Barra, a Modular Functional GPU Simulator for GPGPU

Sylvain Collange, David Defour, David Parello

ELIAUS, Univ. de Perpignan, 52 Av. Paul Alduy, 66860 Perpignan, France
{sylvain.collange,david.defour, 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. Functional simulators are made of several parts. 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.

The complexity and performance of modern GPUs presents significant challenges for researchers interested in exploring architectural innovations, modelling 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. Various academic and industrial works became possible with the availability of such simulators. 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 take advantage of the modularity of this simulation environment and get access to all the module 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 choose Nvidia ISA due to the supremacy of CUDA language in the field of GPGPU. However, the unavailability of the ISA documentation makes our works even more challenging. The simulator takes as input CUDA binary kernels that are normally executed by Nvidia GPUs and simulates the execution through functional stages. This will allow us to study performance metrics such as computation, communication bottlenecks or bandwidth utilization.

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.

2 Simulation

The availability of CPU simulators in the 1990’s for superscalar architectures was the starting point of various 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. Such simulators require a lot of communications between modules composing the simulator, slowing down the simulation. Transaction-level simulators are mostly based on functional models and focus on communications. Transaction-level simulators may be less accurate than their cycle-level counterparts, but will run significantly faster. The fastest simulation is done at functional-level, that mimicks the processor behavior in a simulated environment. Functional-level simulation is not cycle accurate but can integrate power usage and performance model... to quickly estimate performance.

The Cycle-level simulator SimpleScalar [3] was at the origin of various works accompanying the success of superscalar processors in the late 90s. However this simulator was known to be unorganised and difficult to modify and other attempts followed. Concerning GPUs, simulation frameworks targeting the graphics pipeline were introduced such as the cycle-level simulator Attila [9] or the transaction-level simulator Qsilver [12].

The Attila Project1 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.

1 http://attila.ac.upc.edu/wiki/index.php/Main_Page
Qsilver\textsuperscript{2} is another simulation framework for graphics architectures that intercepts and processes streams of OpenGL calls with the help of the Chromium system \cite{6}. 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 detail so that extra experimental pipeline stages could be modelled.

2.1 UNISIM

UNISIM \cite{2} is altogether a modular simulation environment and a module library. It intends to distribute the development effort of simulation tools over multiple research groups. It relies on a combination of modular software development, distributed communication protocols, multilevel abstract modelling, simulators services and an open repository for simulator modules available at www.unisim.org.

Systems are design by assembling multiple hardware blocks, possibly at different levels of detail. SystemC popularized modular multi-granularity simulators. On top of SystemC, UNISIM proposes a standardized architecture-level interface and common communication protocols between these interfaces. Simulators and modules can be integrated in an automatic design space exploration method thanks to the UNISIM simulation engine which can automatically explore all possible architecture configurations, and maintain a permanent ranking of possible architectures, under various performance constraints. Simulators can reuse contributions of academics and engineers uploaded on the common open repository. Some plugging interface and services can be integrated in a simulator based on UNISIM to propose energy, area or cost estimations.

UNISIM groups claim that the modularity and the variety of the library, made of memories, caches and pipeline stage modules, improves simulator development speed, simulator support, and simulation speed. Among such features, UNISIM integrates Genisslib \cite{11}, an instruction set simulator generator that can be used to quickly design support for an instruction level simulator. It allows developers to decode instructions from binary code and associate a behavior to them.

3 CUDA environment

The Compute Unified Device Architecture, also known as CUDA, is a parallel computing environment developed by Nvidia \cite{10}. This environment relies on a stack composed of an architecture, a language, a compiler, a driver and various tools and libraries. The execution of a CUDA program through this stack is described in figure 1. In a typical CUDA program, data are first send from 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.

\footnote{http://qsilver.cs.virginia.edu/}
A CUDA program requires an architecture composed of a host processor, host memory and a graphic card with an Nvidia processor supporting CUDA. GPUs supporting CUDA are mainly graphic processors that implement the unified graphics pipelines proposed by the DirectX 10 standard. 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 processors that are able to execute thousand of threads in parallel thanks to many computational units and hardware multithreading [8]. These units are structured in a two-level hierarchy. The first level is made of clusters of thread processors (TPC). Each cluster embeds several streaming multiprocessors (SMs) and a texture pipeline to access memory. Each SM has its own instruction front-end and is able to load, decode and execute instructions except for memory instructions that are handled by a dedicated memory pipeline shared within a TPC. Each SM is composed of several stream processors (SP) that perform a scalar computation. Every SP among a SM shares the instruction loading and decoding units, working in a SIMD fashion.

