

Archaeopteryx

A Multi-BSP Processor Simulator

Sudnya Damos and Gregory Damos

No Affiliation

mailsudnya@gmail.com, gregory.damos@gatech.edu

I. INTRODUCTION

The transition to many core computing has coincided with the growth of data parallel computation and the evolution of graphics processing units (GPUs) from special purpose devices to programmable cores. The emergence of low cost programmable GPU computing substrates from NVIDIA, Intel, AMD, and ARM have made data parallel architectures commodity from embedded systems through large scale clusters such as the Tsubame [?] and Keeneland systems [?] hosting thousands of GPU chips.

The dominant programming systems involve the use of bulk-synchronous-parallel programming models [?] embodied by languages such as CUDA, OpenCL, and C++-AMP. These data-parallel languages implement *single instruction stream multiple thread* (SIMT) models of computation that specify a large number of data-parallel threads that can be readily exploited by hardware multi-threading and *single-instruction-multiple-data* (SIMD) cores.

In contrast to many-core processors based on out-of-order CPU cores, GPU architectures are designed to exploit the massive data-parallelism of bulk-synchronous programs. Performance is maximized for regular computations where hardware can use SIMD pipelines and bulk data transfers to exploit control and data locality among threads. However, current designs suffer from steep performance cliffs when executing programs with irregular control flow and data access patterns. They are also beginning to introduce new programming hazards as they move to *non-uniform memory access* (NUMA) organizations to reduce on-chip network overheads.

These performance hazards limit the potential of GPU programming. In fact, sequential algorithms mapped to single CPU cores are still competitive with GPUs for many application domains (e.g. database queries, loss-less compression, lexing/parsing, or graph analytics) despite a 300x (and exponentially growing) difference in peak throughput. Two of the most important problems in computing today involve designing architectures with more gradual performance cliffs (without sacrificing parallel efficiency), and designing bulk-synchronous al-

gorithms for irregular applications.

Unfortunately, the tools that have been traditionally used to explore solutions to these problems (architecture simulators and analytical models) fall into the category of applications *without* efficient parallel implementations. This has created a gap between parallel processor and simulator performance that is widening with each successive generation, limiting the ability of architects to explore the design space of possible optimizations and the ability of application designers to evaluate the impact of algorithms on future architectures.

This is even more troubling for functional verification. Although most bugs are still caught with relatively short directed tests [?], a critical class of bugs only occurs during complex interactions between multiple components. These bugs are typically discovered through application stress testing or random program execution. As application complexity continues to grow with processor capability, there is a rising concern that simulator performance will limit application and random test coverage, especially for applications with multiple phases.

It would be desirable to leverage the tremendous growth in computing potential provided by parallel processors for microarchitecture simulation. However, the sequential operation and tight dependency loops in high performance processor pipelines quickly dash the hopes of straightforward solutions. Exploiting fine-grained data parallelism and expressing the operation of a processor in terms of hierarchical bulk data transfers requires a ground-up approach.

In this paper, we assert that the parallel and hierarchical organization of microarchitecture structures used in modern processors provide a natural basis for simulation using data-parallel algorithms. We test this assertion by implementing a composable functional simulator for a new processor architecture, and then augment it with performance models for key micro-architecture structures. We exploit fine-grained data parallelism by using hundreds of threads to simulate a single core. We show how basic parallel algorithm building blocks like reductions, prefix sums, sorts, and histograms can be used to simulate instruction fetch units, register files, thread schedulers, functional units, and caches. Across X

applications, we show that the accuracy of the simulation is comparable to state-of-the-art sequential simulators, and that the performance of the simulator scales across three generations of GPUs. This paper describes the first data-parallel processor simulator that can leverage the technology scaling and microarchitecture enhancements of the previous generation of GPU hardware to simulate the next.

