Identifying scalar behavior in CUDA kernels

Caroline Collange

▶ **To cite this version:**

Caroline Collange. Identifying scalar behavior in CUDA kernels. [Research Report] ENS Lyon. 2011. hal-00555134

HAL Id: hal-00555134

https://hal.science/hal-00555134v1

Submitted on 12 Jan 2011

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.

Identifying scalar behavior in CUDA kernels

Caroline Collange*
ENS Lyon, Université de Lyon, Arénaire,
LIP (UMR 5668 CNRS - ENS Lyon - INRIA - UCBL)

January 12, 2011

Abstract

We propose a compiler analysis pass for programs expressed in the Single Program, Multiple Data (SPMD) programming model. It identifies statically several kinds of regular patterns that can occur between adjacent threads, including common computations, memory accesses at consecutive locations or at the same location and uniform control flow. This knowledge can be exploited by SPMD compilers targeting SIMD architectures. We present a compiler pass developed within the Ocelot framework that performs this analysis on NVIDIA CUDA programs at the PTX intermediate language level. Results are compared with optima obtained by simulation of several sets of CUDA benchmarks.

1 Introduction

The recent development of General-Purpose computing on Graphics Processing Units (GPGPU) has caused a rebirth of Single Program, Multiple Data (SPMD) programming models. Graphics shader languages and GPGPU-oriented languages such as CUDA and OpenCL are all built on a SPMD foundation. They require programmers to express data-parallelism by running many threads (on the order of tens of thousands) all running the same code. These languages are now widely used to program GPUs.

In addition to GPUs, CPUs are an intended target platform for the OpenCL standard [15]. Several CUDA and OpenCL implementations have been released by compiler and hardware vendors [2, 20, 21].

Virtually all high-performance microprocessors and even some embedded microprocessors feature short-vector SIMD extensions. For instance, the Intel x86

*The basis of this work was performed while the author was with NVIDIA Developer Technology.

architecture has been subject to several generations of extensions: 64-bit SIMD with MMX and with AMD 3DNow! [19], 128-bit with SSE [10], and 256-bit and 512-bit extensions have been proposed (with AVX [9] and Larrabee new instructions [22]). As illustrated by the evolution of x86, “short-vector” SIMD extensions tend to evolve toward wider vector sizes. Thus, a compiler which emits scalar instructions only can only reach a small and shrinking fraction of peak performance.

The execution models and instruction sets that GPUs provide are designed and optimized for pure data-parallel calculations [13]. Current GPUs are based on the Single Instruction, Multiple Threads (SIMT) model. This model consists in packing several threads together and executing their instructions in a SIMD fashion, while still retaining their individual ability to follow arbitrary control paths. GPUs provide hardware-based mechanisms to handle instruction divergence at runtime. Likewise, they can handle memory divergence, effectively supporting gather and scatter operations in hardware. On the other hand, CPU instruction sets offer strong scalar processing capabilities.

Because of these differences, a naive translation of an SPMD application to SIMD instructions will yield suboptimal performance. To bridge this gap, a CUDA compiler targeting CPUs needs to identify at compile-time situations that GPUs detect at runtime. In particular, it has to address branch divergence and memory divergence. It also needs to identify scalar instructions and scalar registers to take advantage of the scalar units of CPUs.

We propose a compiler pass that we call *scalarization*. Its goal is to provide the necessary knowledge upon which compiler backends for SIMD instruction sets can rely.

We will first present a quick overview of the CUDA environment and related literature in section 2. We then expose some of the challenges involved in the development of an SPMD-to-SIMD compiler in section 3. We present the scalarization analysis itself in section 4, then evaluate its accuracy in section 5.

2 Background

2.1 The CUDA stack

The programming model followed by GPU environments is based on the SPMD paradigm. The programmer writes one single program, or *kernel*. Many instances of the kernel, or *threads*, are run in parallel. Threads can be told apart by their thread identifier (ID).

In CUDA, threads are grouped together in several Concurrent Thread Arrays (CTAs). Threads are allowed to communicate and synchronize with each other

inside each CTA. By contrast, the execution order of CTAs is undefined and interactions between CTAs are discouraged [18].

