hal-00359342, version 2 - 9 Apr 2009

Barra, a Modular Functional GPU Simulator for GPGPU [*]

Sylvain Collange
ELIAUS, Univ. de Perpignan
52 Av. Paul Alduy
66860 Perpignan, France
sylvain.collange@univ-perp.fr

David Defour
ELIAUS, Univ. de Perpignan
52 Av. Paul Alduy
66860 Perpignan, France
david.defour@univ-perp.fr

David Parello
ELIAUS, Univ. de Perpignan
52 Av. Paul Alduy
66860 Perpignan, France
david.parello@univ-perp.fr

ABSTRACT

The use of GPUs for general-purpose applications promises huge performance returns for a small investment. However the internal design of such processors is undocumented and many details are unknown, preventing developers to optimize their code for these architectures. One solution is to use functional simulation to determine program behavior and gather statistics when counters are missing or unavailable. In this article we present a GPU functional simulator targeting GPGPU based on the UNISIM framework which takes a NVIDIA *cubin* file as input.

1. INTRODUCTION

As Graphics Processing Units (GPUs) gained in flexibility through high-level languages such as CUDA, GPUs gained interest for the acceleration of tasks usually performed by a CPU 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. These new features are the results 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 enormous manpower required in terms of development and validation.

The complexity and performance of modern GPUs presents significant challenges for researchers interested in exploring architectural innovations, 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. Numerous solutions were proposed to simulate CPUs or the graphics pipeline. However, to our knowledge, no simulator taking GPGPU specificities into account is readily available.

We present a modular simulation framework based on UNISIM to perform functional simulation of a GPU targeting GPGPU named *Barra*. We chose to build our simulator on top of UNISIM [2] to benefit of the modularity of this simulation environment and access to the modules already developed (memories, functional units, decoder generator...) 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 simulator driver which loads the input programs, perform management tasks and emulate the graphics/GPGPU driver. We chose the NVIDIA Instruction-Set Architecture (ISA) due to the wide acceptance of the CUDA language in the field of GPGPU. However, the unavailability of the ISA documentation makes our work even more challenging. The simulator takes as input CUDA binary kernels that are normally executed by a NVIDIA GPU and simulates the execution through functional stages. Even though NVIDIA provides various tools such as an emulation mode, a profiler and a debugger to help developers tune their applications, the main strengths of Barra over these proprietary tools lie in its flexibility, geared toward open source and a detailed view on what could possibly happen in a NVIDIA GPU. Barra allows the user to monitor activities of computational units, communication links, registers and memories.

A survey of simulation is given in Section 2. The NVIDIA CUDA framework is described in Section 3. A general view of the proposed framework and features of our simulator and driver are presented in Section 4. Validation and performance comparison are respectively given in sections 5 and 6.

2. 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 targeted. Cycle-level simulators are cycle accurate models characterized by a high accuracy on performance evaluation compared to the real hardware. Transaction-level simulators are mostly based on functional

[*]This work is part of the BioWic project sponsored by the French ANR and is supported by hardware donations from NVIDIA.

hal-00359342, version 2 - 9 Apr 2009

models and focus on 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 unorganised and difficult to modify and other attempts followed. SimpleScalar alternatives were proposed for multicore simulation [13] or full-system simulation [5, 12, 21]. Concerning GPUs, simulation frameworks targeting the graphics pipeline were introduced such as the cycle-level simulator Attila [14] or the transaction-level simulator Qsilver [23].

The Attila Project[1] is based on a full GPU stack emulated in software. This stack is composed of an OpenGL driver, an Attila driver, and a cycle-accurate simulator of the first incarnation of the Attila architecture. Additionally, helper tools to capture open GL traces, play the captured traces and a waveform visualizer for the Attila simulator are proposed.

Qsilver[2] is another simulation framework for graphics architectures that intercepts and processes streams of OpenGL calls with the help of the Chromium system [8]. They extract an instrumented trace which is run in the simulator itself. This simulator may be used to model power consumption or analyze runs of real world applications. The simulator is flexible enough to cover various levels of details so that extra experimental pipeline stages could be modeled.

2.1 Using a modular simulation framework: UNISIM

To improve simulator software development, multiple modular simulation frameworks [2, 3, 20] have been developed during the last decade. The common appealing feature of such environments is the ability to build simulators from software pieces 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 [20] and UNISIM [2].

