Barra: a Parallel Functional Simulator for GPGPU

Caroline Collange, David Defour, David Parello, Marc Daumas

Barra: a Parallel Functional Simulator for GPGPU

Caroline Collange, Marc Daumas, David Defour and David Parello
ELIAUS-PROMES (UPVD) — Perpignan — FRANCE
Email: firstname.lastname@univ-perp.fr

***Abstract*—We present Barra, a simulator of Graphics Processing Units (GPU) tuned for general purpose processing (GPGPU). It is based on the UNISIM framework and it simulates the native instruction set of the Tesla architecture at the functional level. The inputs are CUDA executables produced by NVIDIA tools. No alterations are needed to perform simulations. As it uses parallelism, Barra generates detailed statistics on executions in about the time needed by CUDA to operate in emulation mode. We use it to explore the micro-architecture design spaces of GPUs.**

I. Introduction

We are witnessing a tremendous growth in the use of Graphics Processing Units (GPU) for the acceleration of non-graphics tasks (GPGPU). This is due to the huge computing power delivered by GPUs and the recent availability of CUDA, a high-level and flexible development environment. Meanwhile, commodity graphics hardware is rapidly evolving, adding new features with each successive generation to accelerate execution of graphics routines as well as high performance computing software.

Functional and cycle-level simulations have long been used by CPU architects to study the effects of changes in architectural and micro-architectural designs. New hardware features are proposed and validated by explorations of design spaces based on simulation. This methodology helps executives estimate costs and performances of proposals. In hierarchical design, functional simulators are used for uppermost blocks and timed simulators, such as cycle-level or transaction-level simulators, are used for inner blocks, when necessary.

Complex architectures of modern GPUs carry many significant challenges for researchers interested in exploring architectural innovations and modeling precisely the effects of changes, similarly to what is done for CPUs. Yet, architectures of modern GPUs are largely secret as vendors are reluctant to release architectural details and few GPU simulators are freely available because of the tremendous manpower required in development and validation.

We present a modular and parallel simulator based on the UNISIM framework to perform functional simulations of GPUs targeting GPGPU applications. It is named *Barra*. We chose the NVIDIA architecture due to the wide acceptance of CUDA[1] environment in GPGPU. Our framework integrates two broad functions:

- A simulator of the hardware structures and functional units of the GPU;
- A simulator of the driver which loads the input programs, performs management tasks and emulates the graphics-GPGPU driver.

Barra monitors the activity of computational units, communication links, registers and memories. As it is integrated in an open structural simulation framework, we may later build timed simulators of GPU modules for the exploration of some specific design spaces.

An overview of simulation and the CUDA framework is given in Section II. A general view of the proposed framework and features of our simulator and driver are presented in Section III. Section IV presents our approach to the parallelization of Barra. Validation and performance comparison are respectively given in Sections V and VI.

II. Context

A. Simulation

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

The SimpleScalar cycle-level simulator [4] was at the origin of various works accompanying the success of superscalar processors in the late 1990s. However this simulator was known to be unorganized and difficult to modify. Alternatives to SimpleScalar were proposed for multicore simulation [16] or full-system simulation [6], [15], [25]. Concerning GPUs, simulation frameworks targeting the graphics pipeline were introduced such as the Attila cycle-level simulator [17] or the Qsilver transaction-level simulator [28]. However, the architectural issues were different than those of many-core parallel coprocessors such as modern GPUs.

GPU simulators putting an emphasis on parallel computing have been proposed following the release of CUDA. The Ocelot framework is a compiler infrastructure built around the NVIDIA PTX intermediate language. It includes an emulator to run CUDA programs [11]. As a virtual machine, it is not bound to a specific architecture and its design goal is to deliver the most simple software implementation. GPGPUSim [5] is a cycle-level many-core simulator based on SimpleScalar. It

[1] http://www.nvidia.com/cuda.

simulates an original GPU-like architecture which uses the abstract PTX language as its ISA.

