

Dynamic detection of uniform and affine vectors in GPGPU computations

Sylvain Collange¹, David Defour¹ and Yao Zhang²

¹ ELIAUS, Université de Perpignan, 66860 Perpignan, France
{sylvain.collange,david.defour}@univ-perp.fr

² ECE Department, University of California Davis
yaozhang@ucdavis.edu

Abstract. We present a hardware mechanism which dynamically detects uniform and affine vectors used in Graphics Processing Units, to minimize pressure on the register file and reduce power consumption with minimal architectural modifications. A preliminary experimental analysis conducted with a simulator shows that this optimization can benefit up to 34 % of register file reads and 22 % of the computations of GPGPU applications.

1 Introduction

GPU's are now powerful and programmable processors that have been used to accelerate general-purpose tasks other than graphics applications. These processors rely on a Single Program Multiple Data (SPMD) programming model. This model is implemented by many vector units working in a Single-Instruction Multiple-Data (SIMD) fashion, and vector register files. Register usage is a critical issue as the number of instance of the same program that can be executed simultaneously depend on the number of hardware registers and the register usage per instance. Making this bad situation worse, vectorizing scalar operations in an application makes inefficient use of registers and functional units. To efficiently handle scalar data, CRAY-like vector machines incorporate scalar functional units as well as scalar registers. Modern GPUs lack such scalar support, leaving it to vector units. These vector units execute the same instruction on the same data leading to as many unnecessary operations as the length of the vector when uniform data are encountered. These unnecessary operations involve data transfers and activity in functional units that consume power, which is a critical concern in architectural and microarchitectural designs of GPUs.

We observed through our experiments that standard GPGPU applications use a significant number of vectors to store uniform data. A closer look at the manipulated values shows that this number is even higher when we consider affine values (e.g. 1, 3, 5, 7, ...) stored in a given vector. Motivated by this observation, we propose, simulate and evaluate a technique that tags a vector register file according to the type of registers: uniform, affine or generic vector.

The rest of the paper begins with a brief description of the NVIDIA architecture upon which our model is based. Section 3 presents our performance

evaluation methodology, based on a functional simulator named Barra. We use it to both evidence the presence of redundancy in calculations in Section 4 and evaluate the proposed technique described in Section 5. We discuss technical issues in Section 6. Section 7 presents quantitative results and figures, and Section 8 concludes the paper.

2 Architecture Model

The base architecture we consider in our simulations consists of a vector processor, a set of vector register files, a set of vector units, and an instruction set architecture that mimics the behavior of the NVIDIA GPUs used in the Compute Unified Device Architecture (CUDA) environment [1]. This environment relies on a stack composed of an architecture, a language, a compiler, a driver and various tools and libraries.

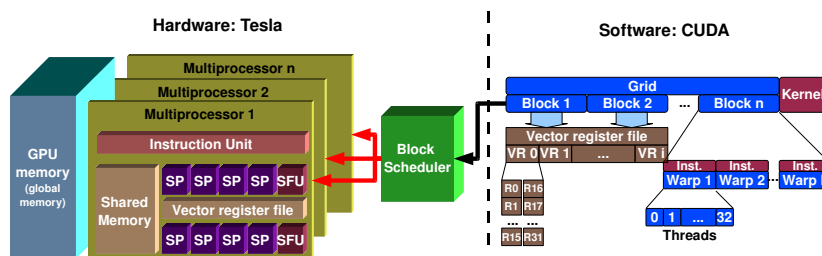


Fig. 1. Processing flow of a CUDA program.

A CUDA program runs on an architecture composed of a host processor CPU, a host memory and a graphics card with an NVIDIA GPU with CUDA supports. Currently all CUDA-enabled GPUs are based on the Tesla architecture, which is made of an array of multiprocessors. Tesla GPUs execute thousands of threads in parallel thanks to the combined use of multiple multiprocessors, SIMD processing and hardware multithreading [2]. Figure 1 describes the hardware organization of such a processor. Each multiprocessor contains the logic to fetch, decode and execute instructions, except memory instructions which are handled by a dedicated memory pipeline shared by two or three multiprocessors. Each multiprocessor embeds two types of vector units, that is, 8 SP units that perform one Multiply-and-Add instruction every twice cycle, and 2 special-function units that perform one instruction every 8 cycles. Such throughput is possible thanks to functional units working at twice the frequency of the vector register file. There are 256 or 512 vector registers, each register being a 32-wide vector of 32-bit values. In addition to the register file, each multiprocessor contains a scratchpad memory (or shared memory, using NVIDIA's terminology) and separate caches for constant data and instructions.