Specifically, this paper makes the following contributions:

- It describes the design of the first processor simulator implemented completely in the CUDA "C" language.
- It introduces a design methodology for simulating parallel processors that achieves near peak utilization of modern GPU accelerators and good scaling across three generations of GPUs.
- It shows that microarchitecture details, such as scratchpad sizes, hardware thread schedulers, register file organizations, cache hierarchies, and core pipelines can be simulated quickly as long as the hierarchical organization of the machine is maintained.
- It compares the results and performance of our parallel simulator with two existing GPU simulators: GPGPU-Sim, and the Ocelot PTX Emulator. We demonstrate a N_x speedup.

A. Outline

This document is organized as follows. Section II briefly describes the processor being simulated. Section III walks through the execution of sample application on the proposed simulator. Section IV provides a high level overview of the simulator framework. Section V drills down into the implementation details of the simulator. This is followed by Section VI-C0j, which provides an evaluation of the speed and accuracy of the simulator. Section VII surveys related work, and Section VIII concludes the paper.

II. VANAHEIMR

Vanaheimr is an architecture that is optimized to execute Multi-BSP programs that express a high degree of parallelism hierarchically. The memory system is composed of memory controllers that are tiled across the processor die and connected to 3D DRAM modules either directly through vias, or indirectly through a silicon interposer. The cache is organized into an H-tree structure, with clusters at each level of the hierarchy. Each level is connected by a stage of an indirect network. The cache bank size, number of levels, as well as degree of connectivity at each level is configurable.

The cores themselves implement a hierarchical SIMD architecture. The main difference between this and a

traditional SIMD processor is that the unit of control that is broadcast to multiple execution units is larger than a single instruction; it is referred to as a sub-kernel. Cores may be heterogeneous at the micro-architecture level to balance sequential against parallel efficiency. However, they all implement the same ISA and execution model. The architecture is intended to scale across regular and irregular applications by varying the ratio of execution units to memory, and the number of levels in the inter-connection network.

This paper is concerned with the design of a functional simulator for the Vanaheimr architecture, so only the essential aspects are mentioned here. Additional details about the architecture are described in related work [?].

A. Architecture

Vanaheimr targets 100 TFLOPs peak throughput at a 5nm technology node on a 300mm² die, while peak single-threaded CPU performance is currently limited to less than 15 GFLOPs [?]. In order to bridge the gap between simulator and hardware performance, the Archaeopteryx simulator is implemented using Multi-BSP algorithms that are intended to execute on a commodity parallel processor, such as NVIDIA Kepler with a peak throughput of 2.5 TFLOPs (GK104) or greater.

Although some components, namely the SIMD lanes, used in the Vanaheimr architecture map naturally onto the CUDA model, the memory system, network, and core control logic exhibit either serial operation in hardware or many fine-grained dependences. These components are modeled individually by leveraging bulk-parallelism where it exists (such as among network buffers and lanes), and falling back to serial execution where it does not (such as the SIMD instruction fetch to broadcast loop). Individual components are composed together hierarchically into levels in the final Multi-BSP simulation that expand or contract the amount of parallelism during execution.

B. Programming Model

Vanaheimr programs are compiled into an intermediate form, VIR (the Vanaheimr Internal Representation), from a high level language such as CUDA with nested parallelism, OpenCL, Phalanx, Sequoia, or any other that can be mapped onto the Multi-BSP model. VIR is a RISC-like instruction set that shares many opcodes and the type system with LLVM. However, it also defines a flexible mechanism for declaring scratchpad memory that is shared at different levels in a hierarchical thread array (HTA).

During compilation, a program is first lowered into VIR, where device agnostic optimizations are applied. Then, a device-specific backend transforms the code to conform to the machine ABI for function calls and

variable layout in memory. It also translates VIR opcodes into machine instructions and performs scheduling and resource allocation of memory, registers, and code. A final binary is produced that is suitable for execution in a SPMD fashion.

Programs begin with a single thread entering a *main* function. This thread may perform scalar operations, or make system calls to allocate memory or interact with IO devices; it may also create additional threads by launching a HTA. HTAs can be launched synchronously or asynchronously. Synchronous launches suspend the calling thread until the HTA returns, while asynchronous launches create a continuation that launches the HTA after the current thread finishes. Parallelism can be specified statically via the launch of a single multi-level HTA, or it can be created dynamically, via the recursive launch of many HTAs with fewer levels. This allows regular applications to avoid the overheads of runtime work creation, and irregular applications to adjust the amount of parallelism to match the shape of data-dependent problems.