Programs expressed in this programming model are mapped to the GPU architecture.

Logical threads are distributed on a hierarchy of wide SIMD, hardware multithreaded and multi-core execution resources. CTAs are scheduled to the execution cores. Threads inside a CTA are grouped together into fixed-width *warps*, which share an instruction pointer, such that all threads in a warp run in lockstep [13]. To maintain the illusion that each thread executes independently, intra-warp thread divergence is handled transparently in hardware. This mechanism is called Single Instruction, Multiple Threads (SIMT) in NVIDIA's documentation.

Various academic projects have grown on top of the CUDA environment. Among them, Ocelot is a compilation framework for CUDA programs [7]. It operates on an intermediate representation based on the NVIDIA PTX intermediate language. It includes a runtime implementing the CUDA Runtime API and supports several back-ends, including interpretation of the intermediate representation, compilation to LLVM [12] and GPU execution.

The internal representation available within Ocelot includes control-flow and data-flow graphs and a Single Static Assignment (SSA) virtual register allocation. The framework also provides a register allocation pass at the PTX level, and allows the generation of traces and statistics from the built-in PTX interpreter.

2.2 Related work

Several solutions have been proposed to compile SPMD languages such as C for CUDA to multi-core CPUs.

Before GPGPU became widespread, techniques to compile stream-based languages to the x86 architecture have been proposed by Gummaraju and Rosenblum [8]. They generate scalar instructions to execute most operations, and SSE vector loads and stores instructions for memory accesses.

The MCUDA project by Stratton et al. introduced source-to-source transformations called loop fission and microthreading, which group together several lightweight GPU threads to form one heavyweight CPU thread [23, 24]. Ocelot's LLVM backend works at the PTX intermediate language level and uses user-space context switching between lightweight threads [7].

These projects enable CUDA applications to take advantage of the concurrent multiprocessing, hardware multithreading and out-of-order execution capabilities of CPUs. However, we believe that there are still many opportunities left open to take advantage of the SIMD extensions offered by CPUs. MCUDA generates scalar C code, then have it vectorized back by the Intel C Compiler *icc*. Its authors

note that some source-level optimizations in the CUDA source code may prevent vectorization by *icc*. Ocelot generates scalar LLVM code for each thread at the time of writing.

An OpenCL compiler that perform vectorization by grouping threads together was proposed by Intel [21]. However, it is still unclear how well this compiler can cope with non-trivial control flow, and whether it can coalesce together consecutive memory access from different threads.

In order to detect and remove redundant computations, a compiler pass called variance analysis was developed by Stratton et al. to identify computations that are invariant on some components of the thread ID [23].

Likewise, a hardware-based technique was proposed to detect scalar operations in CUDA programs at runtime [5]. It defines a vector as *uniform* when all of its components hold the same value. An *affine* vector has successive lanes holding linearly increasing values, each separated to the next one by a constant *stride*. We will reuse these definitions in this paper.

The same approach can be extended to identify uniform branches and unit-strided memory accesses. In particular, affine vectors enable the static detection of vector loads and stores. We consider in this paper a static scalarization analysis. It identify both uniform vectors and affine vectors in PTX code, and use this knowledge to classify instructions, registers, branches and memory accesses at compile-time.

3 Mapping SIMT to SIMD

We discuss in this section how each kind of SIMT instruction can be mapped to the SIMD model.

Vector arithmetic Basic arithmetic SIMT instructions can be readily translated by realizing that a warp on the GPU is the equivalent of a lightweight thread on the CPU, while a GPU thread is just the concept of work processed inside one SIMD lane. Each SIMT arithmetic instruction can then be turned into one or several SIMD instructions. This method was proposed for the compilation of graphics shaders to the Larrabee architecture [22], and is used in shader compilers such as the LLVMPipe project[1].

Branches Current GPUs offer hardware support to manage divergent threads by dynamically selecting between predication or branching with minimal overhead.

[1] `http://zrusin.blogspot.com/2010/03/software-renderer.html`

Short-vector SIMD instruction sets miss this feature, so divergence control has to be implemented in software.