The hardware organization is tightly coupled with the parallel programming model of CUDA. The programming language used in CUDA is based on C with extensions to indicate if a function is executed on the CPU or the GPU. Functions executed on the GPU are called *kernels*. CUDA lets the programmer define if a variable resides in the GPU address space and specify the kernel execution across different granularities of parallelism: *grids*, *blocks* and *threads*. As the underlying hardware is a SIMD processor, threads are grouped together in so-called *warps* which operate on 32-wide vector registers. Each instruction is executed on a warp by a multiprocessor. Warps execute instructions at their own pace, and multiple warps can run concurrently on a multiprocessor to hide latencies of memory and arithmetic instructions. This technique helps hide the latency of streaming transfers, and improve the effective memory bandwidth. The register file of a multiprocessor is logically split between the warps it executes. As a GPU includes several multiprocessors, warps are grouped into blocks. Blocks are scheduled on the available multiprocessors. A multiprocessor can process several blocks simultaneously if enough hardware resources (registers and shared memory) are available.

The compilation flow of a normal CUDA program is a three-step process directed by the CUDA compiler *nvcc* [3]. First, according to specific CUDA directives from the CUDA Runtime API, the program is split into a host program and a device program. The host program is then compiled using a host C or C++ compiler and the device program is compiled through a specific back-end for the GPU. The resulting device code is binary instruction codes (in *cubin* format) to be executed on a specific GPU. The host program and the device program are linked together using the CUDA libraries, which includes the necessary functions to load a cubin either from inside the executable or from a stand-alone file and send it to the GPU for execution.

3 Barra, a Functional Simulator of NVIDIA GPUs

Several options exist to model the dynamic behavior of CUDA programs. CUDA offers a built-in emulation mode that run POSIX threads on the CPU on behalf of GPU threads, thanks to a specific compiler back-end. However, this mode differs in many ways with the execution on a GPU : the behavior of floating-point and integer computations, the scheduling policies and memory organization are different.

GPU simulators running CUDA's intermediate language PTX such as GPUSim [4] or Ocelot [5] can offer a greater accuracy, but still run an unoptimized intermediate code instead of the instructions actually executed by a GPU.

Recent versions of CUDA include a debugger that allows watching the values of GPU registers between each line of source code. This comes at a significant cost in performance, as registers are spilled to an uncached memory after each instruction. Though this mode offers perfect functional accuracy, it cannot be modified for instrumentation or feature evaluation purposes.

Barra simulates the actual instruction set of the NVIDIA Tesla architecture at the functional level. The behavior of all instructions is reproduced with bit-accurate accuracy, with the exception of transcendentals (exp, log, sin, cos, rep, rsq). To our knowledge, Barra is the only publicly-available tool that both executes the same instructions as Tesla GPUs and allows viewing the exact contents of registers during the execution.

The overall structure of this simulator consists of two major parts:

1. the Barra driver,
2. an instruction level simulator.

The simulator takes the binary code compiled normally with NVIDIA's nvcc compiler, extracts necessary information from the execution environment, simulates the execution of each instruction of the input code, and finally produces statistics for the simulated kernels.

3.1 Barra Driver

The Barra framework is designed so that the simulator can replace the GPU with minimal modifications in the development or execution process of a CUDA program. A driver uses a shared library that has the same name of NVIDIA's proprietary one *libcuda.so* so that function calls destined to the GPU can be dynamically captured and rerouted to the simulator. The user can choose to execute a program either on the GPU or on the simulator by setting an environment variable.

The proposed driver includes major CUDA Driver API functions that are necessary to load, decode and execute a CUDA program.