B. Using UNISIM modular simulation framework

Modular simulation frameworks [2], [3], [24] have been developed during the last decade to assist with software development of new simulators. 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 based on criteria of *modularity*, *tools* and *performances*.

All environments suggest that modules share some *architecture interfaces* to provide modularity by allowing module sharing and reuse. Some of them strongly enforce modularity by adding some *communication protocols* to distribute hardware control logic into modules as proposed by LSE [3], MicroLib [24] and UNISIM [2].

The UNISIM environment includes GenISSLib, a code generator that builds an instruction decoder from any 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 perform look-ups of the decoded instruction in the cache. The description language allows users to add some functionalities.

Simulation speed is becoming a main issue of modular frameworks as architecture and software complexity increases. Two solutions have been proposed to tackle this issue. Both make trade-offs between accuracy and simulation speed. The first solution uses sampling techniques [30] and is 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) [27]. To our knowledge, today, UNISIM is the only modular environment offering both cycle-level and transaction-level modeling based on the SystemC standard[2].

Recent techniques [23] have been proposed to improve cycle-level simulation of multicore architectures.

C. CUDA environment

The Compute Unified Device Architecture (CUDA) is a vector-oriented computing environment developed by NVIDIA [19]. It 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 that supports CUDA. CUDA-enabled GPUs are made of an array of *multiprocessors*. GPUs execute thousands of threads in parallel thanks to the combined use of chip multiprocessing (CMP), simultaneous multithreading (SMT) and SIMD processing [14]. Figure 1 describes the hardware organization of these processors. Each multiprocessor contains the logic required to fetch, decode and execute instructions.

[2]The Open SystemC Initiative, http://www.systemc.org/.

The hardware organization is tightly coupled with the parallel programming model of CUDA. The programming language used in CUDA is based on the C language with extensions to indicate that 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 programmers 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 are used on the GPU to 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. A synchronization barrier instruction can synchronize all threads inside a block to prevent some race conditions. It does not synchronize different blocks. Therefore, direct communications are possible inside blocks but not across blocks, as the scheduling order of blocks is not defined.

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

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.

Several tools are provided to assist applications development 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 operations, the scheduling policies and memory organization are different. NVIDIA also provides a debugger starting with CUDA 2.0 [20]. Finally, CUDA Visual Profiler provides some performance evaluation of kernels using hardware counters on the GPU.

III. Barra functional simulator

Barra contains two sets of tools. The first one replaces the CUDA software stack, while the second one simulates the actual GPU.

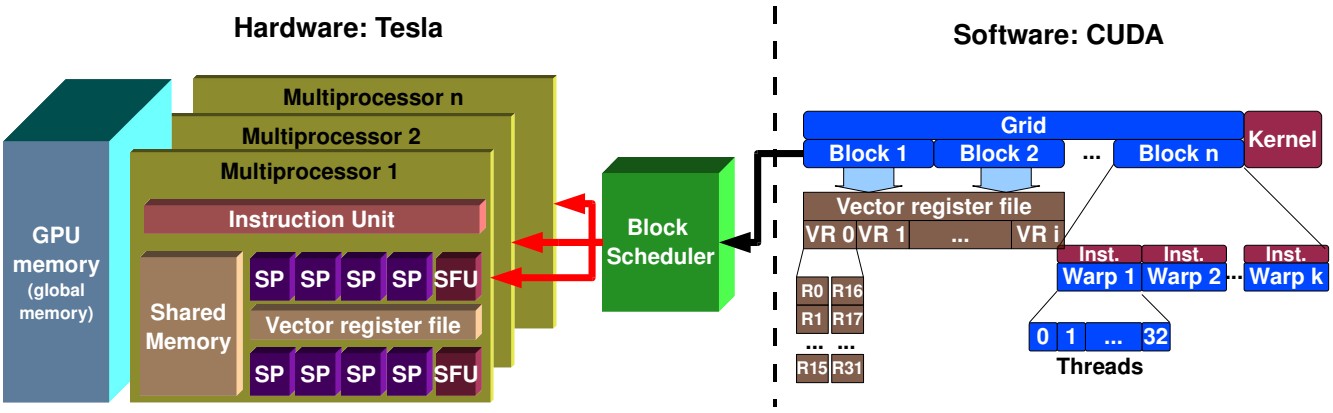

Fig. 1. Processing flow of a CUDA program.