We expect software-based SIMT emulation will suffer from an overhead significantly higher than its hardware counterpart in the general case. However, if we can ensure at compile-time that all threads in a warp take the same code path, we can translate such *uniform* SIMT branch instruction to a scalar branch instruction. The branch becomes an inexpensive operation thanks to the advanced branch prediction mechanisms offered by current CPUs.

An analysis of various CUDA kernels performed by Kerr, Diamos and Yalamanchili shows that typical ratios of divergent branches over all branches range from 9.5% to 32% [11]. Hence, significant optimization opportunities could be exploited by detecting uniform branches statically.

3.1 Memory accesses

In an SPMD program, each thread can load and store data at arbitrary locations in memory. There is no guarantee that the address requested by each thread in a warp will be consecutive. In the general case, SIMT load and store instructions are respectively mapped to SIMD gather and scatter instructions. However, unit-strided access patterns are common in vector computations [3].

Gather and scatter instructions are typically not supported at all on CPUs, so they have to be broken down into several scalar loads or stores. This comes at a price in performance. For instance, a 10-time slowdown was observed for a naive implementation of a CUDA gather on consecutive addresses compared to sequential reads [6].

GPUs solve this issue by dynamically detecting whether the independent small memory transactions can be *coalesced* into fewer larger transactions. This allows performance to gracefully degrade as memory locality decreases [18].

Thus, it is necessary to detect as many unit-strided loads and stores at compile time as possible to compensate for the lack of such hardware support on CPUs.

3.2 Scalar processing

CPUs offer execution resources that are not found on GPUs. In particular, they can execute scalar code efficiently thanks to advanced out-of-order execution mechanisms. While data-parallel computations are best run on SIMD units, scalar Arithmetic and Logical Units (ALUs) are available for bookkeeping work, including address calculations, loop control and other scalar computations. Offloading these scalar calculations to scalar units can free valuable vector computing resources. Thus, we want to identify scalar computations at compile-time.

Similarly, CPUs typically have as many scalar registers as vector registers. Scalar registers also consume less storage and bandwidth resources when they are spilled to memory. Register pressure was shown to be significant issue with data-parallel workloads running a large number of concurrent threads to hide memory latency [25]. On the CPU, live registers have to be saved during context switches, which consumes cache resources.

Experimental simulations suggest that up to 30 % of all instructions operate on uniform or affine vectors [5]. Such uniform and affine instructions can be translated to scalar instructions on CPUs, making them essentially free.

4 A scalarization stage

We implemented a scalarization stage within the Ocelot framework. By performing the analysis during JIT compilation, we benefit from knowledge only known at run-time, such as CTA dimensions. PTX is high-level enough to preserve information about control flow, which allows us to perform optimizations across Basic Blocks (BB).

4.1 Dataflow analysis

We associate a tag to T each virtual register of the SSA representation, where:

$$T \in \{\bot, C(v,a), U(a), A(s,a), G(a)\}.$$

The tag C is associated with constants, U with uniform vectors, A with affine vectors, G with generic non-affine vectors, and $\bot$ with vectors whose state is currently unknown.

We keep the following data along with the tags:

- $v \in \mathbb{Z}$ is the value of the constant,
- $a \in \{\bot, 0, 1, \ldots, a_{\max}\}$ is the *alignment* of the register, defined as the minimum number of trailing zeroes of the binary representation of each component, or $\bot$ for the null vector,
- $s \in \mathbb{Z} \cup \{\top\}$ is the stride of the affine vector, or $\top$ when unknown.

We define a partial order on this structure, such that:

$$\begin{array}{rclrcl} \bot & \leq & C(v) & C(v) & \leq & U(a_{\max}) \\ U(a) & \leq & U(a-1) & U(0) & \leq & A(s,a) \\ A(s,a) & \leq & A(s,a-1) & A(s,a) & \leq & A(\top,a) \\ A(\top,a) & \leq & G(a) & & & \end{array}$$

Table 1: Examples of tag propagation rules across integer operations, where $s + \top = \top$, $a \times \bot = a$, $min(a, \bot) = a \ldots$

$$\frac{x : A(s,a) \quad y : A(s',a')}{z = x + y \Rightarrow z : A(s + s', min(a,a'))} \qquad \frac{x : A(s,a) \quad y : U(a')}{z = x \times y \Rightarrow z : A(\top, a \cdot a')}$$

$$\frac{x : A(s,a) \quad y : A(s,a')}{z = x - y \Rightarrow z : U(min(a,a'))} \qquad \frac{x :: A(s,a) \quad y : C(v,a')}{z = x \times y \Rightarrow z : A(s \cdot v, a \cdot a')}$$

We propagate these metadata through the control flow graph using a forward dataflow analysis [1]. Tags can be shown to form a lattice of finite height, allowing the dataflow analysis to converge.