3.2 Instruction-Level Simulator

The instruction-set simulator executes each assembly instruction according to the model described in Figure 2. First, a scheduler selects the warp ready for execution according to a round-robin policy and reads its current program counter (PC). Then the instruction is fetched and decoded. Then operands are read from the register file or from on-chip memories (scratchpad) or caches (constants). The instruction is executed and its results are written back to the register file.

3.3 Vector register file

General Purpose Registers (GPRs) are dynamically split between threads during kernel launch, allowing a trade-off between the number of registers per threads and the latency hiding capability. Barra maintains a separate state for each active warp in the multiprocessor. These states include a program counter, address and predicate registers, mask and address stacks, a window to the assigned register set, and a window to the shared memory.

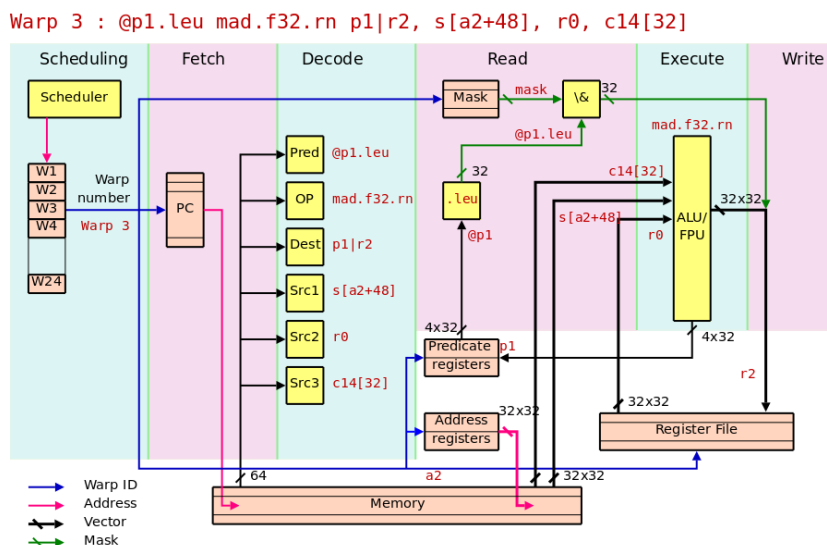


Fig. 2. Overview of the functional execution pipeline during the execution of a MAD instruction.

Multiprocessors of Tesla-based GPUs have a multi-bank register file partitioned between warps using sophisticated schemes [6]. This allows a space-efficient packing while minimizing bank conflicts. However, the specific register allocation policy bears no impact on the functional behavior, apart from deciding how many warps can have their registers fit in the register file.

4 Uniform and affine data in SPMD code

NVIDIA’s so-called “scalar” architecture is actually a pure vector (SIMD) architecture. At the hardware level, all instructions, including memory and control-flow operations, operate on vectors, and all architecturally-visible registers are vectors. Though this provides a clean programming model by abstracting away the vector length and allows scalable implementations, it can become a source of inefficiency when performing inherently scalar operations. This issue is akin to value locality, where a correlation in time is observed in many values appearing in computations [7]. However in vector processors, correlations appear mostly inside vectors rather than between different time steps.

A uniform vector V is defined as having every component contain the same value $V_i = x$. Two main cases lead to *uniform* pattern in vector register file. First when a constant value is loaded and second when control conditions are manipulated as the condition is stored in GPRs. This control condition are used to extend the class of program that can be executed on a GPU by allowing sub-vector control-flow. However, best performance is achieved when the control condition is uniform across a warp [1]. This means that in optimized algorithms, all lanes of a register used as a condition will hold the same value.

Similarly, to maximize memory bandwidth, memory accesses should follow specific patterns, such as the coalescing rules or conflict-free shared-memory access rules. In the Tesla architecture, memory addresses are computed using the regular SIMD ALUs. Programs following NVIDIA guidelines to access memory will operate mostly on consecutive addresses, which are common in vector programming. This corresponds to a *affine* pattern, when threads access memory in sequence. In this case, the vector register V that stores the address is such that each component $V_i = x + iy$. One can notice that the uniform pattern is a specific case of the affine pattern when $y = 0$.

To quantify how often both of these patterns occur, we use Barra to dynamically check for each input and output operand in registers if they are uniform or affine vectors. We perform this analysis on two kinds of applications.

