SimTRaX: Simulation Infrastructure for Exploring Thousands of Cores

Konstantin Shkurko, Tim Grant,
Erik Brunvand, Daniel Kopta, Josef Spjut[a]*,*
Elena Vasiou, Ian Mallett, and Cem Yuksel
University of Utah

UUCS-18-001

[a]NVIDIA

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

March 29, 2018

Abstract

SimTRaX is a simulation infrastructure for simultaneous exploration of highly parallel accelerator architectures and how applications map to them. The infrastructure targets both cycle-accurate and functional simulation of architectures with thousands of simple cores that may share expensive computation and memory resources. A modified LLVM backend used to compile C++ programs for the simulated architecture allows the user to create custom instructions that access proposed special-purpose hardware and to debug and profile the applications being executed. The simulator models a full memory hierarchy including registers, local scratchpad RAM, shared caches, external memory channels, and DRAM main memory, leveraging the USIMM DRAM simulator to provide accurate dynamic latencies and power usage. SimTRaX provides a powerful and flexible infrastructure for exploring a class of extremely parallel architectures for parallel applications that are not easily simulated using existing simulators.

SimTRaX: Simulation Infrastructure for Exploring Thousands of Cores

Konstantin Shkurko
University of Utah
kshkurko@cs.utah.edu

Tim Grant
University of Utah
tgrant@cs.utah.edu

Erik Brunvand
University of Utah
elb@cs.utah.edu

Daniel Kopta
University of Utah
dkopta@cs.utah.edu

Josef Spjut
NVIDIA
josef.spjut@gmail.com

Elena Vasiou
University of Utah
elvasiou@cs.utah.edu

Ian Mallett
University of Utah
imallett@cs.utah.edu

Cem Yuksel
University of Utah
cem@cemyuksel.com

ABSTRACT

SimTRaX is a simulation infrastructure for simultaneous exploration of highly parallel accelerator architectures and how applications map to them. The infrastructure targets both cycle-accurate and functional simulation of architectures with thousands of simple cores that may share expensive computation and memory resources. A modified LLVM backend used to compile C++ programs for the simulated architecture allows the user to create custom instructions that access proposed special-purpose hardware and to debug and profile the applications being executed. The simulator models a full memory hierarchy including registers, local scratchpad RAM, shared caches, external memory channels, and DRAM main memory, leveraging the USIMM DRAM simulator to provide accurate dynamic latencies and power usage. SimTRaX provides a powerful and flexible infrastructure for exploring a class of extremely parallel architectures for parallel applications that are not easily simulated using existing simulators.

CCS CONCEPTS

• **Computing methodologies** → **Simulation tools**; *Graphics processors*; • **Computer systems organization** → *Multiple instruction, multiple data*;

KEYWORDS

Architecture simulation; single program multiple data; LLVM

ACM Reference Format:
Konstantin Shkurko, Tim Grant, Erik Brunvand, Daniel Kopta, Josef Spjut, Elena Vasiou, Ian Mallett, and Cem Yuksel. 2018. SimTRaX: Simulation Infrastructure for Exploring Thousands of Cores. In *Proceedings of 2018 Great Lakes Symposium on VLSI (GLSVLSI '18)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194554.3194650

GLSVLSI '18, May 23–25, 2018, Chicago, IL, USA

1 INTRODUCTION

When designing application-specific accelerator architectures, cycle-accurate simulators are indispensable tools for rapid exploration of the potential design space. Unlike trace-based simulators that can quickly evaluate particular sub-systems (such as disk, memory, network interfaces, etc.) under specific work-loads, cycle-accurate simulators provide the necessary details to capture the precise inner-workings of an entire system during the entire execution of the target application to completion. Thus they provide a testbed for potential modifications to the target application at the software level that would make it more suitable for the custom accelerator architecture.