Memory allocation is performed statically with automatic variables, or dynamically via a malloc equivalent. The main difference between these concepts and their sequential counterparts is that they also specify the level in the HTA at which they are replicated. For example, a statically sized array of integers could be allocated per-thread, once for the entire HTA, or replicated at any other level. Similarly, a call to malloc could be made by each thread to allocate memory to store a private data structure, or by a single-thread to allocate memory that is shared across threads at the same level. The exact mechanism used to specify the scope of a variable is left to the programming language designer. This mechanism is meant to communicate this information to the runtime or compiler, which can control the placement of these memory allocations at the appropriate level in the memory hierarchy.

Synchronization can be performed explicitly via barriers at any level of a HTA, or implicitly through the bulk-synchronization mechanism as HTAs complete. This means that memory that was modified by a synchronously launched HTA will be immediately available to the calling thread when it resumes.

III. EXECUTIVE SUMMARY

This paper begins with a concrete example of the simulation of single application on an instance of the Vanaheimr processor architecture.

A. Application Overview

SAXPY is the canonical example of a data-parallel program from the dense linear algebra application domain. A single multiply-accumulate kernel is applied to



Fig. 1. The SAXPY kernel CUDA "C" source code.



Fig. 2. The main program thread CUDA "C" source code.

each element in a one-dimensional vector. The CUDA C source code for this kernel is shown in Figure 1.

The program begins with a single thread, which allocates the X and Y arrays, then initializes them with random data. It then applies a block-cyclic data-distribution to the X and Y arrays, which partitions them into contiguous blocks that are allocated on the same DRAM module, managed by the same memory controller, and processed by the same core. Once memory has been initialized, the main thread launches the SAXPY kernel synchronously, expanding the amount of parallelism. The dimensions of the input and output data sets are known statically, as are the elements from each set that will be processed by each thread. This simplifies the problems of creating data distributions and HTAs, and they can be created at once when the kernel is launched. More complex applications may require creating additional HTAs or changing data distributions during kernel execution. However, the simple example

of SAXPY is covered first.

Once the kernel returns control transfers back to the main thread and memory is synchronized, the results are verified, memory is freed, and the program exits.

Although SAXPY is a completely data-parallel algorithm with one operation naturally mapping onto a single thread, the kernel is composed of a two-level hierarchy of thread arrays, with each array containing several hundred threads. This is done to reduce the overhead of launching individual threads, and to coarsen the granularity of the input and output data distributions. This coarsening eases the process of mapping thread arrays to cores with good data locality.

The kernel launch specifies the dimensions of the HTA as well as the input parameters to the kernel. A two-level HTA is created. The first level is an array of 1024 threads and the second level is an array of arrays that covers the entire input and output data sets. No scratchpad memory is allocated since this application simply reads an input and writes an output without any reuse or any inter-thread communication. The kernel parameters (pointers to the X, Y arrays and the value of A) are broadcast to each thread; they are initialized before the thread is created. After it has been created, each thread determines its id, finds the offset of the elements in X and Y that it should process, performs the $Y = A * X + Y$ calculation, and exits.

B. Compilation

In order to execute Vanaheimr programs such as SAXPY on the Archeopteryx simulator, they must be compiled into the VIR form from source.

Vanaheimr's programming model extends the baseline defined by CUDA "C" in several ways, for example, by allowing device threads to call CUDA API routines, and by extending the two-level CUDA kernel to an HTA with unbounded levels.

The mapping from CUDA "C" into VIR is performed in a straightforward manner. NVIDIA's CUDA compiler is used as a frontend to lower the C-like language into a virtual assembly language, PTX. A modified version of the Ocelot compiler is then used transform PTX into VIR. During this process, several PTX intrinsics for launching HTAs, declaring statically allocated variables with specific HTA-level scoping, and dynamically allocating scratch-pad memory are recognized as PTX intrinsic functions and converted into VIR abstractions. These abstractions can be analyzed and optimized by the VIR optimizing compiler, which is also written in CUDA.