A. 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 posted for the GPU are captured dynamically and rerouted to the simulator. By setting an environment variable, the user switches between executing an unmodified CUDA program on the GPU and simulating it on Barra.

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

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

B. Barra and Tesla ISA decoding

The Tesla instruction set was introduced with the Tesla architecture in 2005. Since that time NVIDIA has developped, debugged and optimized a toolset containing a compiler, a debugger, an emulator and many libraries. Though the ISA of the Fermi architecture [21] is not binary-compatible with the Tesla one, independent analysis[3] has shown that it keeps most of the traits of the Tesla ISA.

Table I in Section V shows the number of static PTX instructions and the number of static Tesla instructions for some benchmarks and kernels. It is difficult to correlate these numbers as PTX to Tesla ISA compilation is a complex process. Most compiler optimizations occur during this step, including optimizations that can affect the semantics of programs such as fusions of additions and multiplications into either truncated of fused multiply-and-adds. Simulation at the PTX instruction set level may lead to low accuracy. Therefore, we simulate directly Tesla ISA to remain as close as possible from what really occurs in GPUs. We recovered the specifications of Tesla 1.0 ISA as NVIDIA, unlike AMD [1], does not document its ISA. This was done by completing the harnessing work started in the decuda project [29].

This instruction set is designed to run compute-intensive floating-point programs. 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 for the 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 of a 32-bit immediate constant inside a 64-bit instruction word.

An example of the instruction format of floating-point multiplication-additions in single precision (MAD) is given in Figure 2. These instructions can address up to 3 source operands (indicated by Src1, Src2 and Src3) in General Purpose Registers (GPR), shared memory (sh mem), constant memory (const mem) or as immediate constants (imm). The destination operand is indicated by Dest. Extra fields such as predication control and instruction format are defined. Each piece of information is mostly orthogonal to the other pieces and can be decoded independently.

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 the other fields.

[3] http://0x04.net/cgit/index.cgi/nv50dis.

