hal-00359342, version 4 - 24 Sep 2009

Barra, a Parallel Functional GPGPU Simulator

Sylvain Collange, David Defour and David Parello

ELIAUS, University of Perpignan, firstname.lastname@univ-perp.fr

Abstract. We present a GPU functional simulator targeting GPGPU based on the UNISIM framework which takes unaltered NVIDIA CUDA executables as input. It simulates the native instruction set of the Tesla architecture at the functional level and generates detailed execution statistics. Simulation speed is competitive with the less-accurate CUDA emulation mode thanks to optimizations which exploit the inherent parallelism of GPGPU applications. Furthermore, it opens the way for GPU microarchitecture design space exploration.

1 Introduction

As Graphics Processing Units (GPUs) gained in flexibility through high-level languages such as CUDA, interest for the acceleration of non-graphics tasks (GPGPU) raised thanks to the high computational power of GPUs. Therefore we are witnessing a tremendous growth in the usage of GPUs for high-performance computation solutions. Commodity graphics hardware is rapidly evolving, with each successive generation adding new features to accelerate execution of graphics routines as well as high performance computing software. Furthermore, architectures of modern graphics processors are largely secret, vendors being reluctant to release architectural details. New hardware features are the result of design space exploration techniques based on architecture simulation which helps manufacturers determine their validity and performance. However few GPU simulators are freely available because of the tremendous manpower required in terms of development and validation.

The complexity and performance of modern GPUs provides significant challenges for researchers interested in exploring architectural innovations and modeling fine-grained effects as it is already the case for CPUs. Functional and cycle-level simulation has long been used by CPU architects to study the effects of architectural and microarchitectural design changes. Functional simulators form the essential first blocks of timed simulators such as cycle-level or transaction-level simulators.

We present a modular and parallel simulator based on the UNISIM framework to perform functional simulation of a GPU targeting GPGPU named *Barra*. Our framework can be broken down into two broad areas: First the simulator of the hardware structures and functional units of the GPU, and second, the driver simulator which loads the input programs, perform management tasks and emulate the graphics/GPGPU driver.

hal-00359342, version 4 - 24 Sep 2009

We chose the NVIDIA architecture due to the wide acceptance of the CUDA language in the field of GPGPU[1]. Barra allows the user to monitor activities of computational units, communication links, registers and memories. Moreover, as Barra is integrated in a open structural simulation framework, it allows timed GPU modular simulators for design space exploration to be built upon it.

An overview of simulation and the CUDA framework is given in Section 2. A general view of the proposed framework and features of our simulator and driver are presented in Section 3. Section 4 presents our approach to simulator parallelization. Validation and performance comparison are respectively given in sections 5 and 6.

2 Context

2.1 Simulation

The availability of CPU simulators in the 1990's for superscalar architectures was the starting point of various academic and industrial researches in the computer architecture community. Simulation can be done at various levels, depending on the accuracy required. Cycle-level simulators are cycle accurate models characterized by a high accuracy on performance evaluation with respect to the real hardware. Transaction-level simulators are mostly based on functional models and focus on timing communications. The fastest simulation is done at functional-level, which mimics the processor behavior in a simulated environment.

The cycle-level simulator SimpleScalar [4] was at the origin of various works accompanying the success of superscalar processors in the late 1990's. However this simulator was known to be unorganized and difficult to modify and other attempts followed. SimpleScalar alternatives were proposed for multicore simulation [15] or full-system simulation [6,14,23]. Concerning GPUs, simulation frameworks targeting the graphics pipeline were introduced such as the cycle-level simulator Attila [16] or the transaction-level simulator Qsilver [26]. However, the architectural issues are different than those of a many-core parallel coprocessor such as a modern GPU.

Following the release of CUDA, GPU simulators putting an emphasis on parallel computing have been proposed. The Ocelot framework, a compiler infrastructure built around the NVIDIA PTX intermediate language, offers an emulator which runs CUDA programs [9]. As a virtual machine, it is not bound to a specific architecture and focuses on software implementation simplicity. GPUSim [5] is a cycle-level many-core simulator based on SimpleScalar. It simulates an original GPU-like architecture which uses the abstract PTX language as its ISA.

[1] www.nvidia.com/cuda

hal-00359342, version 4 - 24 Sep 2009

2.2 Using a modular simulation framework: UNISIM

To assist software development of simulators, multiple modular simulation frameworks [2,3,22] have been developed during the last decade. The common appealing feature of such environments is the ability to build simulators from software components mapped to hardware blocks. Modular frameworks can be compared on *modularity*, *tools* and *performances*.

To provide modularity, all environments suggest that modules share *architecture interfaces* to allow module sharing and reuse. Some of them strongly enforce modularity by adding *communication protocols* to distribute the hardware control logic into modules as proposed by LSE [3], MicroLib [22] and UNISIM [2].