Kernel parameters are declared explicitly in VIR in kernel definitions. For example, the VIR representation of the SAXPY kernel in Figure ?? includes the variables X, Y, and A in the kernel prototype. These parameters

are accessed via explicit load instructions in the kernel body. In a real implementation, these parameter accesses would be either converted into reads from pre-initialized registers or loads from memory by a backend compiler for Vanaheimr. However, since this form of the program is simulated directly, they are supported by the simulator and complete in a single instruction.

As simulated HTAs are launched, they can specify the launch of any level in the hierarchy to be replicated for each partition of a data distribution. This allows the work distribution mechanism to schedule thread arrays on SMs that are physically closer to the associated data partition. In this example, the partitions of the X and Y arrays are striped across the memory controllers in Vanaheimr, and the corresponding thread arrays are scheduled on the nearest SM.

C. Execution

Once the program has been compiled into a binary, it is ready for execution. The Archeopteryx simulator is a stand-alone CUDA program that takes a compiled VIR binary as an input. The simulator begins by starting up the host CPU threads that implement the extensions to the CUDA API, and launching a single CUDA kernel with a single thread. This thread uses the CUDA API extensions to open the VIR binary and read the symbol table. It determines the memory requirements and HTA configuration of the binary's "main" function, sets up the simulator state, points the work distributor at a descriptor of the first simulated HTA, and finally enters the main simulator loop.

The simulator loop proceeds in a conservative time-step fashion over four main components, the SMs, the network, and the memory controllers, and the work distributor. A single CUDA kernel is launched by the main simulator thread for each component. The SM component launches a thread array for each SM in the simulated machine, which begins executing threads until they block on memory events that have not yet been simulated. The network model launches one thread array for each router in the network, which propagates flits for the current time-step. The memory model launches one thread array for each memory controller, which arbitrates among current requests and generates responses. Finally, the work distributor tracks simulated threads that are currently executing along with those that are waiting for memory or SM contexts to become available. When a new thread array is launched on an SM, it allocates memory for data structures associated with those threads, such as stack memory, and initializes registers and memory, such as the thread id or stack pointer, conforming to the Vanaheimr ABI.

Once set in motion, this process continues until all threads in a simulated HTA have completed and no

pending HTAs have been launched. At this time, the simulation is complete.

For the SAXPY program, the binary's "main" starts with a single thread. This is scheduled on an SM, which executes instruction by instruction up to the point where it launches the main SAXPY kernel. This launch is performed by compiler-generated code to create a HTA descriptor and push it into a queue owned by the work-distributor. Since the call is synchronous, the thread saves its live state (registers, stack) and exits. The return point is indicated in the HTA descriptor. Eventually the work-distributor will read from the launch queue and begin launching thread arrays on idle SMs. These threads will execute the SAXPY kernel and eventually return. Once all threads have returned, the work-distributor starts up the main thread again at the instruction after the HTA launch, and it continues to completion. At this point, the simulation is finished.

IV. FRAMEWORK DESIGN

A. Simulator Structure

As a consequence, only the kernels from CUDA applications can be simulated.

B. Asynchronous Multi-BSP Kernel Launches

a) *Program Representation:* Intrinsic functions are lowered as late as possible, below the VIR level, allowing a backend compiler to map them directly onto hardware acceleration if available. For simplicity, the Archaeopteryx core model executes one machine instruction for each virtual instruction as a baseline.

C. File I/O

Archeopteryx supports file I/O operations or other system calls from any thread in the simulated program, including the main thread, but also from individual threads in a HTA. It also makes use of this functionality within the implementation of the simulator components. Currently, this functionality is primarily used for reading input files, accessing the simulated program's binary, and reporting messages to the simulator console such as aggregate statistics or error messages.