As a side effect, the analysis performs independent constant propagation on both the base and stride of affine vectors. As the stride of a memory address is typically a constant, most computations on affine vectors become scalar computations, saving the need to compute the stride at runtime.

4.2 Intra-block propagation

Inside each basic block (BB) of the control flow graph, we propagate the metadata across each instruction. For basic integer instructions such as addition, multiplications and bit shifts, output tags are computed as a function of input tags using intuitive rules. A few examples of rules are listed on Table 1.

Likewise, floating-point operations just propagate the uniform and constant properties, but do not carry the other metadata such as aligment.

4.3 The influence of control flow

When a divergent branch is encountered during execution, some threads of the warp are temporarily disabled. In the SIMD model, this translates to partial writes to vector registers. Only the vector components corresponding to enabled threads will be written, while the other components will retain their former value.

This creates conceptual input dependencies from both the mask and the previous result, as illustrated in the conceptual SIMT execution pipeline figure 1.

When all vector input operands and the contents of the destination register are uniform, but the warp is in a divergent state, the vector result written to the register file will not generally be uniform. At compile-time, this demands an accurate analysis of control-flow divergence. The accuracy of the uniform branch detection is itself conditioned by the accuracy of the uniform register detection. Thus,

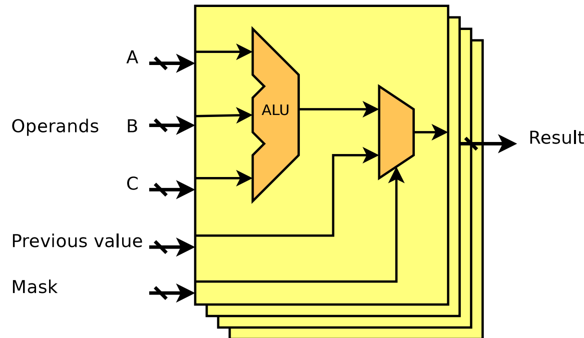

Figure 1: Functional view of an SIMT execution pipeline. As opposed to a scalar pipeline, the output value may depend on both the previous value and the current thread activity mask. Thick crossed lines are vectors, and thin lines are scalars.

Listing 1: Example of false dependency on mask.

```
1  __global__ void kernel(int * g) {
2      int i = 0, j;
3      do
4      {
5          j = i;
6      }
7      while(i++ < threadIdx.x);
8      g[threadIdx.x] = j;
9  }
```

the issues of detecting scalar computations and uniform control flow are tightly intermixed.

Some of these constraints are actually false dependencies, and observing them can lead to an overly pessimistic estimate. As an example, let us consider the synthetic CUDA code snippet in listing 1.

In this code, *threadIdx.x* represents the thread identifier. We assume that no optimization such as copy-propagation is performed. During the first loop iteration, the loop counter i is uniform across all threads, and get incremented in lockstep by every thread. However, the loop exit condition is not uniform across threads. Some threads will keep executing the loop and incrementing the counter, while other threads will be disabled until all threads exit the loop. Inactive threads do not commit data to the register file. The vector register storing i for a warp will not be considered uniform.

A closer examination reveals that the variable i is dead upon the end of the

loop. Also, a thread that was disabled during the execution of the loop will not be enabled again before the loop finishes for all the threads of the warp. The actual contents of the inactive lanes of register *i* do not matter, as they will not be read back by any subsequent instruction. In this case, *i* can be safely replaced by a scalar register.

Conversely, the variable *j* is alive after the end of the loop, and must still be considered as a generic vector.