The UNISIM environment provide GenISSLib, a code generator that generates an instruction decoder from a high-level description of the instruction set. The generated decoder is based on a cache containing pre-decoded instructions. On their first encounter, binary instructions are decoded and added to the cache. Subsequent executions of the same instruction simply require look-up of the decoded instruction in the cache. Furthermore, the description language allows the user to add functionalities.

As architecture and software complexity increases, simulator performance becomes the main issue of modular frameworks. Two solutions have been proposed to tackle this issue. Both use a trade-off between accuracy and simulation speed. The first solution is sampling techniques [28] suitable for single-thread simulation. The second solution is better suited for multicore and system simulation. It suggests to model the architecture at a higher level of abstraction with less details than cycle-level modeling: transaction-level modeling (TLM) [25]. To our knowledge, UNISIM is as of today the only modular environment offering both cycle-level and transaction-level modeling based on the SystemC standard [11]. Finally, recent techniques [21] have been proposed to improve cycle-level simulation of multicore architectures.

2.3 The CUDA environment

The Compute Unified Device Architecture (CUDA) is a vector-oriented computing environment developed by NVIDIA [18]. This environment relies on a stack composed of an architecture, a language, a compiler, a driver and various tools and libraries.

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 support. All current 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 chip multiprocessing (CMP), simultaneous multithreading (SMT) and SIMD processing [13]. Figure 1 describes the hardware organization of such a processor. Each multiprocessor contains the logic to fetch, decode and execute 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

hal-00359342, version 4 - 24 Sep 2009

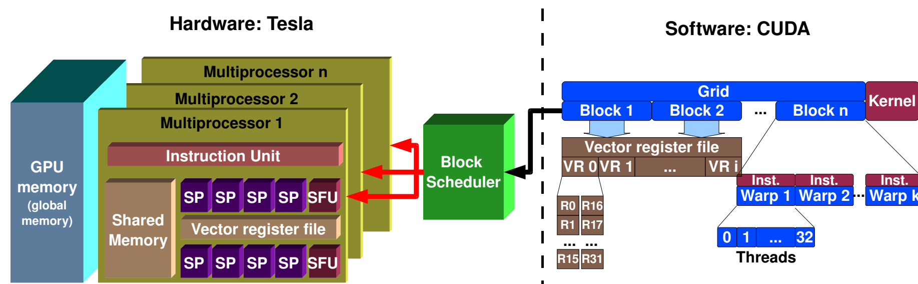

Fig. 1. Processing flow of a CUDA program.

extensions to indicate if a function is executed on the CPU or the GPU. Functions executed on the GPU are called *kernels* and follow the single-program multiple-data (SPMD) model. CUDA lets the programmer define which variables reside in the GPU address space and specify the kernel execution domain across different granularities of parallelism: *grids*, *blocks* and *threads*.

Several memory spaces on the GPU match this organization. Each thread has its own *local* memory space, each block has a distinct *shared* memory space, and all threads in a grid can access a single *global* memory space and a read-only *constant* memory space. To avoid race conditions, a synchronization barrier instruction can synchronize all threads inside a block, but not between blocks. Therefore, direct communication is possible inside a block but not across blocks, as the scheduling order of blocks is not defined.

This logical organization is mapped to the physical architecture. First, threads are grouped together in so-called *warps*, which contain 32 threads each in the Tesla architecture. Each warp follows a specific instruction flow, with all its threads running in lockstep, in an SIMD fashion. Then, each multiprocessor processes one or more blocks at a given time depending on the availability of hardware resources (register file and shared memory). Warp instructions are interleaved in the execution pipeline by hardware multithreading. For instance, the GT200 implementation supports up to 32 warps simultaneously. This technique helps hide the latency of streaming transfers, and thus allows the memory subsystem to be optimized for throughput rather than latency. Finally, blocks are scheduled on multiprocessors, taking advantage of CMP-type parallelism.

Likewise, the logical memory spaces are mapped to physical memories. Both local and global memories are mapped to uncached off-chip DRAM, while shared memory is stored on a scratchpad zone inside each multiprocessor, and constant memory is accessed through a cache present inside each multiprocessor.

To assist applications development, several tools are provided in the CUDA environment. First, a built-in emulation mode runs user-level 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. NVIDIA also includes a debugger since CUDA 2.0

hal-00359342, version 4 - 24 Sep 2009

[19]. Finally, the CUDA Visual Profiler allows performance evaluation of kernels using hardware counters on the GPU.

3 The Barra functional simulator

Barra is divided in two parts. The first replaces the CUDA software stack, while the second simulates the actual GPU.

3.1 CUDA driver emulator

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. The Barra driver is placed inside a shared library that has the same name and exports the same symbols as NVIDIA's proprietary one *libcuda.so*, so that function calls destined to the GPU can be dynamically captured and rerouted to the simulator. Thus, the user can choose whether to execute an unmodified CUDA program on the GPU or simulate it on Barra by setting an environment variable.

