	✓
5	`mov %r2 %tid`	•	✓
6	`jump$ST`	•	✓
7	`mov %r2 0`	✓	•
8	`st %r1 %r2`	✓	✓

Figure 1. (Left) A GPU program in assembly format. (Right) The execution trace of the program, illustrating a divergence at the branch.

reaches a synchronization point, the execution of PE_1 resumes, and this thread processes the instructions at block B_{ZR}. The threads resume lock-step execution at block B_{ST}. The main problem of divergences is the loss of performance: in a SIMD machine, instructions are fetched for all the threads. Yet, threads in the waiting stack are not allowed to write back their results. Thus, we have an excess of "dead-code execution", which slows down the runtime of the parallel programs and accounts for higher dissipation of power.

3. The Divergence Analysis

We have designed and implemented a compiler analysis that detects which variables have always the same value for all the threads. These variables are called *uniform*. Non-uniform variables are called *divergent*. Our analysis works recursively; thus, we mark as *divergent* the following variables:

1. The thread identifier $\texttt{T}_{id}$.
2. Variables created by atomic instructions.
3. Variables that are data dependent on divergent variables.
4. Variables that are control dependent on divergent variables.

Items 1 and 2 above give us the base cases of our definitions. Items 3 and 4 give us the recursive cases. We say that a variable u is *data* dependent on a variable v if u appears in the right side of an instruction that uses u. Thus, detecting data divergences is a trivial operation guided by the syntax of the program. For instance, variable `%f1` is data dependent on variable `%r1` in Figure 1, due to the second instruction in block B_0. On the other hand, discovering the so called *control* dependencies is more complicated. We say that a variable u is control dependent on a variable v if v is used in a conditional test, and the outcome of this test defines the value of u. For instance, in Figure 1, variable `%r2` is control dependent on variable `%p1`, because its value depends on how we branch at the end of block B_0. If we take the "then" clause of the branch, then variable `%r2` will be assigned the value $\texttt{T}_{id}$, otherwise it will receive the constant zero.

We have solved the problem of identifying control dependencies by relying on an old program representation called the *Gated Static Single Assignment* (GSA) form [Ottenstein et al. 1990]. This program representation was initially conceived to help

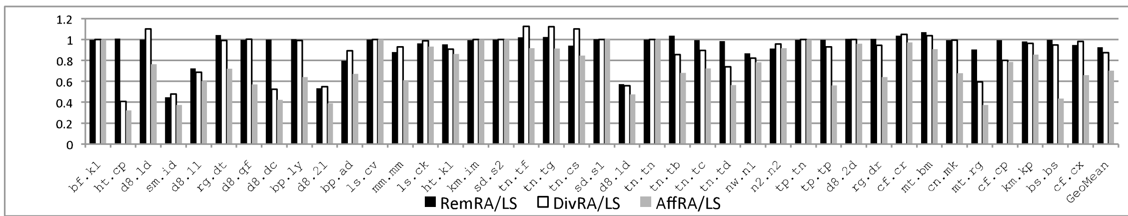


Figure 2. Relative speedup of different register allocators. Every bar is normalized to the time given by Ocelot's linear scan register allocator. The shorter the bar, the faster the kernel.

compilers to synthesize hardware to encode specific programs. The fact that we can use GSA form to identify divergences in SIMD programs is one of the contributions of this work. We use an almost linear time algorithm to convert a program into GSA form. Once a program is in the GSA format, we can reduce the problem of identifying control dependencies to the problem of computing data dependencies. This reduction is **another contribution** that we claim in this work. A complete description of this technique can be found in Section 3 of our TOPLAS paper [Sampaio et al. 2013].

4. Divergence Aware 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 GPU register allocators, such as the one used in the NVIDIA compiler, or in Ocelot [Diamos et al. 2010], 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 of a typical GPU. By mapping uniform variables to shared memory we have been able to improve the runtime of the code produced by a traditional register allocator by more than 29%.

Figure 2 quantifies these results. We have implemented different versions of register allocators - all of them able to map uniform variables to shared memory. Our three different allocators could deliver meaningful speedups on top of Ocelot's original allocator, which maps every spill, be it divergent or uniform, to local memory. Our best allocator, which in Figure 2 we have called AffRA, spills into uniform memory using the divergence analysis that we have described in the beginning of the present section. As mentioned before, this allocator speedups the code produced by Ocelot in over 29%. Nowadays, AffRA is the official allocator of the Ocelot compiler, and we know that some of its techniques are also being implemented in the LLVM [Lattner and Adve 2004] backend that generates codes for GPUs.

5. Conclusion

This dissertation has introduced the notion of *divergence analyses* for GPUs. Our analyses are currently in use in an industrial-strength compiler, and it companion register allocator has been enjoying active use in the open source community. Thus, we close this presentation by highlighting once more that our work, in addition to its publications, has found its way into end-users of the serious and fast-growing industry of graphics processing units. More importantly, this work has laid solid foundations to the compiler analysis of SIMD code. This theory, which relies on decades of advancements in the field of programming languages, is described in our dissertation.

Software and Reproducibilidade: All the - more than 10,000 lines of - code implemented in this work is available at `http://code.google.com/p/gpuocelot/`.

References

Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Jr., W. M. (2011). Divergence analysis and optimizations. In *PACT*, pages 320–329. IEEE.

Coutinho, B., Sampaio, D., Pereira, F. M. Q., and Jr., W. M. (2013). Profiling divergences in GPU applications. *Concurrency and Computation: Practice and Experience*, 25(6):775–789.

Diamos, G., Kerr, A., Yalamanchili, S., and Clark, N. (2010). Ocelot, a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *PACT*, pages 354–364. IEEE.

Karrenberg, R. and Hack, S. (2011). Whole-function vectorization. In *CGO*, pages 141–150. IEEE.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE.

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 *PLDI*, pages 257–271. ACM.

Sampaio, D., Coutinho, B., and Pereira, F. M. Q. (2010). Detecting divergent branches in simd architectures. In *LTPD*. SBC.

Sampaio, D., de Souza, R. M., Collange, S., and Pereira, F. M. Q. (2013). Divergence analysis. *Trans. Program. Lang. Syst.*, 35(4).

Sampaio, D., Martins, R., Collange, S., and Pereira, F. M. Q. (2012a). Divergence analysis with affine constraints. In *SBAC-PAD*, pages 137–146. IEEE.

Sampaio, D. N., Gedeon, E., Pereira, F. M. Q., and Collange, S. (2012b). Spill code placement for simd machines. In *SBLP*, pages 12–26.

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 *CGO*, pages 111–119. IEEE.

Zhang, E. Z., Jiang, Y., Guo, Z., Tian, K., and Shen, X. (2011). On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*, pages 369–380. ACM.