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, if a variable is in the GPU address space and the type of parallelism for the kernel execution in terms of grid, block and threads. Nvidia states that programs developed for the GeForce 8 series will also work without modification on all future Nvidia video cards, thanks to a binary-compatible intermediate language. Through high-level CUDA, Nvidia gives developers access to the native instruction set, memories and the parallel computational elements in CUDA GPUs.

The execution of a CUDA programs requires CUDA components that are now included in all Nvidia graphics drivers. This driver includes a dynamic compiler that can retarget executables for the hardware at runtime. The CUDA compiler nvcc can output three kinds of files. A binary CUDA file cubin, an assembly-
level intermediate language PTX file or a C file. The generation of the cubin file requires the use of either the low level API called CUDA driver API or the high level API called CUDA runtime API built on top of the former.

4 Barra simulation framework

Our GPU simulator Barra is based on the UNISIM framework described in Section 2.1. We set up a user-friendly environment so that minimal modifications are required to simulate the execution of an Nvidia cubin program at a functional level on Barra, compared to the development process of a normal CUDA program targeting a Nvidia GPU. However, due to the complexity of the selected ISA, we currently only support a subset of the Nvidia instruction set as well as hardware capabilities.

Fig. 2. Barra simulation framework.

4.1 Barra driver

The design flow of a normal CUDA program is a three-step process directed by the nvcc compiler. 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 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.

For an execution on the simulator, the kernel is compiled as if it is going to
be executed by the GPU but it is linked against the Barra library (driver). The
library captures events and calls that normally interact with the GPU, so that
the program interacts with the simulator instead. The proposed Barra driver
includes major functions of the Driver API so that a basic CUDA program can
be loaded, decoded and executed on our simulator. Available functions are:

- `cuInit` sets up a valid environment. It initializes the UNISIM environment
to be able to use the Barra simulator and allocate necessary resources such
as memory.
- `cuDeviceGet` returns a reference to a simulated GPU.
- `cuCtxCreate` allows the creation of a context, analogous to a process running
on the GPU.
- `cuModuleLoad` 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` allocate memory in the simulator environment and
exchanges user data with device memory.
- `cuFunctionSetBlockShape` 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*` families of functions specify parameters to the kernels passed via
shared memory.
- `cuLaunch` loads the kernel in device memory, reset GPU states and sets the
execution environment registers.
- `cuLaunchGrid` is not yet implemented but will be used to support more than
one multiprocessor.

Therefore, we observe that the set of functions proposed by Barra Driver
allows the captures of communications between the CPU and the GPU involved
in a CUDA program. As a consequence, common CUDA programs can be
linked against Barra driver without any modifications so that the binary code
is executed on Barra simulator. We designed others functions for debugging
purposes such as `barFunctionDump` that dumps the assembly code. An example
of a simple example CUDA program using barra’s functions in given in Listings
1.1. The corresponding execution trace is given in Listings 1.2.