First, we used the examples from the CUDA SDK. Even though these examples are not initially meant to be used as benchmarks, they are currently the most standardized test suite of CUDA applications. As code examples, they reflect the best practices in CUDA programming.

The second benchmark is a bioinformatics application. RNAFold_GPU is a CUDA program which performs RNA folding. Based on dynamic programming, it achieves a 17-time speedup compared to a multicore implementation [8].

The proportion of uniform and affine inputs/outputs from and to registers is depicted in figure 3. Uniform or affine input data represent the percentage of uniform (affine) vectors among the data transferred between the register file and functional units.

Similarly, uniform or affine output data is the proportion of uniform (affine) data written back to the register file. It can be observed that whenever the output is uniform (affine), the operation itself is executed on uniform (affine) data only.

We observe that respectively an average of 27 % (44 %) of data read from the vector register file are uniform (affine) and 15 % (28 %) of data written back are uniform (affine). This proportion of uniform or affine inputs/outputs is significant enough to justify specific optimizations.

5 Proposed technique

In this section we describe a technique which can detect if registers contain uniform or affine data as defined in Section 4. The first objective is to minimize memory and bus activity between the register file and the functional units for the proportion of input data captured by the proposed technique. The second objective links to the first one and goes further, by detecting uniform or affine data that are provided at the input of functional units and that remains uniform or affine at the output. In that case, the result can be computed by dedicated scalar hardware like in Cray processor or by relying on the existing vector hardware. As we target power reduction, the scalar solution will provide automatic reduction. However, this solution will introduce irregularities in the register file at a cost of a complex design. The second solution can benefit of techniques that were not

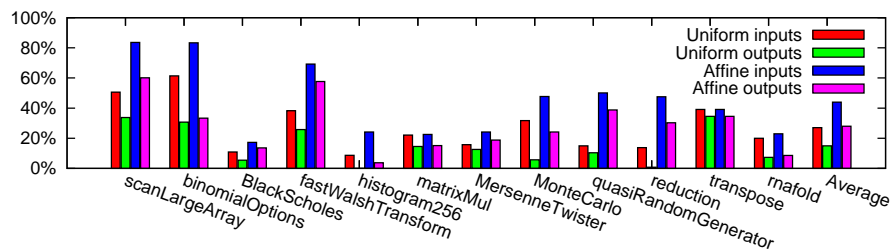


Fig. 3. Proportion of uniform and affine operands in registers. Averages are 27 % uniform inputs, 15 % uniform outputs, 44 % affine inputs and 28 % affine outputs.

available at the time Cray machines were designed like clock-gating. The second solution can reuse the same vector hardware, with one or two scalar units used to compute the result, the other units being shut down using stage-based clock gating as in the case of the IBM Cell FPU [9].

Instructions executed by GPUs show that most of uniform and scalar data come from a broadcast of some data or a copy of the register that contains the thread identifier. For these cases, uniform and scalar detection can be done statically for once at compile time or dynamically in hardware.

A static detection involves architectural as well as microarchitectural modifications. First, each instruction and register detected as uniform or scalar by the compiler has to be tagged in the instruction word. Then at runtime during the decode stage, the hardware can automatically schedule instructions according to the tag data. A dynamic detection keep the instruction set unchanged. However the information and the detection related to uniform and scalar data is transferred from the compiler to the hardware. This solutions requires for example a tagged vector register file.

We tested the dynamic solution based on a tagged vector register file where each tag contains the type of data stored in the associated register (uniform, affine or generic vector) using the Barra simulator. At kernel launch time, the tag of the register that contains the thread identifier is set to the affine state. Instructions that broadcast values from a location in constant or shared memory set the tag of the result to the uniform state. Tags are then propagated across arithmetic instructions according to a simple set of rules, as shown in table 1. We arbitrarily restrict the allowed strides to powers of two to allow efficient hardware implementations, and conservatively make multiplications between affine and uniform data return vectors. Additionally, the information stored in this tag may be used by memory access units as it gives information about memory access patterns.