When the enclosing BB can be executed in a divergent state, we also need to check whether the destination register is used (or merged) inside another block from a strictly enclosing nesting level of control flow structure. Such blocks are post-dominators of the considered BB. We can then walk the post-dominator tree that Ocelot provides, and look for uses of the considered register in sources of ϕ instructions. If and only if the register is used in a post-dominator, then we need to consider a control dependency exists.

Using barriers as hints of uniformity Even with the dead register optimization on minimal SSA form described above, the analysis is still often overly pessimistic because some uniform branches are not identified. One way to improve it is to take advantage of hints provided by higher-level structure.

CUDA supports an explicit CTA-wide synchronization barrier instruction (`__syncthreads`), with the restriction that it cannot be used in a divergent state [18]. Breaking this rule results in an undefined behavior. Therefore, it is safe to assume that any BB containing a barrier will always be executed in an uniform (synchronized) state.

4.4 Assumptions

Several assumptions condition the accuracy of scalarization.

Thread ID The thread identifier exposed by CUDA is a three-dimensional vector. This 3D thread ID is mapped to a linear physical thread ID. With D_x, D_y, D_z being the dimensions of the thread identifier space in a CTA, and (x, y, z) the vector thread ID, the linear ID is computed as $(x + yD_x + zD_xD_y)$ [17]. The proposed scalarization analysis requires D_x to be multiple of the warp size. However, CUDA allows other values of D_x. Though odd block dimensions are strongly discouraged [16], using $D_x = 16$ is a common practice. Block dimensions are known only at the time of kernel launch, rather than at compile-time.

Fortunately, the Ocelot environment provides a runtime and Just-In-Time (JIT) capabilities. This allows us to either disable scalarization when we encounter an

incompatible dimension, or to select the warp size as a function of the CTA dimensions. The latter choice is possible because SIMD extensions on CPUs have shorter widths than GPU warps, so adjusting the warp size equates to selecting the number of times each instruction will be replicated.

Pointer parameters A typical CUDA programming practice is to allocate device memory from host code through the CUDA runtime, then pass the allocated address as a parameter to the kernel. The CUDA runtime will always align memory on 256-byte boundaries [17]. However, the programmer can choose to perform pointer arithmetic in host code and break the alignment before passing the pointer to the kernel. Conservative assumptions will prevent any kind of static analysis on pointer alignment.

This problem can be also solved by performing checks for parameter alignment in the Ocelot runtime during kernel launch. If the assumptions made at compile-time do not hold, the kernel can be recompiled by the JIT.

Overflows Integer variables in C can silently overflow and wrap around. Together with type casts, this can break the affine property of a vector. This problem was recognized in the dynamic case and speculation and instruction replay has been proposed as a workaround [5]. This solution requires specific hardware support.

For the proof-of-concept analysis we present in this paper, we assume that this corner case does not happen. Ensuring this in the general case will require performing an additional interval analysis on the content of variables, and/or extending the PTX model with memory allocation information.

5 Accuracy

5.1 Test setup

We consider the benchmarks listed in table 2. SGEMM is a dense matrix multiply kernel developed by Volkov for the G80 architecture [26]. It is run on 256×256 matrices. 3DFD is a 3D finite differences stencil kernel [14]. The CTA size considered is 64×8, on $256 \times 256 \times 100$ volume. These two benchmarks are representative of small, highly-tuned kernels.

We also used Rodinia [4], UIUC Parboil[2] and examples from the CUDA SDK [3]. These three sets are part of the test suite of Ocelot. We only considered the benchmarks that we were able to run successfully under Ocelot's emulation mode.