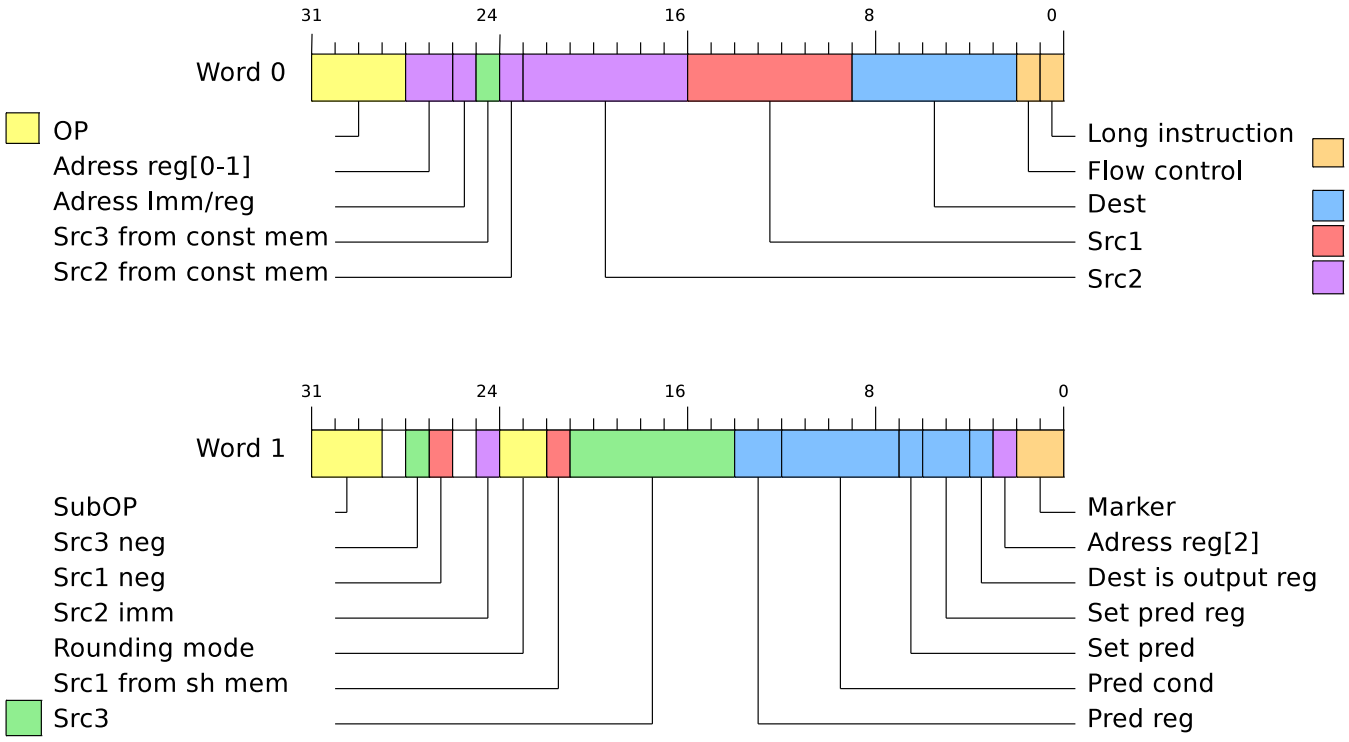

Fig. 2. Opcode fields of a MAD instruction.

C. Instruction execution

Instructions are executed in Barra according to the model described in Figure 3:

- A scheduler selects the next warp for execution with the corresponding program counter (PC).
- The instruction is loaded and decoded as described in Section III-B.
- 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.
- 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 is followed by a call to a dedicated approximation unit. This unit is described in [22]. It contains some dedicated operators using tabulated values. An exact simulation of this unit will require some exhaustive tests on every possible value in single precision. Barra's current implementation of transcendental functions is based on a similar range reduction followed with a call to the standard math library of the host.

Single-precision floating-point arithmetic operations flush all input and output NaNs to zero as specified by the architecture.

D. Simulation of fine-grained parallelism

Tesla differs in several aspects from conventional multi-core processors as it is a throughput-oriented architecture.

1) Register management: GPRs are dynamically split between threads during kernel launch, allowing to trade some reduced parallelism against a larger number of registers per thread. Barra maintains a separate state for each active warp in the multiprocessor. The state includes 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.

Multiprocessors of Tesla-based GPUs have a multi-bank register file partitioned between warps using sophisticated schemes [13]. This allows a space-efficient packing that 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.

2) Warp scheduling: Each warp has a state flag indicating whether it is ready for execution. At the beginning, 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.

The current warp is marked as *Waiting* when a synchronization barrier instruction is encountered. 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