The UNISIM environment provide GenISSLib: a builder of instruction set libraries. From an instruction-set description, GenISSLib generates a decoder which allows to quickly build an instruction set simulator. The generated decoder is build around 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 a cache look-up of the decoded instruction. Furthermore, the description language allows the user to add functionalities.

[1]http://attila.ac.upc.edu/wiki/index.php/Main_Page
[2]http://qsilver.cs.virginia.edu/

As architecture and software complexity is increasing, simulator performance is becoming 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 [25] suitable for single-thread simulation. The second solution is better suited for multicore and system simulation. It suggests to model architecture at a higher level of abstraction with less details than cycle-level modeling: transaction-level modeling (TLM) [22]. 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 [9]. Finally, recent techniques [19] have been proposed to improve cycle-level simulation of multicore architectures.

3. CUDA ENVIRONMENT

The Compute Unified Device Architecture (CUDA), is a vector oriented computing environment developed by NVIDIA [17]. This environment relies on a stack composed of an architecture, a language, a compiler, a driver and various tools and libraries. In a typical CUDA program, data are first sent from the main memory to the GPU memory, then the CPU sends instructions to the GPU, then the GPU schedules and executes the kernel on the available parallel hardware, and finally results are copied back from the GPU memory to the CPU memory.

A CUDA program requires an architecture composed of a host processor, host memory and a graphics card with an NVIDIA processor supporting CUDA. GPUs supporting CUDA are currently all based on the Tesla architecture. The first processor to support CUDA was the NVIDIA GeForce 8800 and most of its successors from the GeForce, Quadro and the Tesla lines now support it.

CUDA-enabled GPUs from NVIDIA are build around an array of multiprocessors that are able to execute thousand of threads in parallel thanks to many computational units and use of hardware multithreading [10]. Figure 1 describes the hardware organization of such processor. Each multiprocessor contains the logic to load, decode and execute instructions, except for memory instructions which are handled by a dedicated memory pipeline shared between two or three multiprocessors. Each multiprocessor embeds two kinds of vector units which perform an operation at a rate of one Multiply and Add instruction every 2 cycles thanks to 8 SP units, and one special functions instructions every 8 cycles thanks to two SFU units. Such throughput is possible thanks to functional units working at twice the frequency of the vector register file. The register file is comprised of 256 or 512 vector registers of size 32×32-bit. In addition to the register file, each multiprocessor includes a shared memory and constant, texture and instruction caches.

The hardware organization is tightly coupled with the organization of the parallelism defined at the programming level in 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 *kernel*. CUDA let the programmer defines if a variable resides in the GPU address space and the type of parallelism for the kernel execution in terms of *grid*, *block* and *threads*. As the underlying hardware is a vector proces-

hal-00359342, version 2 - 9 Apr 2009

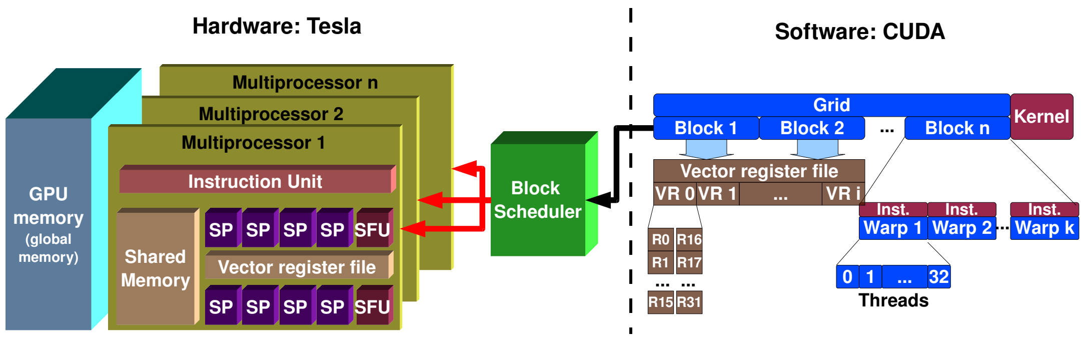

Figure 1: Processing flow of a CUDA program.

sor, threads are grouped together in a so called *warps* which operate on vector registers. Therefore the warp size is 32. At each cycle an instruction is executed on a warp by a multiprocessor. As some instructions have high latency, latency can be covered by scheduling several warps on a multiprocessor, each warp executing instructions at its own pace. When kernels are not using shared memory, the number of warps that can be handled by a multiprocessor is equal to the size of the vector register file divided by the number of registers required to execute the kernel. As a GPU includes several multiprocessors, warps are grouped in blocks. Blocks are scheduled on the available multiprocessors. A multiprocessor can handle several blocks simultaneously if enough hardware resources (registers and shared memory) are available.