[2] `http://impact.crhc.illinois.edu/parboil.php`
[3] `http://developer.nvidia.com/object/gpucomputing.html`

Table 2: Benchmark sets considered

Set	Applications	
SGEMM	SGEMM	
3DFD	3DFD	
Rodinia	hotspot	srad
SDK	AlignedTypes	BicubicTexture
	Bitonic	BlackScholes
	BoxFilter	Clock
	CppIntegration	DwtHaar1D
	FastWalshTransform	Histogram256
	Histogram64	ImageDenoising
	Mandelbrot	MatrixMul
	MersenneTwister	MonteCarlo
	MonteCarloMultiGPU	PostProcessGL
	RecursiveGaussian	Reduction
	ScalarProd	Scan
	ScanLargeArray	SimpleAtomicIntrinsics
	SimpleTemplates	SimpleTexture
	SimpleTexture3D	SimpleVoteIntrinsics
	SimpleZeroCopy	SobelFilter
	SobolQRNG	Template
	ThreadFenceReduction	Transpose
	TransposeNew	lt-SimpleGL
Parboil	rpes	tpacf
	sad	pns

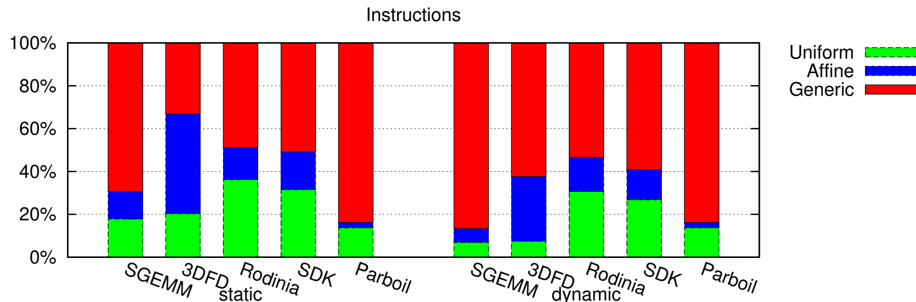

Figure 2: Classification of static and dynamic instructions as predicted by the static analysis.

We perform a scalarization analysis of every application kernel, and run the applications through the Ocelot emulator. Each register is classified according to its observed state (uniform, affine or generic) across the whole program execution. Equivalently, this can be through of running the scalarization analysis on traces of execution. The result provides an upper bound on how many scalar patterns can be identified at compile-time, assuming no modifications are made to the control-flow graph. We selected a warp size of 16 to model a wide SIMD instruction set.

The results quantify the accuracy of scalarization. As the analysis and simulation are both performed at the PTX level before most compiler optimizations take place, the results should not be considered as indicative of actual performance.

5.2 Instructions

Figure 2 presents a breakdown of the outputs of static instructions and dynamic instructions, as estimated by the scalarization analysis. This can be used to evaluate the ratio of scalar instructions to vector instructions.

The static instruction statistics show more scalar operations than their dynamic counterparts. Indeed, scalar calculations are often part of initialization code that is placed outside the inner loop by the programmer or the compiler. This is especially significant in highly-tuned code like SGEMM and 3DFD, where less than 8% of dynamic instructions write uniform data. The duplication of pointers and other scalar data was recognized by the authors of SGEMM. It lead them to favor smaller CTA dimensions and perform more work per thread to amortize the overhead [26].

The other benchmarks exhibit much less variation between the static and dynamic results, indicating that a significant number of scalar instructions are part of the inner loop.

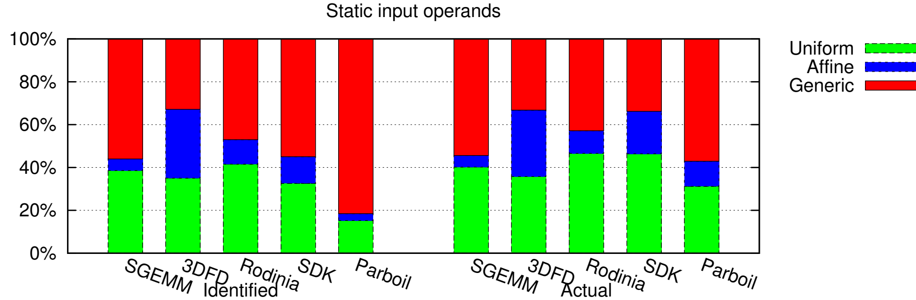

Figure 3: Classification of the input operands of static instructions.

5.3 Operands

We present the breakdown of input operands in figure 3. Despite having few address calculations inside its inner loop, SGEMM has a high number of uniform input operands. Indeed, elements from the B matrix block are broadcast from the same location in shared memory to all threads in the inner loop of the algorithm [26]. This effect remain when we consider dynamic instruction counts.