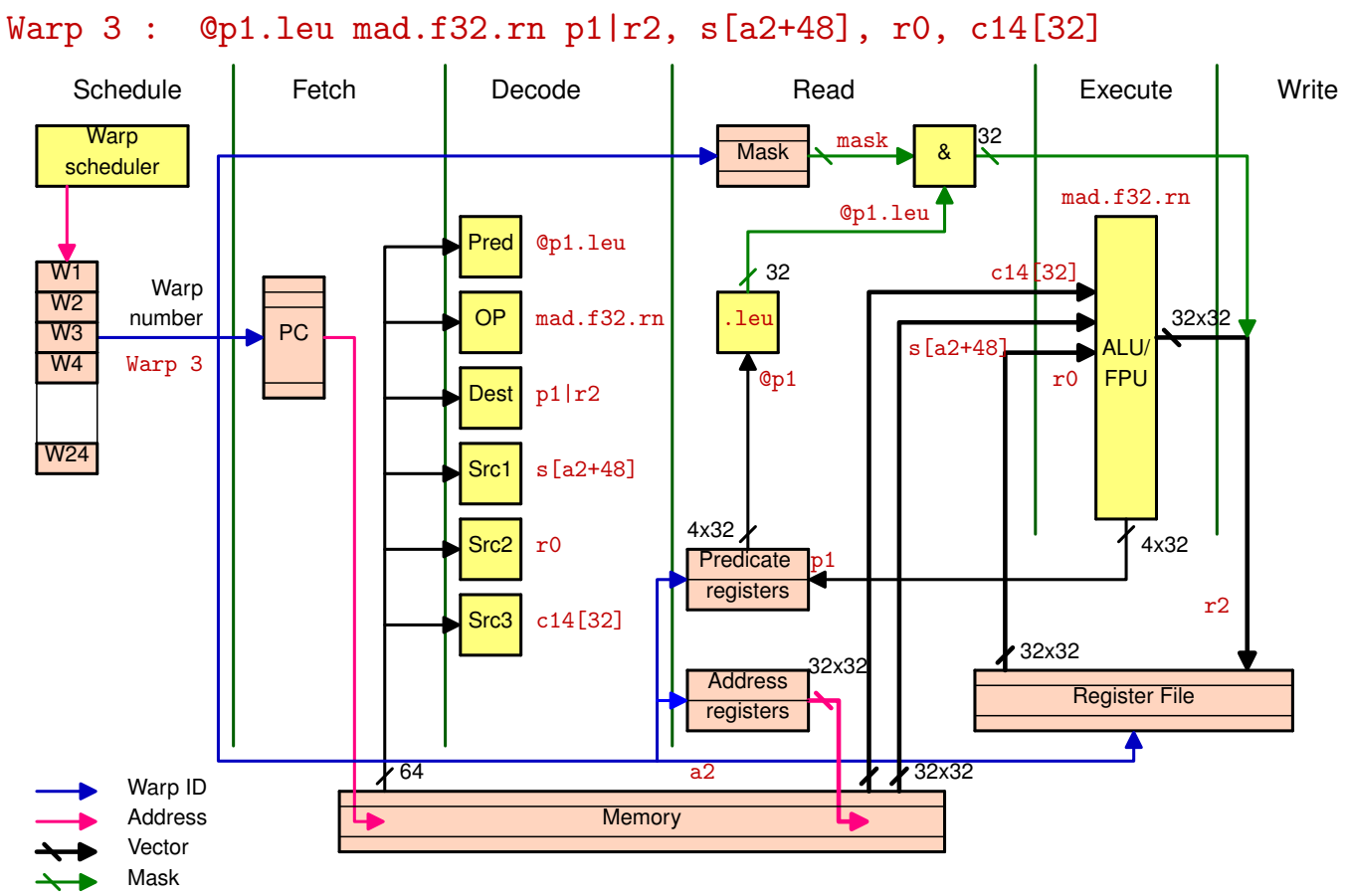

Fig. 3. Functional overview of a multiprocessor execution pipeline during the execution of a MAD instruction.

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. A new set of blocks is scheduled to the multiprocessor as soon as all warps of running blocks have reached the *Inactive* state.

3) Branch handling: Tesla architecture allows divergent branches across individual threads in a warp to be executed transparently thanks to some dedicated hardware [10]. 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 (see Figure 4).

IV. Simulator parallelization

Data-parallel programs such as kernels developed in CUDA expose a significant amount of explicit parallelism. This fact can be exploited to accelerate functional simulation. Both multithreading and SIMD enable GPU simulation to run efficiently and accurately on current multi-core processors.

A. Simulating many-core with multi-core

CUDA programming model is designed to reduce as much as possible coupling across multiprocessors. The scheduling order of blocks is not specified, global synchronization is not allowed, communications between blocks are restricted and relaxed requirements on memory consistency enable efficient and scalable hardware implementations. We use these features to simulate each multiprocessor in a different thread of the host.