This functionality is currently implemented with message passing between CUDA threads and a supporting runtime executing on the host CPU. The NVIDIA CUDA runtime is augmented with additional functionality for receiving messages through shared GPU-CPU memory queues, decoding them, performing I/O operations or system calls, and replying back to threads. This is done due to a lack of device drivers or other OS-level functionality that is available on current GPUs to CUDA applications. If future GPUs support system calls directly, an updated implementation of Archaeopteryx could map these operations directly onto system calls and bypass the CPU completely.

V. SIMULATOR CORE DESIGN

(I) Overview of the simulated architecture

Currently the trend in processor architecture development is to incrementally modify and enhance features without completely redesigning something from scratch. This trend helped us leverage the current underlying features to simulate the next architecture and evaluate the performance of similar workloads on them. The crux of design for the simulator core was to follow the SIMT paradigm, the cache hierarchy, the ownership of ALUs as well as the memory management unit. We used the CUDA programming language to write the simulator. We chose this parallel language since it already has the concept of a group of threads (similar to an HTA), hardware lanes where the threads are executed as well as a single execution entity that is a thread. We leveraged CUDA to map our new architecture's concepts onto existing hardware.

The simulator models the Streaming Multiprocessor of the target architecture. This entails the part of the core where actual instructions are executed for many hundreds of threads.

At the start of the simulation, an initial state is set up by a runtime which determines the various architectural configurations like the total number of simulated threads, the number of threads in each HTA, the amount of shared memory for each HTA as well as the local memory per thread. These attributes can be varied by architects to perform experiments and studies to determine the best set of values for architectural components in the next generation.

We describe the flow of an instruction through the simulated pipeline through all stages in order to explain the execution model in detail. The fetch stage occurs when an instruction is loaded from a binary file that stores it in an ABI of the simulated architecture. Currently there is only a single binary loaded, but it is easy to expand the runtime to handle multiple binaries in a given simulation instance. The ABI format is described in the earlier section. Once fetched, there is a lookup into an array of all the opcodes for the architecture we are simulating. When a match is found, the execute stage for that opcode occurs. In the execute stage, the operands are decoded depending on the instruction opcode. After decoding the operands, the actual operation corresponding to the instruction is performed on the decoded operands and the result is stored in the destination. The program counter is then advanced to the next instruction.

The main goal for writing parallel programs that run on GPUs is to maximize throughput by cleverly scheduling as much work as possible on real hardware. In order to achieve this behavior in our simulator, the work was distributed onto the actual hardware using two levels of scheduling. The first level scheduler mapped a

group of threads onto a group that actual hardware could execute simultaneously. The scheduling policy used for it was a straightforward round robin algorithm. We chose round robin policy due to its simplicity and plan to enhance the system by offering a variety of scheduling policies easily replaceable by changing the configuration file. When a given group of threads execute on the hardware, they can diverge on their execution paths. In such a case, they usually synchronize at a reconvergence point that is contained at the post dominator's basic block. A reconvergence point is point at which all threads explicitly pause execution until all other threads in the group reach the same point. The second scheduler implements a priority based scheduling policy which gives highest priority to the thread that is farthest away from such a barrier. At each step a reduction is performed on all the threads in the group to determine the thread with the highest priority. This ensures that the thread that is the most divergent gets priority of executing until it reaches the same point as other threads. Once they all reach the reconvergence point, the expectation is that all the threads will resume executing the same code path in lock step.

Usually in a SIMT pipeline, most of the threads execute the same path. Our simulator leverages this property and caches the access made to the binary during fetch stage. Thus multiple redundant accesses to the disk for reading the binary per threads are avoided.

In parallel architectures, one way to achieve a high throughput is by combining bursts of memory requests generated by multiple threads into one transaction. In order to make this easily possible, the layout of memory allocations by the program has to be strided such that when each thread is accessing the n -th index of a memory block, they are linearly next to the other. Such a phenomenon is called memory coalescing. It has become a prerequisite to obtain optimal performance from any high performance application written in a parallel language. We use coalescing while laying out the register files for each thread in a group. Thus, in an allocated array of memory that is used as register files for all the threads in that block, the n -th register for thread-1 will be next to the n -th register of thread-0. This enables the hardware to combine accesses.