5.4 Load/stores

We classify loads and stores among the following categories:

- Uniform, when every thread in a warp requests the same address.
- Aligned unit-strided, when threads in a warp access contiguous data which starts at an address multiple of the vector width. This matches the coalescing requirements of the G80 architecture [18].
- (Unaligned) unit-strided, when data is contiguous but not generally aligned,
- Non-unit strided, when addresses from neighbor threads are separated by a constant stride, but data is not contiguous.
- Gather or scatter, the general case that does not belong to any other category.

Figures 4 and 5 respectively depict the classification of loads and stores to global memory.

We witness some writes (scatters) at uniform addresses. While they look like race conditions at first sight, these instructions are actually predicated such that only the first thread of the warp is active. They represent explicit scalar store instructions, as could happen for instance at the last step of a reduction.

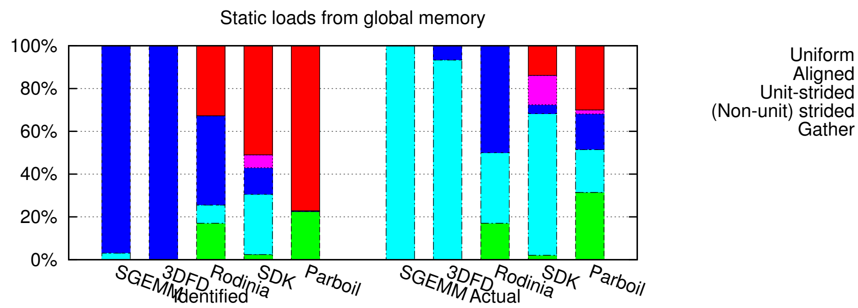

Figure 4: Global loads.

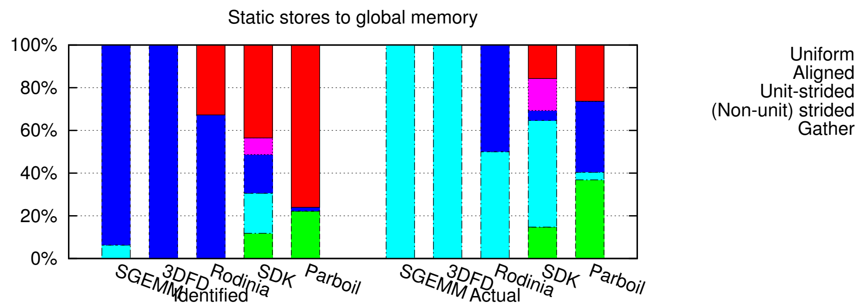

Figure 5: Global stores.

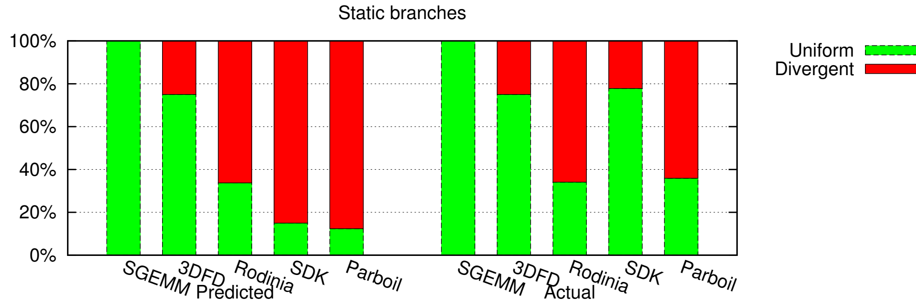

Figure 6: Classification of branches.

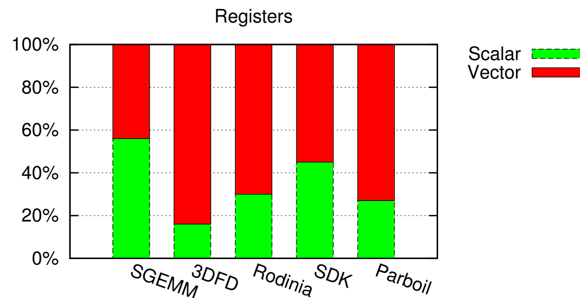

Figure 7: Types of PTX registers after register allocation.