The proposed Barra driver includes major functions of the Driver API so that CUDA programs can be loaded, decoded and executed on the simulator. It plays roughly the same role as the operating system and loader do in a CPU simulator.

Though the CUDA model comprises logically separated memories (constant, local, global, shared) and the Tesla hardware contains physically separated memories (DRAM and shared memories), all types of memory are mapped at different addresses in a single physical address space in Barra. The virtual address space is currently mapped directly to the physical space. We will provide virtual address translation in the future, permitting stricter address checking, multiple CUDA contexts and allowing the performance modeling of TLBs.

3.2 Barra and Tesla ISA decoding

The Tesla instruction set was introduced with the Tesla architecture in 2005. Since that time NVIDIA worked on tools (debugger, emulator, compiler, libraries, ...), optimized and debugged them, making that ISA mature. NVIDIA claims that the ISA is not fixed and might change in the future. However, given the investment in time and manpower related to the development, validation and optimization to design an ISA from scratch, it is likely that NVIDIA will avoid such situation unless forced by major architectural changes. Most compiler optimizations also happen during the compilation from PTX to the Tesla ISA, including optimizations that can affect the semantics of the program such as fusion of additions and multiplications into either truncated of fused multiply-and-add. Table 1 in Section 5 shows the number of static PTX instructions and the number of static Tesla instructions for benchmarks and kernels. As PTX to Tesla compilation is a complex process involving optimizations, it is difficult to

hal-00359342, version 4 - 24 Sep 2009

correlate these numbers. Thus simulating the PTX instruction set may lead to poor accuracy. Therefore, we simulate the Tesla ISA directly to keep as close to a real execution as possible.

However, NVIDIA, unlike AMD [1], does not document this ISA. Thus, we completed the harnessing work done in the decuda project [27] to recover the specifications of the Tesla 1.0 ISA.

This instruction set is designed to run compute-intensive floating-point programs. As such, it is a four-operand instruction set centered on single-precision floating-point operations. It includes a truncated multiply-and-add instruction and transcendental instructions evaluating reciprocal, square root reciprocal, base-2 exponential and logarithm, sine and cosine accurate to 23 bits. Transparent execution of thread-based control flow in SIMD is possible thanks to specific branch instructions containing reconvergence information.

Most instructions are 64-bit wide, but some instructions have an alternate 32-bit encoding. Another encoding allows embedding a 32-bit immediate inside a 64-bit instruction word.

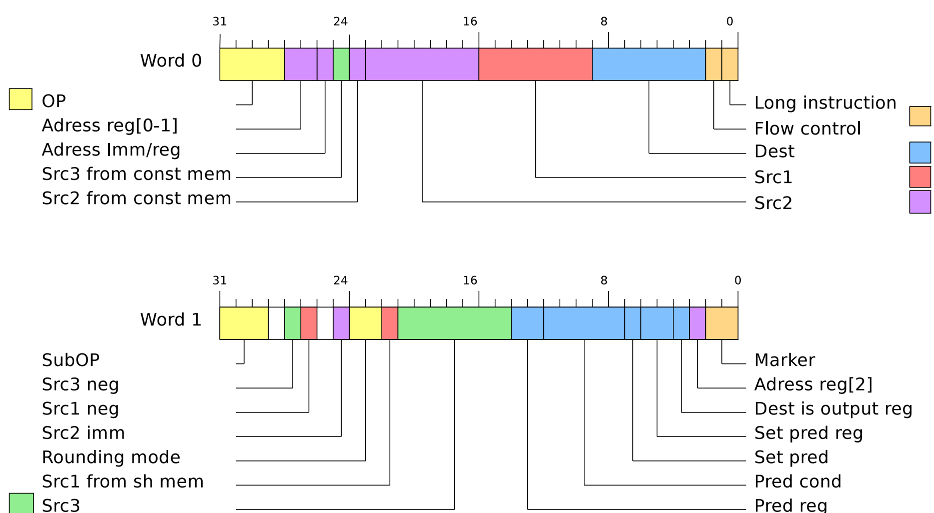

Fig. 2. Opcode fields of a MAD instruction.

An example of the instruction format of a floating-point multiplication-addition instruction in single precision (MAD) is given in figure 2. This instructions can address up to 3 source operands (indicated by Src1, Src2 and Src3), addressing either General Purpose Registers (GPRs), shared memory (sh mem), constant memory (const mem) or designate an immediate constant (imm). The destination operand is indicated by Dest. Extra fields such as predication control and instruction format are included. Each part is mostly orthogonal to other parts and can be decoded independently.

hal-00359342, version 4 - 24 Sep 2009

Taking advantage of this orthogonality, we use the GenISSLib library to generate six separate decoders working on the whole instruction word (opcode, destination and predicate control, src1, src2, src3, various flags), each being responsible for a part of the instruction, while ignoring all other fields.