Our tests suggest that the block scheduler of CUDA 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. Our approach complies with the static scheduling of CUDA and it removes the need to perform a global synchronization after each scheduling round. At warp level, the fine-grained multithreading is simulated as described in Section III-D.

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. This can be challenging to implement and validate and can impact performance. Fortunately, CUDA

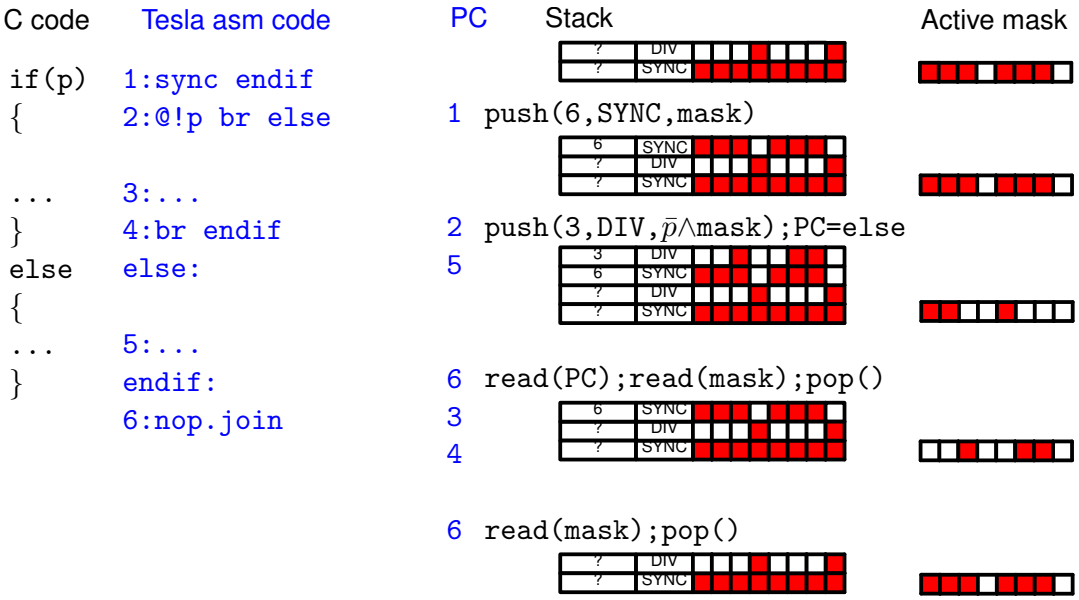

Fig. 4. Example of SIMD forward branch.

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

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

B. Simulating SIMD with SIMD

The Tesla architecture uses 32-way SIMD instructions to execute regular code efficiently. It helps amortize the cost of instruction fetching, decoding and scheduling. It also helps simulation as the part of time dedicated to the actual execution of instructions increases with the complexity of the architecture.

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 Denormals-Are-Zero and Flush-To-Zero SSE flags are enabled to reflect the behavior of the GPU operators and to prevent denormals from slowing down the simulation. The implementation of floating-point instructions, including min and max functions, complies with the behavior of GPUs concerning NaN propagation as long as all input NaNs are encoded as canonical QNaNs.

V. Validation

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

Most of these examples use a data-set reduced for 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 I give on Barra the same results than executions on GPUs, except for the ones that use transcendentals instructions, as it was expected given the difference in implementation. CUDA emulation mode is less accurate. 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.

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 compared 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[4]) only. 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

[4]A texture processing cluster is a hardware structure containing two to three multiprocessors sharing memory access units.

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 I
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).

instruction, branch, branch divergence and memory transaction count in Figure 5.

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 many times on one single block. The profiler seems to select a different TPC to instrument at each kernel call in the round-robin to mitigate the effect of such load imbalance. However, the load imbalance effect remains and affects the counters as the number of calls (202) is not multiple of the number of TPCs (8).