NVIDIA states that programs developed for the GeForce 8 series will also work without modifications on all future NVIDIA video cards, thanks to a binary-compatible intermediate language (PTX). Through high-level CUDA, NVIDIA gives developers access to the native instruction set, memories and the parallel computational elements in CUDA GPUs. The format of the native instruction set, which we will call *Tesla ISA*, is tightly coupled with the underlying hardware.

The design flow of a normal CUDA program is a three-step process directed by the CUDA compiler *nvcc* [**?**]. First, thanks to specific CUDA directives from the high level CUDA runtime API, the program is split in a host program and a device program. The host program is then compiled using the gcc/g++ compiler in a Linux environment or Microsoft Visual C++ on a pure Windows platform and the device code is compiled through a modified PathScale Open64 C compiler for execution on the GPU. The resulting device code is a binary CUDA file (*cubin* file) that includes program and data to be executed on a specific GPU. The class of NVIDIA GPU architectures for which the CUDA input files must be compiled can be changed by an nvcc option (*gencode* option). The host program and the device program are linked together thanks to the CUDA library which includes necessary functions to load a cubin file and send it to the GPU for execution. The CUDA runtime API gives the developer the ability to manage devices, memory, synchronization objects, textures and interoperability with other languages. This API is based on the Driver API which offers the same functionalities with more control.

CUDA framework includes proprietary tools such as an emulation mode which compile a CUDA program for an entire execution on the CPU. This mode is very useful for debugging by making possible system calls such as *printf* in a kernel. However it lack of accuracy compared to a real execution on a GPU (ex: floating-point behavior, threads and warps execution/synchronization). For an accurate simulation with debugging information, NVIDIA includes since the 2.0 release of CUDA a debugger. In this case, information provided by the debugger is limited [16].

4. BARRA SIMULATION FRAMEWORK

Our GPU simulator Barra is based on the UNISIM framework described in Section 2.1. It is distributed under BSD licence available for download [3] and is part of UNISIM framework located in the `/unisim/devel/unisim_simulators/cxx/tesla` path.

It is based on a user-friendly environment so that minimal modifications are required to simulate the execution of an NVIDIA cubin program at a functional level, compared to the development process of a CUDA program targeting a NVIDIA GPU. All of the instructions of the Tesla ISA necessary to fully simulate the execution of the examples of the CUDA SDK listed in section 5 are supported.

4.1 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. To achieve this goal, an inspection of the CUDA framework shows that a driver consisting in a shared library that has the same name of the proprietary one *libcuda.so* can captures calls destined to the GPU and reroute them on the simulator. The choice for the user to execute a program either on the GPU or on Barra depends of a simple modification of 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 our simulator. Among all the the available functions the main ones are:

- *cuInit* which sets up a valid environment. It initial-

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

hal-00359342, version 2 - 9 Apr 2009

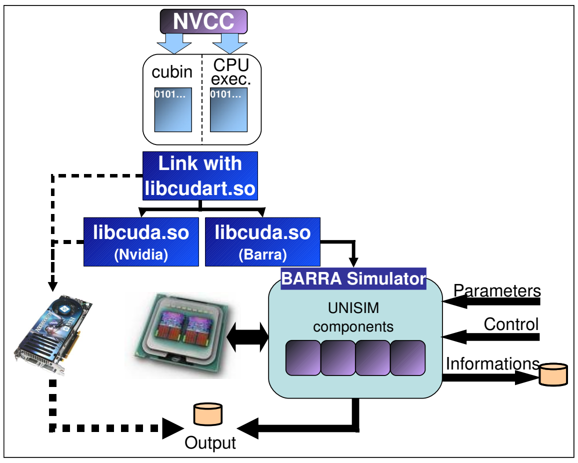

Figure 2: Barra simulation framework.

izes the UNISIM environment to be able to use the Barra simulator and allocate necessary resources such as memory.