A code with multiple conditions jumps into different basic blocks. These conditions typically cause some threads in a group of threads to take different paths and execute different code hampering the high throughput that SIMT is known for. Such a structure is known as divergent code. It is one of the main factors that programmers should consider while writing parallel programs. A naive way to look up opcode would be a switch statement. This would compile into a series of conditional branches with as many code blocks as the size of the

ISA. To optimize the performance of our simulator we introduced a jump table for opcode, operand look up. This jump table eliminated all the divergent code blocks except one.

The fastest way to access data on a GPU is from a register. So to run the simulation at the maximum memory bandwidth, all the data accesses would have to be from the register file. We should note that register files in hardware cannot be dynamically indexed. Thus mapping registers from the simulated binaries onto hardware register file would not be possible without self modifying code. In this project we concern ourselves with writing a parallel simulator to execute parallel code in the most optimal way. Writing applications with self modifying code is beyond the scope of this project. Additionally, mapping data accesses to register file in hardware would cause the accesses from the simulated binary to compete with the accesses simulator's binary. We should note that while performing architectural studies we would require to simulate a system with much bigger register file size causing starvation in some cases. The next option would be to use shared memory for this data that was being maximally accessed, but that would require additional support to handle spilling/overflow. We chose L1 cache over shared memory since the eviction of an entry from cache and algorithm to leverage locality is implicit in L1 cache. The register file was organized in a way that caused all the threads' access to a certain index register to be on the same cache line, thus allowing memory coalescing. This way, we did not have to bake the operands from the simulated binary onto actual registers on hardware, thus keeping programs flexible and allowing a the simulator once compiled to simulate multiple CUDA binaries one after the other. We should keep in mind that L1 cache was better than shared memory in this case due to the size of registers per thread and nature of these accesses being aligned to a cache line of L1. However the tag line can prove to be a limitation of L1 cache access speed for other data access patterns.

Synchronization across all the threads within a block is implemented using the well known 'barrier' mechanism. The hardware implements barriers across threads that are simultaneously executing on it. Barriers across threads in a block that are scheduled in different groups are implemented using a priority mechanism. Each schedulable group of threads has a barrier bit set to true that is set only when all the threads contained in it have reached the barrier. When the scheduler looks for a group to run, it does not select a group with the barrier bit set to false since that indicates there are some threads in it that haven't yet reached the barrier and synchronized to run in lock step.

Our simulator models shared memory by mapping simulated shared memory on real hardware shared mem-

ory. For simulated architectures that have a bigger shared memory configuration, the spilled over memory is mapped on global memory. The address translation unit then swaps this chunk of memory from global address space to shared address space as and when needed.

Shared memory is simulated by using the hardware shared memory as a software managed cache. The load store units make a request to the software cache to access data from some addresses. The software cache is a n-? way associative cache. There is a conflict resolution unit that arbitrates the requests from all the load store units to the software cache. It also coalesces all the requests to the cache such that they map on to as fewer cache lines as possible for best performance. The accesses that map to the same cache line but require a line flush are serialized by the conflict resolution unit. For simplicity the simulator has as many load store units as real hardware. This does not simulate memory cycle accurate memory model, but does yield high performance while simulating applications as discussed in section ?

VI. EXPERIMENTAL EVALUATION

A. Micro-benchmarks

These benchmarks are intended to measure the limits of performance on the simulator.

- b) *Instruction Throughput:*
- c) *Thread/HTA Launch Rate:*
- d) *Barrier Rate:*
- e) *Function Call Rate:*
- f) *Branch Rate:*
- g) *Memory Requests:*

B. Simulator Speed

C. Simulator Accuracy

- h) *Dynamic Instruction Count:*
- i) *Branch Divergence:*
- j) *Memory Transactions:*

VII. RELATED WORK

- [?], [?]
- [?]
- [1]
- [?], [?], [?]
- [?], [?]
- [?]

VIII. CONCLUSION

REFERENCES

- [1] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneous systems," in *Proceedings of The Nineteenth International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2010.