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 the *matrixMul* one, 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 as *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 [26]. We simulated and profiled the *transposeNew* example, which implements the same algorithm while avoiding partition camping and obtained consistent results, which confirms that the observed discrepancy is caused by this phenomenon. We are currently investigating whether the difference in memory transaction count is due to sampling artifacts or actually reflects some hardware mechanism.

As it was discussed in Section III-B, 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.

VI. 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, 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.

We observe that even when run on one core, Barra is competitive with the CUDA emulation mode in terms of speed though it is 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

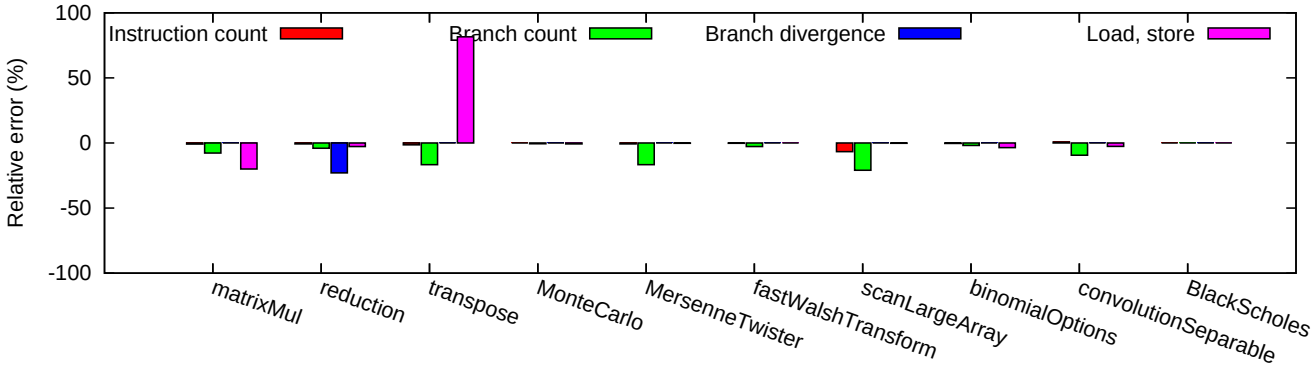

Fig. 5. Relative difference between Barra statistics and GPU hardware counters.

Program	Instruction count	Branches	Divergence	Load, store
Prec	0	0	0	0
matrixMul	-0.83	-7.69	0	-20
reduction	-0.77	-3.99	-22.99	-2.7
transpose	-1.44	-16.67	0	81.58
transposeNew	0	4.93	0	45.59
MonteCarlo	0.01	-0.57	0	-0.68
MersenneTwister	-0.77	-16.67	0	-0.19
fastWalshTransform	-0.24	-2.71	0	0
scanLargeArray	-6.63	-20.93	-0.02	-0.19
binomialOptions	-0.33	-1.85	0	-3.55
convolutionSeparable	0.66	-9.43	0	-2.56
BlackScholes	-0.05	-0.04	0	0.02