Table 1. Examples of rules of uniform and affine tag propagation. For each operation, the first row and first column indicate the tag of the first and second operand, respectively (Uniform, Affine or Vector). The central part contains the computed tag of the result.

| | | | | | |
|---|-------|---|-------|----|-------|
| + | U A V | × | U A V | << | U A V |
| U | U A V | U | U V V | U | U A V |
| A | A V V | A | V V V | A | V V V |
| V | V V V | V | V V V | V | V V V |

Cost. A tag array contains two bits per vector register. Each multiprocessor of a NVIDIA GT200 GPU features 512 1024-bit registers, for a total register-file size of 512 kb (not accounting for error-correction codes). In a basic implementation, the associated tags would require 1 kb of register file, making it comparatively almost negligible.

In terms of latency, reading the tags adds one level of indirection before reading registers. In NVIDIA GPUs, registers are read in sequence for a given instruction to minimize bank conflicts [6]. Therefore, operand reads can be pipelined with tag reads. Additionally, GPUs can tolerate large instruction latencies by fast context switching between threads. The tag of the output can then be computed using a few boolean operations from the tags of the input, so the required hardware modifications are minimal. Support for broadcasting a word across all SIMD units is already available to handle operands in Constant and Shared memory.

Benefits. When an input or output operand is known to be uniform, only one lane needs to be accessed. Likewise, affine vectors v such that $v_i = x + iy$ can be encoded using the base x and the stride y . Thus, their storage requirements are only two vector lanes. This reduces the used width of the register file ports and internal buses, thus saving power.

Computing a uniform or affine result function of uniform and affine inputs can be performed using only one or two Scalar Processing (SP) units with a throughput of one cycle instead of the full SIMD width during two cycles. Indeed, most arithmetic operations on affine vectors can be reduced to operations on the base and stride.

6 Technical issues

Some issues may limit the efficiency of the proposed method and must be overcome.

Partial writes. GPUs handle branch divergence using predication. A predicated instruction does not write in every lane of its output register, keeping some of them in their previous state. In this case, even if the output value is uniform (or affine), the uniform (affine) property cannot be guaranteed for the destination register.

Half registers. The Tesla architecture allows access to lower/higher 16-bit sub-registers inside regular 32-bit registers. To handle this correctly, separate tags are needed for the lower, higher and whole parts to correctly track uniform/affine information.

Overflows. An arithmetic overflow may occur in a lane of an affine register, even if the base and stride are both representable. Overflows have no direct consequences when using two's-complement arithmetic, but casts between signed and unsigned formats of various sizes can occur, resulting for instance in an overflowing 16-bit affine value being extended into a non-affine 32-bit value.

This problem can be worked around by checking for overflows when performing affine computations, and re-issue the offending instruction as a vector operation when one is detected. Support for re-issuing instructions is already present to handle bank conflicts in the constant cache and scratchpad memory. As overflows should not occur in address calculations of correct programs, we expect it to be a rare occurrence. Indeed, we did not encounter this case in any of the benchmarks we ran.

Conversions from affine to generic vector. When an affine operand is combined with a vector operand, it needs to be first converted to a vector. As long as strides lengths are restricted to small powers of two, this can be implemented efficiently in hardware. However, it may be advantageous to reuse the conventional SIMD ALUs to perform the conversion, then re-issue the instruction if this situation is infrequent enough.

7 Results and validation

Figures 4 and 5 represent the respective proportions of uniform and affine operand captured with the proposed technique. We observe that on average, 19 % of inputs and 11 % of outputs can be identified as uniform data. These ratio go up to 34 % and 22 % respectively when considering affine data.

This means that the proposed methods reduce the bus activity between the register file and functional units for 34 % of the reads transfers. Likewise, the activity within the functional units can be reduced during 22 % of the operations executed in GPGPU computations. The power reduction brought by this technique, proportional to the activity reduction, is known to be of a critical issue for GPU [10]. Future works have to precisely quantify it.

It can be noted that the tag technique is not optimal, as it fails to detect some uniform and affine vectors. This is mostly due to the partial write effect as described in Section 6, and complex address calculations involving multiplication, division or modulo operations. Further work may improve the accuracy of the detection.