Listing 1.1. CUDA program using Barra functions.
```c
// test.c
extern "C" global void my_kernle(float * data, float p0) {
    float ret = p0 + 4.0f;
    int index = threadIdx.x;
    data[index] = ret;
}

Listing 1.2. Execution trace of the source code 1.1.
```

```c
#include <cuda.h>

int main(int argc, char * argv[]) {
    int numthreads = 40;
    CUmodule m = 0;
    CUdevice dev = 0;
    CUcontext ctx = 0;
    CUfunction hfunc;
    cuInit(0);
    cuDeviceGet(&dev, devid);
    cuCtxCreate(&ctx, 0, dev);
    cuModuleLoad(&m, "test.cubin");
    cuModuleGetFunction(&hfunc, m, "my_kernle");
    float * data = 0;
    int size = numthreads * sizeof(float);
    CUdeviceptr gdata = 0;
    // Allocate GPU and CPU memory
    cuMemAlloc(&gdata, size);
    cuMemsetD32(gdata, 0, size);
    data = (float *) malloc(size);
    cuFuncSetBlockShape(hfunc, numthreads, 1, 1);
    // Set execution environment
    cuFuncSetSharedSize(hfunc, 0);
    cuParamSetSize(hfunc, 8);
    cuParamSetv(hfunc, 0, &gdata, 4);
    cuParamSetf(hfunc, 4, 1.0f);
    barFunctionDump(hfunc); // Debug function for BARRA
    cuLaunch(hfunc); // Execute the kernel
    cuMemcpyDtoH((void *) data, gdata, size);
    // Get data back
    for (int x = 0; x < numthreads; ++x) printf("%f ", data[x]);
    printf("\n");
    free(data);
    return 0;
}
```
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 believe the hardware behaves the same way, in order to allow DMA transfers from main memory directly to shared memories. 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 and allowing the performance modelling of TLBs.

Each kernel defined in CUDA is executed \( N \) times in parallel by \( N \) threads. Depending on the available hardware and communication constraints, these threads are grouped together in blocks of threads. There is a hardware limit on the number of threads included in a single block (e.g. 512 on the Tesla architecture), however a kernel can be processed by multiple equally-shaped thread blocks. Thread blocks are executed concurrently by SM. The user specifies the dimensions of the grid and block through \texttt{cuFunctionSetBlockShape} and \texttt{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 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. We haven’t implemented a valid block scheduling strategy in Barra which means that only one SM is supported.

### 4.2 Barra and Tesla ISA decoding

Nvidia G80 and GT200 processors share the same instruction set architecture, which we will call the Tesla ISA. One can notice that PTX intermediate language...
is different from the Tesla ISA which represent two different concepts. However, Nvidia, unlike AMD [1], is not revealing this ISA. Thanks to the harnessing work done in the decuda project [13], we collected informations on up to 60 different instructions covering most of the instructions commonly used in a CUDA kernel.

The Tesla architecture accepts a 64-bit four-operand instruction set, with some instructions that can be compressed to 32-bit instruction words. The information related to the instruction length is a one bit field. 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.

![Fig. 3. Opcode fields of an FMAD instruction.](image)

An example of the instruction format of a floating-point multiplication-addition instruction in single precision (FMAD) is given in figure 3. 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. All combinations 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 sub-fields 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.

We implemented in Barra the parsing of all 64-bit encodings and 15 of the most frequently used instructions including the integer and floating-point add, mul, mad, mac and mov, shifts, scatter, gather and branch instructions. Adding other instructions only requires extending the opcode sub-decoder, most of the decoding works being already performed by the other decoders.

4.3 Barra hardware assignation

GPRs are dynamically split between threads during kernel launch, allowing to trade parallelism for more registers. We maintain 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 windows to the assigned register set, and a windows to the shared memory. For functional simulation, warps are scheduled with a round-robin policy.

A specific marker embedded in the instruction word marks the end of the kernel. When it is encountered, the current warp is flagged as inactive and is ignored by the scheduler in subsequent scheduling rounds. In the current implementation, execution is stopped when every warp of a given block have reached the inactive state. In future versions of Barra, we will implement block scheduling which will reset simulator states and assign a new block of warps whenever the previous situation will be reached.

Thanks to dedicated hardware, Tesla ISA offers transparent branch handling working in a SIMD fashion [4,5]. However the detailed hardware implementation is largely unknown. We simulate the Nvidia branch handling using an if address stack, a loop address stack and a mask stack. Our implementation is inspired from other designs described in [1,7].

In terms of memory the core simulator required 5.8 MB for the execution of the example given in appendix A. The simulator core represent 7000 lines of C++ and ISA description code while the driver represent 2800 lines of code in addition to the UNISIM environment. The simulation process is 200 to 800 slower than a real execution on a GPU. The simulator is now part of the UNISIM project, available for download and located in the /unisim/devel/unisim_simulators/cxx/tesla path.
5 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.

Future work will focus on extending supported instructions and hardware features as well as simulation on multiple multiprocessors. We will also work on cycle level simulation of the execution pipeline and memory subsystem.

References