This paper describes SimTRaX: a set of tools that resulted from exploring the design space of massively parallel accelerator architectures made up of simple in-order cores combined with shared compute and memory resources. The SimTRaX toolchain enables quick and easy experiments by supporting the following key features:

- Thousands of concurrent hardware threads
- Configurable thread resource sharing
- Fully cycle-accurate simulation
- Functional emulation mode for application debugging
- LLVM compiler support for custom hardware features
- Integrated source-level debugging and profiling
- Accurate DRAM system modeling
- Ample performance to run applications to completion rather than sampling specific kernels

Unlike most other simulators, SimTRaX is designed to simulate a large number of concurrent hardware threads with cycle-accuracy without introducing high-level approximations (modern computation platforms provide ample performance for this approach). We also provide an easy mechanism to allow software-access to various custom hardware features via LLVM compiler support [19]. We can use source-level debugging symbols provided by LLVM to debug and profile simulated architectures and applications on a wide variety of metrics (e.g. time spent per line, or energy per function call). Furthermore, accurate simulation of the dynamic random access memory (DRAM) behavior with SimTRaX produces reliable results, especially for applications that process a large amount of data with

unpredictable access patterns. Thus, SimTRaX has been used to simulate various hardware/software designs for accelerating fundamental graphics applications [17, 18, 29–31], particularly ones that are ill-suited for existing GPUs.

2 BACKGROUND

Although existing simulators enable experimentation within their own expected parameters, we have found that exploring a design space that is massively parallel can benefit from a new simulation infrastructure. Architectures in this class can be thought of as highly parallel tiled architectures but with every thread processing unit having its own program counter (PC), and thus able to run its own code in a multiple-instruction multiple-data (MIMD) mode. In practice, we find that these architectures, and the applications that map well to them, are often more correctly categorized as single-program multiple-data (SPMD) processing. In this case all hardware processors execute the same program, but because they have their own PCs, can be at different points in that program depending on their own data and control flow[1]. This can leverage data parallelism and at the same time allow sharing of large and complex functional units. Some examples of tiled architectures of this general sort include TRaX [30, 31], Rigel [13, 15, 16], Copernicus [10], STRaTA [17, 18], and SGRT [20]. In addition to tile-based simulators, there are many simulators for more traditional CPU and GPU architectures.

2.1 Uni- and Multi-Core

CPU architectures are designed to allow one to tens of threads to make progress at a high rate and are optimized for the latency of a given thread of execution rather than the throughput of the entire system. These architectures use hardware features such as out-of-order (OOO) processing and prediction to improve performance, often at the cost of power and transistor area. Such simulators include SimpleScalar [4], Simics [23], gem5 [2] and Hornet [26].

SimpleScalar supports a single or small number of MIPS-like cores while others support additional instruction sets, such as x86, ARM, and SPARC. These simulators are also capable of modeling sophisticated memory hierarchies. Simics, gem5 and Hornet also support booting an operating system, simulating peripheral devices, and allow varying the simulation fidelity from cycle-accurate to reduced accuracy which enables fast forwarding a running application.

2.2 GPU

The rise of data-parallel computing in recent years has created the need to simulate massively parallel single-instruction multiple-data (SIMD) or single-instruction multiple-thread (SIMT) architectures exemplified by graphics processing units (GPUs). GPGPU-Sim [1, 9] is capable of running unmodified CUDA and OpenCL workloads on a simulated NVIDIA-like GPU architecture. While the use of unmodified workloads is tremendously useful, keeping the simulator up to date with the fast pace of CUDA and OpenCL development, and the corresponding hardware changes, presents an ongoing challenge. GPGPU-Sim has been extended with a detailed power model to explore potential energy-saving techniques [21]. Recently, GPGPU-Sim and gem5 have been combined [25], which underscores the flexibility and interoperability of available simulation platforms. Multi2Sim [32] is another combined GPU-CPU simulator which implements an assortment of GPU and CPU architectures and is intended to study the interaction between CPU and GPU in a heterogeneous execution environment.

[1]Note that single-instruction multiple-thread (SIMT) processing as defined by NVIDIA [22] is often considered similar to single-program multiple-data (SPMD). However, SIMT support for divergent thread execution only tracks the divergence and still must mask off the results from diverged threads within a SIMT group. SPMD in our model allows threads to make individual progress while diverging.