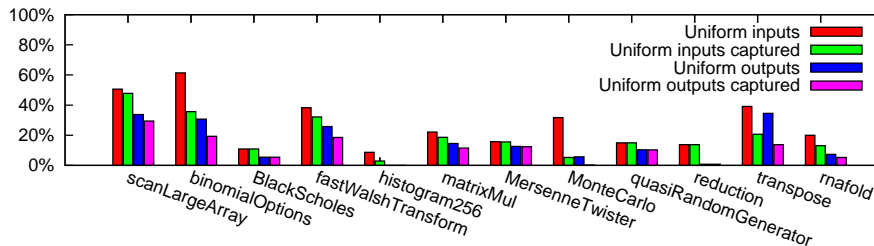


Fig. 4. Proportion of uniform operands in registers captured using our technique.

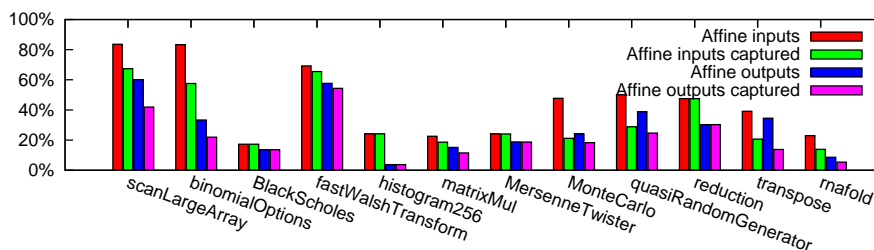


Fig. 5. Proportion of affine operands in registers captured using our technique.

8 Conclusion

In this paper, we presented a technique to exploit two forms of value locality specific to vector computations encountered in GPUs. The first one corresponds to the uniform pattern present when computing conditions which avoid divergence in sub-vectors. The second one corresponds to the affine pattern used to access memory efficiently. An analysis on common programs used in the field of GPGPU showed that both of them are common. The novel idea of using both forms of value locality with the proposed modifications significantly reduces the power required for data transfers between the register file and the functional units as well as the power drawn by the SIMD arithmetic units. Future work will focus on improving the accuracy of the hardware-based dynamic technique presented in this article, as well as considering software-based static implementations.

Acknowledgments

We thanks John Owens for his valuable comments on this work and Guillaume Rizk for the discussion related to bioinformatics applications. This work was partly supported by the french ANR BioWic.

References

1. NVIDIA: NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.2. (2009)
2. Lindholm, J.E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* **28**(2) (2008) 39–55
3. NVIDIA: The CUDA Compiler Driver NVCC, Version 2.0. (2008)
4. Bakhoda, A., Yuan, G., Fung, W.W.L., Wong, H., Aamodt, T.M.: Analyzing CUDA workloads using a detailed GPU simulator. In: proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Boston (April 2009) 163–174
5. Diamos, G., Kerr, A., Kesavan, M.: Translating GPU binaries to tiered SIMD architectures with Ocelot. Technical Report GIT-CERCS-09-01, Georgia Institute of Technology (2009)
6. Lindholm, E., Siu, M.Y., Moy, S.S., Liu, S., Nickolls, J.R.: Simulating multiported memories using lower port count memories. US Patent US 7339592 B2 (March 2008) NVIDIA Corporation.
7. Balakrishnan, S., Sohi, G.S.: Exploiting value locality in physical register files. In: MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (2003) 265
8. Rizk, G., Lavenier, D.: GPU accelerated RNA folding algorithm. In: Computational Science – ICCS 2009. Volume 5544 of LNCS., Springer (2009) 1004–1013
9. Mueller, S., Jacobi, C., Oh, H.J., Tran, K., Cottier, S., Michael, B., Nishikawa, H., Totsuka, Y., Namatame, T., Yano, N., Machida, T., Dhong, S.: The vector floating-point unit in a synergistic processor element of a CELL processor. In: 17th IEEE Symposium on Computer Arithmetic (ARITH-17). (June 2005) 59–67
10. Collange, S., Defour, D., Tisserand, A.: Power Consumption of GPUs from a Software Perspective. In: ICCS 2009. Volume 5544 of Lecture Notes in Computer Science., Springer (2009) 922–931