- *cuDeviceGet* which returns a reference to a simulated GPU.
- *cuCtxCreate* which allows the creation of a context, analogous to a process running on the GPU.
- *cuModuleLoad* which reads a cubin file, parses it, extracts data, instructions and functions and places them in a hash table.
- *cuModuleGetFunction* lookup a function from the hash table and returns a pointer on it.
- *cuMemAlloc*/*cuMemcpy* which allocate memory in the simulator environment and exchanges user data with device memory.
- *cuFunctionSetBlockShape* which assigns various variables, and checks that the block size required is compatible with the selected GPU. These parameters are passed via shared memory as it is the case with an actual NVIDIA GPU.
- *cuParam** family which specify parameters to the kernels passed via shared memory.
- *cuLaunch** which loads the kernel in device memory, reset GPU states and sets the execution environment registers.

We observe that the set of functions proposed by Barra Driver allows the capture of communications between the CPU and the GPU involved in a CUDA program. We designed others functions for debugging purposes such as *barFunctionDump* which dumps the assembly code. Developers interested in other data such as statistic usage of functional unit, memory bandwidth necessary to achieve full speed, accuracy required or register contentions and dependencies can develop their own counters within the 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), we map all types of memory at different addresses in a single physical address space. We currently map the virtual address space 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.

The user specifies the type of parallelism by setting the dimensions of the grid and block through *cuFunctionSetBlockShape* and *cuLaunchGrid*. We believe these parameters are only a convention used by the CUDA driver, and have no hardware equivalent. Indeed, grid parameters and block shape are passed to the kernel through static locations in shared memory, while thread ID is written in the General Purpose Register (GPR) 0 before the execution starts. The only information known to the hardware (core simulator) is a pointer to a window in shared memory for each warp. When initializing the multiprocessor, we split the shared memory in as many windows as blocks can run concurrently on the multiprocessor, and then set the shared memory pointer of each warp to the window of the block it belongs to.

Thread blocks are executed concurrently by the array of multiprocessors. Blocks are scheduled on the available multiprocessor such that increasing the number of blocks and/or multiprocessors will minimize the execution time on a real GPU. However Barra is currently a functional simulator and not a cycle accurate simulator such that there is no use of simulating more than one multiprocessor as it will not reflect the reality. In addition, determinism will be lost if more than one multiprocessor is used on a multicore architecture. The CUDA debugger behaves the same way, by sequentially executing blocks when the step-by-step mode is activated.

4.2 Barra and Tesla ISA decoding

The Tesla instruction set was introduced with the unified 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. As a consequence, we choose to simulate the Tesla ISA and not PTX to be closer to a real execution, as most of the compiler optimizations happen during the compilation from PTX to the Tesla ISA.

However, NVIDIA, unlike AMD [1], does not document this ISA. Thanks to the harnessing work done in the decuda project [24], we collected data for an integration in Barra on up to 60 different instructions. They cover most of the instructions commonly used in a CUDA kernel such as integer and floating-point add, mul, mad, mac and mov, shifts, memory scatter and gather, branch instructions. . .

The Tesla architecture includes a 64-bit four-operand instruction set, with some instructions that can be compressed to 32-bit instruction words. Another encoding allows em-

hal-00359342, version 2 - 9 Apr 2009

bedding a 32-bit immediate in a 64-bit instruction word. The compiler ensures 64-bit alignment by pairing short instructions. The decoding stage is made easier by using fields in the lower half of long instructions that are similar with fields of short instructions. Therefore, a RISC-like decoder can handle all instructions without resorting to unaligned fetches or shifting.

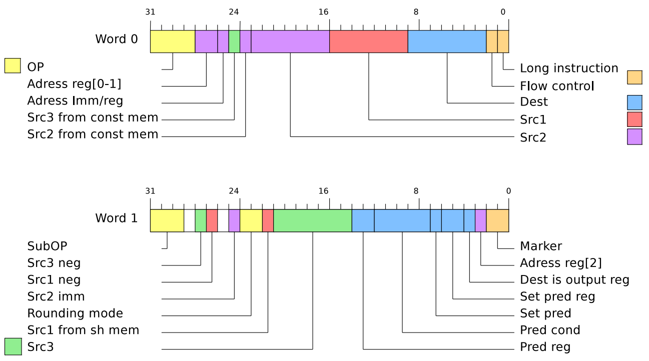

Figure 3: Opcode fields of an FMAD instruction.