5.5 Uniform branches

Ratios of uniform branches over all conditional branches are shown on figure 6.

Many CUDA applications make comparisons between two affine vectors of identical stride. In such cases, the result of the comparison is an uniform vector. When we added support for this specific rule, the ratio of affine instructions identified reached 41 %, versus 16 % initially. This experiment shows that the affine vector detection has applications beyond identifying unit-strided memory transactions.

5.6 Registers

To evaluate the register pressure reduction to be gained from scalarization, we used the register allocator built in Ocelot to find the lowest number of registers such that there is no spilling to memory, first by including all registers, and then by ignoring scalar registers (fig. 7). The number of scalar registers required if allocations of

scalar and vector registers were performed separately may be higher than results show, due to the inability to coalesce scalar and vector registers together. The estimation presented here is still meaningful when considering that scalar registers are not a limiting resource, as with short-vector SIMD extensions.

6 Conclusion

We presented a compiler analysis which is able to identify redundant or regular operations in SPMD programs, such as replicated scalar instructions, scalar variables, uniform branches, scalar loads and stores, and vector loads and stores.

This analysis can assist code generation for short-vector SIMD architectures such as x86 with SSE or AVX by providing the supporting information. Together with code transformation such as loop fission and insertion of context switches, it opens the way for efficient CUDA-to-x86 compilers.

Remaining challenges include finding ways to take advantage of dynamic information using just-in-time compilation, and improving the accuracy of uniform branch detection by a finer analysis of the control-flow structure.

References

[1] Alfred V. Aho, Monica S. Lam, , Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2006.

[2] AMD. ATI stream technology. `http://www.amd.com/stream`.

[3] Krste Asanović. *Vector microprocessors*. PhD thesis, University of California, Berkeley, 1998.

[4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. *IEEE Workload Characterization Symposium*, 0:44–54, 2009.

[5] Caroline Collange, David Defour, and Yao Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. In *Europar 3rd Workshop on Highly Parallel Processing on a Chip*, 2009.

[6] Gregory Diamos. The design and implementation ocelot's dynamic binary translator from PTX to multi-core x86. Technical report, Georgia Institute of Technology, 2009.

[7] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems. In *Nineteenth International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[8] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. *IEEE/ACM International Symposium on Microarchitecture*, pages 343–354, 2005.

[9] Intel. *Intel Advanced Vector Extensions Programming Reference*, 2009.

[10] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manuals Volume 1: Basic Architecture*, 2010.

[11] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. A characterization and analysis of GPGPU kernels. Technical Report GIT-CERCS-09-06, Georgia Institute of Technology, 2009.

[12] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, 2004.

[13] John Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[14] Paulius Micikevicius. 3d finite difference computation on GPUs using CUDA. In *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84, 2009.

[15] Aaftab Munshi. The OpenCL specification. *Khronos OpenCL Working Group*, 2009.

[16] NVIDIA. *NVIDIA CUDA Best Practices Guide*, 2010.

[17] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 3.0*, 2010.

[18] NVIDIA. *NVIDIA CUDA Programming Guide, Version 3.2*, 2010.

[19] Stuart Oberman, Greg Favor, and Fred Weber. AMD 3DNow! technology: architecture and implementations. *IEEE Micro*, 19(2):37–48, 1999.

[20] PGI. *PGI CUDA-x86: CUDA Programming for Multi-core CPUs*, November 2010.

[21] Ofer Rosenberg. Optimizing opencl on cpus. OpenCL BOF in SIGGRAPH 2010, 2010.

[22] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[23] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 111–119, 2010.

[24] John A. Stratton, Sam S. Stone, and Wen-Mei W. Hwu. MCUDA: An efficient implementation of cuda kernels for multi-core CPUs. In *Languages and Compilers for Parallel Computing: 21th International Workshop*, pages 16–30, 2008.

[25] Vasily Volkov. Programming inverse memory hierarchy: case of stencils on GPUs. *GPU Workshop for Scientific Computing, International Conference on Parallel Computational Fluid Dynamics (ParCFD)*, May 2010.

[26] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.