3.3 Instruction execution

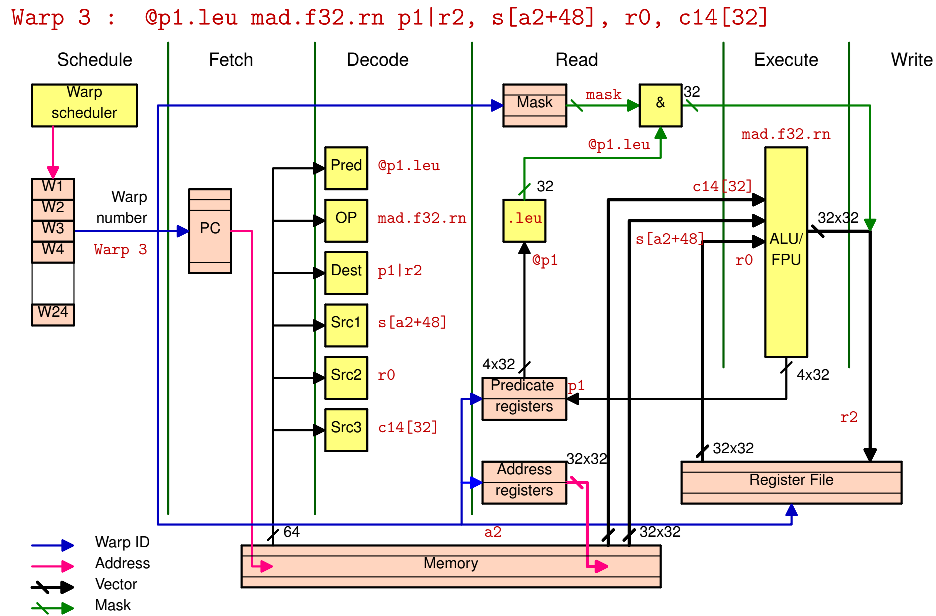

Fig. 3. Functional overview of a multiprocessor execution pipeline during the execution of a MAD instruction.

Instructions are executed in Barra according to the model described in Figure 3. First a scheduler selects the next warp for execution with the corresponding program counter (PC). Then the instruction is loaded and decoded as described in section 3.2. Then operands are read from the register file or from on-chip memories (shared) or caches (constant). As the memory space is unified, a generic gather mechanism is used. Then the instruction is executed and the result is written back to the register file. Integer and floating-point instructions can optionally update a flag register containing zero, negative, carry and overflow flags.

Evaluation of transcendental functions in the Tesla architecture is a two step process: A range reduction based on a conversion to fixed point arithmetic followed by a call to a dedicated unit. This unit is described in [20] and involves

hal-00359342, version 4 - 24 Sep 2009

dedicated operators with tabulated values. An exact simulation of this unit will require exhaustive tests on every possible values. Therefore, the current implementation of transcendental function evaluations in Barra is based on a similar range reduction followed with a call to the host standard math library.

All single-precision floating-point arithmetic operations flush all NaNs in input and output to zero as the architecture requires.

3.4 Simulation of fine-grained parallelism

As a throughput-oriented architecture, Tesla differs in several aspects from conventional multi-core processors.

Register management GPRs are dynamically split between threads during kernel launch, allowing to trade less parallelism for more registers per thread. Barra maintains a separate state for each active warp in the multiprocessor. These states include a program counter, address and predicate registers, a hardware stack for control-flow execution, a window to the assigned register set, and a window to the shared memory. During functional simulation, warps are scheduled with a round-robin policy.

Multiprocessors of Tesla-based GPUs have a multi-bank register file partitioned between warps using sophisticated schemes [12]. This allows a space-efficient packing and minimizes bank conflicts. However, the specific register allocation policy bears no impact on functional behavior, apart from deciding how many warps can have their registers fit in the register file. Therefore, we opted for a plain sequential block allocation inside a single unified register file.

Warp scheduling Each warp has a state flag indicating whether it is ready for execution. At the beginning of the execution, each running warp has its flag set to *Active* while other warps have their flag set to *Inactive*. At each step of the simulation, an *Active* warp is selected to have one instruction executed using a round-robin policy.

When a synchronization barrier instruction is encountered, the current warp is marked as *Waiting*. If all warps are either *Waiting* or *Inactive*, the barrier has been reached by all warps, so *Waiting* warps are put back in the *Active* state.

A specific marker embedded in the instruction word signals the end of the kernel. When encountered, the current warp is flagged as *Inactive* so that it is ignored by the scheduler in subsequent scheduling rounds. Once all warps of running blocks have reached the *Inactive* state, a new set of blocks is scheduled to the multiprocessor.

Branch handling Thanks to dedicated hardware, the Tesla architecture allows divergent branches across individual threads in a warp to be executed transparently [8]. This is performed using a hardware-managed stack of tokens containing an address, an ID and a 32-bit mask. The ID allows forward branches, backward branches and function calls to share a single stack (fig. 4).