An example of the instruction format of a floating-point multiplication-addition instruction in single precision (FMAD) is given in figure 3. The instruction set is divided in up to 32 general instructions (encoded in the OP and Flow control fields), each ALU instruction being dividable further in 16 sub-instructions (SubOP field). These instructions can address up to 3 source operands (indicated by Src1, Src2 and Src3), pointing to general register, shared memory (sh mem), constant memory (const mem) or immediate constant (imm). The destination operand is indicated by Dest. Extra fields such as predicate control, instruction format, marker…are included. Each part is mostly orthogonal to other parts and can be decoded independently. For instance, in a four-operand instruction, the output can be a GPR or a special output register, the first input can come either from a GPR or shared memory, the second input can reference a GPR, a constant memory location or an immediate constant, while the third input is either a GPR or a constant memory location. Most combinations of the former are valid, which renders a monolithic decoder impracticable.

The GenISSLib library included in UNISIM responsible for automatically generating instruction decoders allows the use of several sub-decoders for subfields of CISC type instruction sets. However, taking advantage of this feature requires each group of fields which are related to be contiguous. This is not the case for the Tesla instruction set, where related flags are scattered across the whole instruction words. Instead, we chose 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.

4.3 Instruction execution

Instructions are executed in Barra through a model described in Figure 4. 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 4.2. The decoded instruction is stored in a dedicated buffer to accelerate future accesses to the same instruction. This optimization helps reducing the simulation time. Then operands are read from the register file or from on-chip memories (shared) or caches (constant). As we unify the memory space, a generic gather mechanism is used. Then the instruction is executed and the result is written back to the register file.

Integer instructions can update a flag register containing carry, overflow, zero and negative flags as described in [15]. This behavior is properly simulated in Barra. Verification and validation by experimentation show us that this matches the actual hardware. Barra also supports integer arithmetic in 16-bit on half registers.

Floating-point arithmetic has always been more complex to handle, especially when it comes to GPUs [6]. Barra uses the floating-point arithmetic emulation library proposed with the UNISIM environment. This allows us to exactly mimic the behavior of floating-point units of the targeted hardware such as the truncated multiply-and-add, the flush-to-zero and denormals-as-zero behaviors. In addition, this makes the simulator independent of the host architecture contrary to the emulation mode of CUDA. In the native ISA, all floating-point instructions can update a flag register like integer instructions. Our tests shows that this corresponds to a classification of the instruction result. Although the encoding is peculiar, as positive infinity returns the same flags as normal numbers, we simulate this behavior in Barra as we aim for bug-for-bug compatibility.

Evaluation of transcendentals functions 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 called *Special Function Unit* (SFU), is described in [18] and involves dedicated operators with tabulated values. An exact simulation of this unit will require exhaustive tests on every possible values. Therefore, 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.

4.4 Barra hardware assignation

4.4.1 Register management

GPRs are dynamically split between threads during kernel launch, allowing to trade parallelism for more registers. 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. For functional simulation, warps are scheduled with a round-robin policy.

Streaming multiprocessors of Tesla-based GPUs have a multibank register file partitioned between warps using sophisticated schemes [11]. 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.