TABLE II
NUMERICAL VALUES FROM FIGURE 5

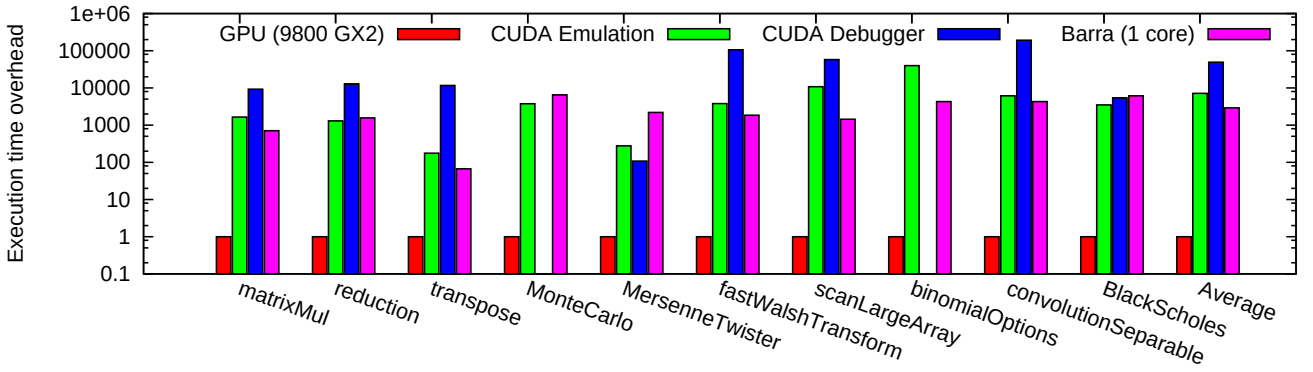

Fig. 6. Compared execution time of native execution, source-level emulation by the CUDA emulation mode, run inside the CUDA debugger and functional simulation with Barra, normalized by native execution time.

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.

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 II-B 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 I.

VII. CONCLUSION AND FUTURE WORK

We described the Barra driver and simulator, and showed that it is possible to simulate the execution of CUDA programs at the functional level despite the unavailability of the descrip-

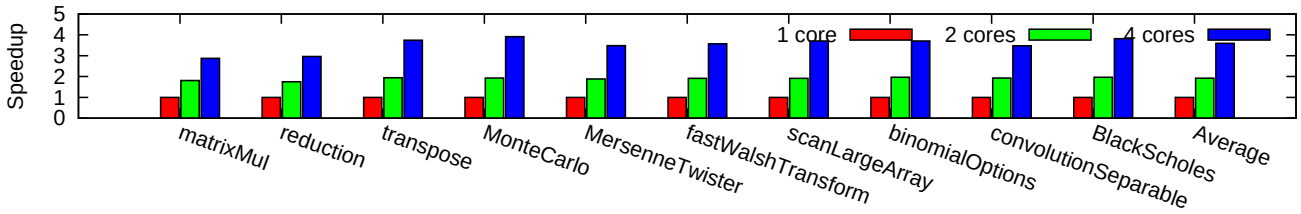

Fig. 7. Impact of the number of cores on parallel simulation speed.

tion of the ISA used by NVIDIA GPUs. 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 is possible to test the scalability of programs without the need to physically test them on various configurations. Our work also enables a deeper understanding of GPU and many-core architecture through extensive analysis of the state-of-the-art NVIDIA Tesla architecture [7], [8], [9].

Barra is distributed under BSD license, available for download[5] 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 [18] 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 [27] and simulation parallelization techniques such as parallel discrete event simulation [12]. The availability of a more accurate timing model opens doors for the integration of other models such as power consumption [8].

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.

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

[7] C. Collange, M. Daumas, D. Defour, and D. Parello. Comparaison d'algorithmes de branchements pour le simulateur de processeur graphique barra. In *13ème Symposium sur les Architectures Nouvelles de Machines*, pages 1–12, 2009.

[8] Caroline Collange, David Defour, and Arnaud Tisserand. Power consuption of GPUs from a software perspective. In *9th International Conference on Computational Science*, pages 922–931, 2009.

[9] 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 (HPPC)*, number hal-00396719, pages 1–10, June 2009.

[10] 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.

[11] 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.

[12] Richard M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] NVIDIA. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 2.0*, 2008.

[20] NVIDIA. *CUDA-GDB : The NVIDIA CUDA Debugger, Version 2.2*, 2009.

[21] NVIDIA. Nvidia's next generation cuda compute architecture: Fermi, 2009.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] Greg Ruetsch and Paulius Micikevicius. *Optimizing Matrix Transpose in CUDA*. NVIDIA CUDA SDK Application Note, 2009.

[27] 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.

[28] 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.

[29] Wladimir J. van der Laan. Decuda and Cudasm, the cubin utilities package, 2008. http://www.cs.rug.nl/˜wladimir/decuda.

[30] 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.