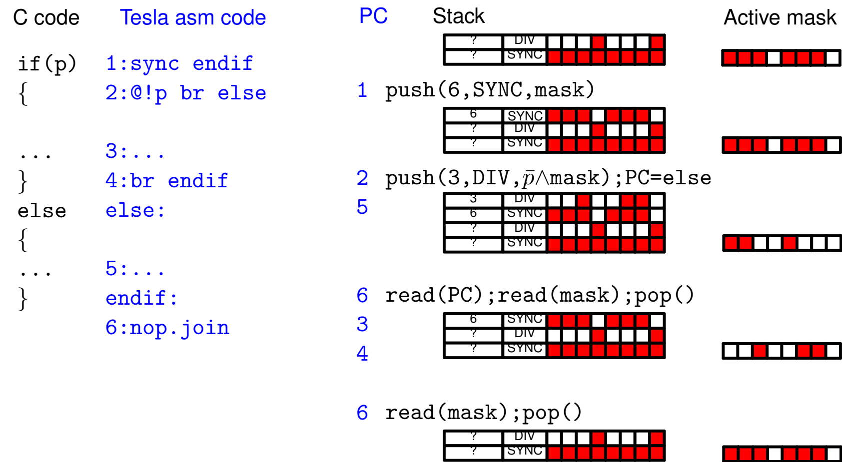

Fig. 4. Example of SIMD forward branch.

4 Simulator parallelization

Data-parallel programs such as CUDA codes expose a significant amount of explicit parallelism. This fact can be exploited to accelerate functional simulation.

Simulating many-core with multi-core The CUDA programming model is designed to reduce the coupling between multiprocessors to a minimum. As such, the block scheduling order is undefined, no global synchronization is possible and communications between blocks are minimized. By relaxing the memory consistency model, this enables efficient and scalable hardware implementations. We exploit this decoupling ability in software by simulating each multiprocessor in a different host thread.

Our tests suggest that the CUDA block scheduler dispatches blocks across multiprocessors in a round-robin fashion, and performs a global synchronization barrier between each scheduling round.

We followed a slightly different approach to block scheduling in Barra by distributing the scheduling decisions across worker threads. It follows the same static scheduling order as the CUDA hardware scheduler, but removes the need to perform a global synchronization after each scheduling round.

At warp level, the fine-grained multithreading is still simulated as described in section 3.4.

Simulators of general-purpose processors need to handle dynamic memory allocation and self-modifying code in simulated programs. This requires using cache-like data structures that can grow as needed to store data and instructions. Sharing such structures in a multithreaded environment requires locking techniques, which can be challenging to implement and validate and can impact

hal-00359342, version 4 - 24 Sep 2009

hal-00359342, version 4 - 24 Sep 2009

performance. Fortunately, the CUDA programming model prevents this kind of irregular behavior in the simulated code by following the Harvard architecture model and requiring the user to explicitly allocate before the execution begins all the memory that will be accessed inside a GPU kernel. Accordingly, we can pre-allocate all data and instruction memories of the simulator in lock-free data structures.

The strong isolation rules between blocks of the CUDA programming model benefits hardware implementations as well as simulation on multi-cores.

Simulating SIMD with SIMD GPUs make heavy use of SIMD to execute regular code efficiently. The Tesla architecture executes 32-way SIMD instructions. This amortizes the hardware cost of instruction fetching, decoding and scheduling. This benefit also applies to simulation, where the ratio of simulation time dedicated to the actual execution of instructions drops as architecture complexity increases.

To further benefit from the regularity introduced by the SIMD model, we implement the basic single-precision floating-point instructions (add, mul, mad, reciprocal, reciprocal square root) with SSE SIMD instructions using C intrinsics when they are available. The Denormal-As-Zero and Flush-To-Zero SSE flags are enabled to reflect the behavior of the GPU operators as well as preventing denormal values from slowing down the simulation. The implementation of floating-point instructions, including min and max, follows the same NaN propagation behavior as the GPU as long as all input NaNs are encoded as QNaNs.

Both multithreading and SIMD enable GPU simulation to run efficiently and accurately on current multi-core processors.

5 Validation

We used examples from the NVIDIA CUDA SDK to compare the execution on our simulator with real executions on Tesla GPUs. Even though these examples are not initially meant to be used as benchmarks, they form currently the most standardized test suite of CUDA applications. As code examples, they reflect the best practices in CUDA programming.

Most of these examples use a reduced data-set size when run in emulation mode. We made sure they always run the complete data-set. We inserted synchronization barriers where it was missing to get correct timings.