hal-00359342, version 2 - 9 Apr 2009

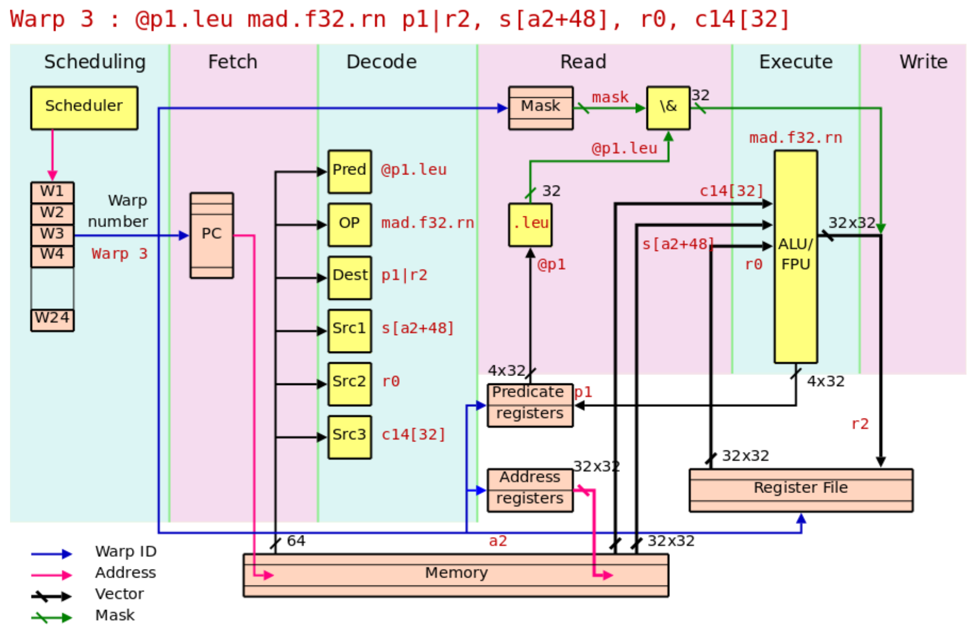

Figure 4: Functional overview of the execution pipeline during the execution of a MAD instruction.

4.4.2 *Thread 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. Even though NVIDIA scheduling policy is not documented explicitly, it is likely that the actual hardware offers more flexibility by overlapping the execution of multiple blocks in a pipelined fashion.

4.4.3 *Branch handling*

Thanks to dedicated hardware, the Tesla architecture allows divergent branches across individual threads in a warp to be executed transparently [7]. This is performed using a hardware-managed stack of tokens containing a 32-bit mask, an address and an ID. The ID allows forward branches, backward branches and function calls to share a single stack.

The branch mechanism used in Barra is described in Figure 5. The mask stack symbolized by rows of squares corresponding to the threads of a given warp is used to handle divergent branches such as in a if-then-else structure. In case a divergent forward branch is encountered, the destination address is put in the forward address stack, then the current mask is pushed, a new mask is set as an inverted condition and the not-taken branch starts its execution. Instructions are executed until the target address is reached or passed by NPC such as in the case of a uniformly taken forward jump. If the target is reached exactly, then the address and mask are popped from the stack and the execution can continue normally. If the target is about to be passed by a forward jump instruction, that is when the target is located between the PC and the NPC, then the target and NPC are swapped and the mask is inverted. A similar technique deals with backward jumps.

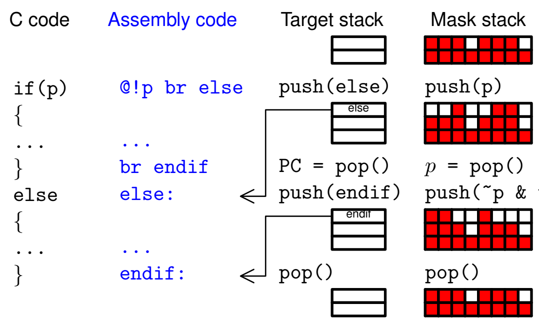

Figure 5: Example of forward branch handling.

5. VALIDATION

We used the examples from the NVIDIA CUDA SDK to compare the execution on our simulator with real executions on Tesla GPUs. We successfully tested the examples

hal-00359342, version 2 - 9 Apr 2009

bitonic	deviceQuery	dwtHaar1D
fastWalshTransform	histogram64	histogram256
matrixMul	matrixMulDrv	MersenneTwister
MonteCarlo	reduction	scan
scanLargeArray	simpleTemplates	transpose

Table 1: CUDA SDK examples which run successfully on Barra

listed in table 1 on Barra. Executions of these examples on Barra give the same results than executions on GPUs, except for those that use transcendentals instructions, as it was expected given the difference in implementation.

As it was discussed in section 4.2, 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 extend and check the instruction coverage.