2.3 Tiled

Tiled architectures generally achieve their performance through increased parallelism (more hardware computation threads) but with each of the thread processors being quite simple and stripped-down compared to a traditional core. Such architectures can scale into thousands of threads.

Some simulators leverage dynamic binary translation to enable simulating thousands of connected cores. PriME [8] focuses on a large number of chips executing a heterogeneous multi-threaded workloads simultaneously, and runs individual cores for an interval before synchronizing them. Zsim [28] scales well when simulating large architectures in parallel by first ignoring latencies and contentions, and then relies on event driven simulation to determine them.

Graphite [24] implements a modular memory system and provides three different accuracy modes. Although not cycle-accurate, it attempts to provide accurate timing estimates by using time stamps for synchronization. Sniper [5] extends Graphite with improvements in cache hierarchy and core model fidelity.

Among tile-based simulators, Rigel [13, 15] is probably the most similar to SimTRaX. Rigel simulates thousands of execution threads, allows configuration of the interconnect and the memory system, and includes development tools based on LLVM. Rigel applications are developed using a simple task-based bulk synchronous parallel programming model to be executed in a SPMD paradigm. While Rigel development seems to have stopped in 2012, it demonstrates that simulation infrastructures that include support for massive parallelism and compiler integration are interesting in a variety of domains.

3 SIMULATOR ARCHITECTURE

SimTRaX was designed to help explore application-specific accelerator architectures, modeled as a tiled, hierarchical, parallel architecture with a few thousand processing units. An example is shown in Figure 1. The lowest level is composed of simple in-order Thread Processors (TPs) that have limited execution resources such as register files, integer execution units, small scratchpad memories, and program counters. At the middle level, a Thread Multiprocessor (TM) can be formed by tiling a number of TPs alongside units they all share access to. The shared resources can include L1 caches (both instruction and data), complex computation units (such as floating point division), and special functional units tailored to the application space. At the highest level, a chip can consist of many TMs which share access to chip-wide resources, such as L2 data caches which connect to memory controllers for off-chip DRAM access. Representing the hardware in this way offers the ability to

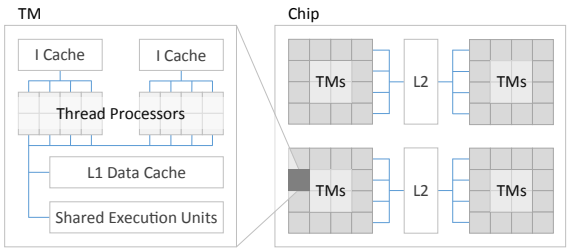


Figure 1: An example of a hierarchically organized parallel architecture [18] supported by SimTRaX. A number of simple in-order hardware Thread Processors (TPs) share access to execution units (FPU, etc.) and multi-banked instruction and L1 data caches. This structure forms a Thread Multiprocessor (TM), each of which share access to an L2 data cache with the other TMs tiled on chip. Off-chip memory channels are connected to each of the L2 data caches.

simulate a range of architectures, some of which would not map easily to other simulation platforms.

Although this hardware abstraction is general in terms of the types of parallel applications it supports, it works especially well with applications that can leverage a SPMD programming model. In this model, a single serial program is executed by each TP which relies on its own program counter. Parallelism is achieved through distributing data. Applications rely on an Application Programming Interface (API) to access synchronization hardware (atomics, barriers) at both the TM and chip-wide levels. These are used for distributing data and controlling application flow. When appropriate, each shared hardware unit arbitrates atomic access between all threads that request it.

The software design for SimTRaX mirrors the hardware description, and can be seamlessly compiled into either a cycle-accurate simulator or a functional simulator customized for the specific application. In the cycle-accurate mode, the simulator first initializes memories, caches, and hardware units based on configuration files that specify details like capacity, functionality, area, latency, issue width, and energy consumption. It then lets the application configure and load the main memory to prepare for execution. The simulation loop iteratively signals clock rise and fall across all functional units in the architecture. The simulator runs until all threads finish execution, then reports execution statistics and lets the application post-process memory to generate its output.