Executions of the examples from Table 1 on Barra give the same results than executions on GPUs, except for the ones that use transcendentals instructions, as it was expected given the difference in implementation. It should be noted that the official CUDA emulation mode does not offer the same accuracy. For instance, results returned by the dwtHaar1D example from the CUDA SDK differ by 0.5 units in the last place (ulps) on average and by 1681 ulps in the worst case between CUDA emulation and execution on a GPU.

hal-00359342, version 4 - 24 Sep 2009

Program	Kernel	St. PTX	St. ASM	Dyn. ASM
binomialOptions	binomialOptionsKernel	153	114	401,131,008
BlackScholes	BlackScholesGPU	134	99	5,201,694,720
convolutionSeparable	convolutionRowGPU	67	52	38,486,016
	convolutionColGPU	99	100	38,338,560
dwtHaar1D	dwtHaar1D	92	87	10,204
fastWalshTransform	fwtBatch1Kernel	110	107	57,606,144
	fwtBatch2Kernel	47	46	54,263,808
	modulateKernel	26	24	2,635,776
matrixMul	matrixMul	83	114	66,880
MersenneTwister	RandomGPU	159	223	31,526,528
	BoxMuller	86	68	16,879,360
MonteCarlo	MonteCarloOneBlock...	122	132	27,427,328
reduction	reduce5_sm10	62	40	4,000
	reduce6_sm10	75	59	20,781,760
scanLargeArray	prescan<false,false>	107	94	14,544
	prescan<true,false>	114	102	423,560,064
	prescan<true,true>	122	108	257,651
	uniformAdd	28	27	42,696,639
transpose	transpose_naive	29	28	1,835,008
	transpose	52	42	2,752,512

Table 1. Benchmarks and kernels we consider along with their static PTX instruction count (St. PTX), and static and dynamic assembly instruction counts (St. ASM and Dyn. ASM respectively).

During functional simulation, we collected statistics about instruction type, operands, branch divergence, memory access type on a per-static-instruction basis. We did not observe any variation in the statistics generated between single-threaded and parallel functional simulation.

We compare these statistics with the hardware counters during a run on a GPU by using the CUDA Profiler, which provides statistics on a per-kernel-execution basis. GPU hardware counters are currently usable on one texture processing cluster (TPC) only[2]. Therefore an extrapolation is needed to estimate the performance of the whole kernel. The precise meaning, unit and scale used for each counter is not documented. As the profiler documentation reports, “users should not expect the counter values to match the numbers one would get by inspecting kernel code.” However, we were able to match the value of most of these counters with statistics obtained from simulation. We report the relative differences observed for instruction, branch, branch divergence and memory transaction count in figure 5.

[2] A texture processing cluster is a hardware structure containing two to three multiprocessors sharing memory access units. While the profiler documentation claims the instrumentation is done on a multiprocessor basis, our results suggest that it is done at TPC granularity instead.

hal-00359342, version 4 - 24 Sep 2009

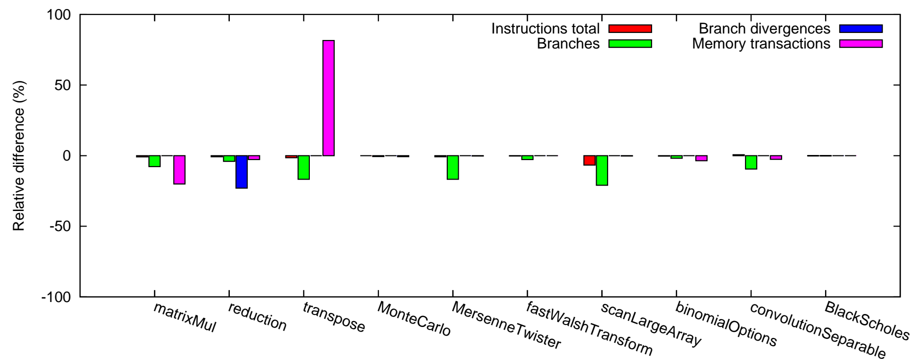

Fig. 5. Relative difference between Barra statistics and GPU hardware counters.

The instruction counts are consistent, except in the *scanLargeArray* benchmark. A closer analysis of the performance counters reveals that the kernel prescan<true,false> is launched multiple times on one single block. To mitigate the effect of such load imbalance, the profiler seems to select a different TPC to instrument at each kernel call in round-robin. However, as the number of calls (202) is not multiple of the number of TPCs (8), the load imbalance effect remains and affects the counters.

We were not able to find out the exact meaning of the branch instruction counter. We found it to be consistently equal or higher than the number of all control flow instructions encountered in Barra.

The *transpose* application, and *matrixMul* to a lesser extent, show discrepancies in the number of memory instructions reported. The *transpose* benchmark is known to be affected by a phenomenon dubbed *partition camping*, which occurs when most memory accesses over a period of time are directed to a narrow subset of all DRAM banks, causing conflicts [24]. We simulated and profiled the *transposeNew* example, which implements the same algorithm while avoiding partition camping and obtained consistent results, which confirms that the discrepancy we observed is caused by this phenomenon. We are currently investigating whether the difference in memory transaction count is due to sampling artifacts or actually reflect an hardware mechanism.

As it was discussed in section 3.2, the Tesla ISA is undocumented and some instructions that we have not yet encountered will not be correctly handled by Barra. We use both synthetic test cases such as those provided with *decuda* and real-world programs such as the CUDA SDK examples to check and extend the instruction coverage.