6. RESULTS

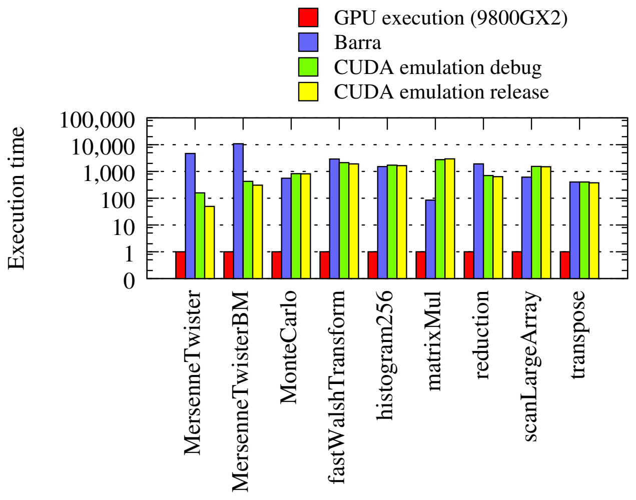

Figure 6: Compared execution time of native execution, simulation and source-level emulation, normalized by native execution time.

The key problem with simulation is the time necessary to simulate the execution of a normal program compared to execution time on a real architecture. We compared and reported in figure 6 the execution time of various programs selected in the CUDA SDK for an emulation (with and without debug information), for a simulation with Barra on a 3.0 Ghz Core 2 Duo E8400 and a native execution on a high-end GPU (9800 GX2). Reported time is normalized to the native execution time for each program. Barra was compiled with `gcc -O2` and the release mode of the CUDA emulation uses `gcc -O3`.

For compute-intensive codes with high workload per thread and little synchronization, CUDA source-level emulation is up to 80 times faster than simulation on Barra when compiled with compiler optimizations enabled. On the other side, simulation with Barra is up to 10 times faster than emulation in debug mode on benchmarks depending on numerous thread synchronizations such as matrixMul. The emulation mode of CUDA may uses thousand of POSIX threads which causes system, memory management and synchronization overheads compared to the execution model of Barra based on a single thread. Compiler optimizations provide little improvement, as the bottleneck lies in the host pthread implementation.

We observe that the simulation time using Barra is similar to the emulation time using CUDA emulation even though Barra is more accurate and provides more flexibility and statistics on the execution. 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 4.3 amortizes instruction decoding cost, providing a significant speed benefit with small kernels operating on large datasets. The extra cost of emulating floating-point arithmetic using integer arithmetic is also not as high as expected, thanks to the existing optimized implementation.

7. CONCLUSION AND FUTURE WORK

In this article we describe the Barra driver and simulator. We show that despite the unavailability of the description of the ISA used by NVIDIA GPUs, it is possible to emulate the execution of an entire CUDA program at the functional level. The development of Barra in 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 a program on various configurations. As a side effect, our work enabled a deeper understanding of GPU and many-core architecture through extensive analysis of the current NVIDIA GPU architecture.

Future work will focus on extending the supported instructions and hardware features covered as well as simulation on multiple multiprocessors. This will go through an integration of transaction-level modeling following the TLM 2.0 standard with extensions for gather/scatter operations or thread scheduling transactions. The availability of a more accurate timing model open 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.

hal-00359342, version 2 - 9 Apr 2009

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] 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.

[6] S. Collange, M. Daumas, and D. Defour. État de l'intégration de la virgule flottante dans les processeurs graphiques. *Revue des sciences et technologies de l'information*, 27/6:719–733, 2008.

[7] 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.

[8] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. *ACM Trans. Graph.*, 21(3):693–702, 2002.

[9] The Open SystemC Initiative. Systemc.

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

[11] John E. 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.

[12] 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.

[13] 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.

[14] 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.

[15] Nvidia. *NVIDIA OpenGL Extension Specifications for the GeForce 8 Series Architecture (G8x)*, February 2007.

[16] Nvidia. *CUDA-GDB : The NVIDIA CUDA Debugger, Version 2.1 Beta*, 2008.

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

[18] 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.

[19] David Parello, Mourad Bouache, and Bernard Goossens. Improving cycle-level modular simulation by vectorization. *Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO'09), Held in conjunction with the 4th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, 2009.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Wladimir J. van der Laan. Decuda and cudasm, the cubin utilities package, 2009. http://www.cs.rug.nl/ wladimir/decuda/.

[25] 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.