6 Simulation speed results

We compared and reported in figure 6 the execution time of the benchmarks in CUDA emulation mode, in a single-threaded functional simulation with Barra,

hal-00359342, version 4 - 24 Sep 2009

inside the CUDA-gdb debugger with a native execution on a GPU. Reported time is normalized to the native execution time for each program. The test platform is a 3.0 GHz Intel Core 2 Duo E8400 with a NVIDIA GeForce 9800 GX2 graphics board on an Intel X48 chipset, running Ubuntu Linux 8.10 x64 with gcc 4.3 and CUDA 2.2. The -O3 option was passed to gcc. The debugger from CUDA 2.3 Beta was used as it is the first version compatible with our architecture. When run within the CUDA debugger, the *MonteCarlo* and *binomialOptions* benchmarks did not complete within 24 hours, so we could not report their performance. We did not include these benchmarks when computing the average of CUDA-gdb timings.

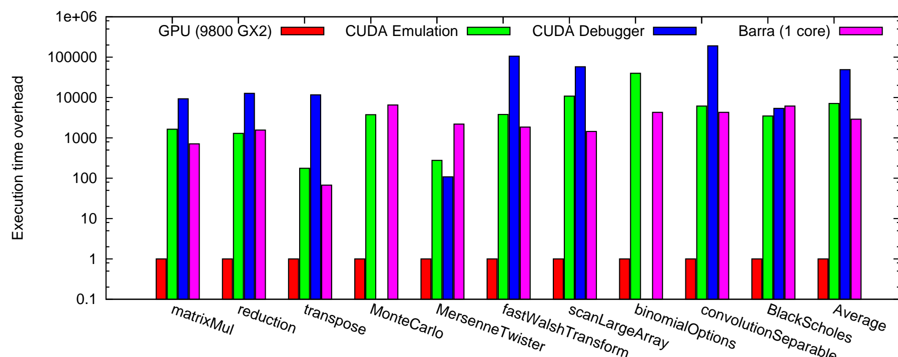

Fig. 6. Compared execution time of native execution, source-level emulation by the CUDA emulation mode, run inside the CUDA debugger and functional simulation with Barra, normalized by native execution time.

We observe that even when run on one core, Barra is competitive with the CUDA emulation mode in terms of speed, as well as being more accurate. This is likely because simulating fine-grained intra-block multithreading using user-managed threads as the emulation mode does causes thread creation and synchronization overhead to dominate the execution time.

The CUDA debugger usually suffer from an even greater overhead, likely caused by synchronizations across the whole system and data transfers to and from the CPU after the execution of each instruction.

To quantify the benefits of simulator parallelization, we simulated the same benchmarks on a quad core Intel Xeon E5410-based workstation running Red Hat 5 and gcc 4.1 with a number of threads ranging from 1 to 4. The average speedup is 1.90 when going from 1 to 2 cores and 3.53 when going from 1 to 4 cores. This is thanks to the CUDA programming model that reduces dependencies and synchronizations needed between cores. On the other hand, the CUDA emulation mode runs programs using user-managed threads and does not take advantage of multiple cores, which would require kernel-managed threads.

hal-00359342, version 4 - 24 Sep 2009

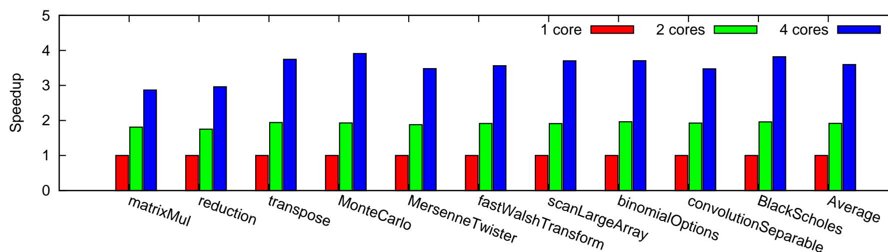

Fig. 7. Impact of the number of cores on parallel simulation speed.

We observe that the simulation time using Barra is similar to the emulation time using CUDA emulation even though Barra is more accurate, provides more flexibility and generates statistics for each static instruction. Thanks to the SIMD nature of Barra, we perform more work per instruction that amortize instruction decoding and execution control as in a SIMD processor. Moreover, integration into the UNISIM simulation environment enable faster simulation. For example, the cache of predecoded instructions used by GenISSLib as described in section 2.2 amortizes the instruction decoding cost. Its speed benefit is especially significant for GPU simulation, where the dynamic-to-static instruction ratio is particularly high, as evidenced by Table 1.

7 Conclusion and future work

We described the Barra driver and simulator, and showed that despite the unavailability of the description of the ISA used by NVIDIA GPUs, it is possible to simulate the execution of an entire CUDA programs at the functional level. The development of Barra inside the UNISIM environment allows users to customize the simulator, reuse module libraries and features proposed in the UNISIM repository. Thanks to this work it will be possible to test the scalability of programs without the need to physically test them on various configurations. As a side effect, our work enable a deeper understanding of GPU and many-core architecture through extensive analysis of the state-of-the-art NVIDIA Tesla architecture. Barra is distributed under BSD license, available for download[3] and is part of the UNISIM framework.The low-level placement of the Barra driver makes it programming language-agnostic and will allow a seamless integration into the NVIDIA OpenCL [17] software stack as it becomes publicly available.

Future work will focus on building performance models around the functional simulator, such as a modular transaction-level model. Our success in parallelizing functional simulation suggests that the relaxed memory consistency model of CUDA could also be exploited to accelerate transaction-level simulation through temporal decoupling [25] and simulation parallelization techniques such as parallel discrete event simulation [10]. The availability of a more accurate timing

[3] http://gpgpu.univ-perp.fr/index.php/Barra

hal-00359342, version 4 - 24 Sep 2009

model will open doors for the integration of other models such as power consumption [7].

References

1. Advanced Micro Device, Inc. *AMD R600-Family Instruction Set Architecture*, December 2008.
2. David August, Jonathan Chang, Sylvain Girbal, Daniel Gracia-Perez, Gilles Mouchard, David A. Penry, Olivier Temam, and Neil Vachharajani. UNISIM: An open simulation environment and library for complex architecture design and collaborative development. *IEEE Comput. Archit. Lett.*, 6(2):45–48, 2007.
3. David I. August, Sharad Malik, Li-Shiuan Peh, Vijay Pai, Manish Vachharajani, and Paul Willmann. Achieving structural and composable modeling of complex systems. *Int. J. Parallel Program.*, 33(2):81–101, 2005.
4. Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.
5. Ali Bakhoda, George Yuan, Wilson W. L. Fung, Henry Wong, and Tor M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In *proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 163–174, Boston, April 2009.
6. Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.
7. Sylvain Collange, David Defour, and Arnaud Tisserand. Power Consuption of GPUs from a Software Perspective. In *ICCS 2009*, volume 5544 of *Lecture Notes in Computer Science*, pages 922–931. Springer, 2009.
8. Brett W. Coon and John Erik Lindholm. System and method for managing divergent threads in a SIMD architecture. US Patent US 7353369 B1, April 2008. NVIDIA Corporation.
9. Gregory Diamos, Andrew Kerr, and Mukil Kesavan. Translating GPU binaries to tiered SIMD architectures with Ocelot. Technical Report GIT-CERCS-09-01, Georgia Institute of Technology, 2009.
10. Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.
11. The Open SystemC Initiative. SystemC. http://www.systemc.org/.
12. Erik Lindholm, Ming Y. Siu, Simon S. Moy, Samuel Liu, and John R. Nickolls. Simulating multiported memories using lower port count memories. US Patent US 7339592 B2, March 2008. NVIDIA Corporation.
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. Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.
15. Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33:2005, 2005.

hal-00359342, version 4 - 24 Sep 2009

16. Victor Moya, Carlos Gonzalez, Jordi Roca, Agustin Fernandez, and Roger Espasa. Shader performance analysis on a modern GPU architecture. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 355–364, Washington, DC, USA, 2005. IEEE Computer Society.
17. Aaftab Munshi. The OpenCL specification. *Khronos OpenCL Working Group*, 2009.
18. NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.0*, 2008.
19. NVIDIA. *CUDA-GDB : The NVIDIA CUDA Debugger, Version 2.2*, 2009.
20. Stuart F. Oberman and Michael Siu. A high-performance area-efficient multifunction interpolator. In Koren and Kornerup, editors, *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (Cap Cod, USA)*, pages 272–279, Los Alamitos, CA, July 2005. IEEE Computer Society Press.
21. David Parello, Mourad Bouache, and Bernard Goossens. Improving cycle-level modular simulation by vectorization. *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO'09)*, 2009.
22. Daniel Gracia Perez, Gilles Mouchard, and Olivier Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 43–54, Washington, DC, USA, 2004. IEEE Computer Society.
23. Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Trans. Model. Comput. Simul.*, 7(1):78–103, 1997.
24. Greg Ruetsch and Paulius Micikevicius. *Optimizing Matrix Transpose in CUDA*. NVIDIA CUDA SDK Application Note, 2009.
25. Gunar Schirner and Rainer Dömer. Quantitative analysis of the speed/accuracy trade-off in transaction level modeling. *Trans. on Embedded Computing Sys.*, 8(1):1–29, 2008.
26. J. W. Sheaffer, D. Luebke, and K. Skadron. A flexible simulation framework for graphics architectures. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 85–94, New York, NY, USA, 2004. ACM.
27. Wladimir J. van der Laan. Decuda and Cudasm, the cubin utilities package, 2008. http://www.cs.rug.nl/˜wladimir/decuda.
28. Thomas F. Wenisch, Roland E. Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.