SimTRaX was designed to help explore application-specific accelerator architectures. As such, the execution model assumes accelerator memory to be loaded with appropriate data prior to execution and does not boot an OS. It should be possible to integrate SimTRaX as a co-processor in a full-system simulator.

Implementing a functional unit requires several components. First, it must provide initialization, clock rise, and clock fall functionality. Second, each unit must provide a description of the assembly instructions it supports. Custom instructions require API hooks to expose them to applications. The API hooks get converted into intrinsics which are later compiled into assembly instructions as described in Section 5.1. For functional simulation, these special "instruction" API calls are implemented with simple C++ functions that mimic the described behavior. The data flow between units is specified at simulator compile time; the functional unit must specify what other hardware units it connects to and designate at what level of the hierarchy it is shared.

A number of high-fidelity DRAM simulators could be used to provide detailed information about the memory system behavior [27, 33, 34]. For accuracy and ease of integration with a cycle-accurate simulator like SimTRaX, such a simulator should itself be cycle accurate and able to respond to individual memory requests rather than driven by memory traces. SimTRaX relies on a modified version of the Utah Simulated Memory Module (USIMM) simulator [6]. When thousands of threads attempt to access DRAM during execution, complex access patterns can emerge, leading to dynamic latencies far different from a simple average. For the cycle-accurate simulation, cache line requests that percolate to the DRAM memory controller are added into its read or write queues. Once all functional units have completed their clock simulation, SimTRaX simulates clock cycles for DRAM, maintaining the correct ratio between DRAM and processor frequencies.

3.1 Thousands of Hardware Threads

SimTRaX is capable of handling thousands of simple in-order hardware threads because they are organized hierarchically. There are additional benefits because the simulated machine is programmed using a SPMD model. First, it greatly simplifies programming a large parallel machine thus expediting benchmark application development. Secondly, it encourages the communication between TPs to remain sparse, thus enabling independent simulation even though they share access to some hardware units. Graphics and image processing applications are examples of broad application categories that can make effective use of the SPMD programming model.

The simulator is designed to keep overhead low when modeling the execution flow for each TP. For example, checking if an instruction can issue requires a few pointer de-references and comparisons mainly from checking if an execution unit has leftover capacity from its maximum issue width. The memory subsystem is more expensive, especially during cache misses, because it involves checking whether a particular cache line is in flight. During each clock, first we simulate all TM-wide units that can enqueue requests into globally-shared units. Then global units are simulated followed by DRAM simulation last. The issue unit within each TM manages the state of each TP and controls the order in which TPs issue instructions on any given cycle.

Based on application-specific co-processors developed with SimTRaX so far, there was no need for hardware shared memory subsystems such as snooping, cache directories, invalidations, etc. As a result, support for such systems is not currently included. While this limits SimTRaX's ability to simulate general purpose processors, it aids the ability to handle many threads cheaply. Applications which do require limited data sharing could, rely on special load/store instructions which bypass caching, similar to those existent in GPUs.

To accelerate the simulation times, SimTRaX utilizes multiple software simulation threads, each in charge of several TMs. Such granularity enables simulated hardware threads to share access to

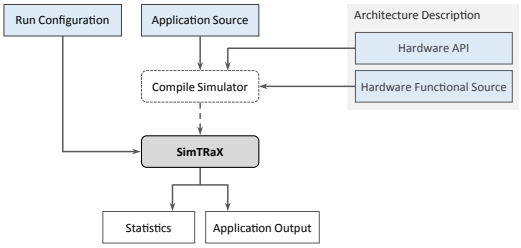


Figure 2: Work flow when using SimTRaX in functional mode. Both *application source* and *architecture description* are compiled into SimTRaX executable tailored for the specific application. Blue color highlights inputs.

TM-wide resources without any simulator synchronization. However, whenever a TP accesses a chip-wide unit like the L2 data cache, SimTRaX serializes accesses using typical software synchronization primitives or atomic operations. Thus simulation times depend in part on the particular architecture design and the frequency of accesses to globally-shared functional units by the simulated application. USIMM aggregates all memory requests generated by TMs, which are simulated by different threads. At the end of each clock cycle, USIMM simulates each DRAM channel in parallel. All simulation threads must also synchronize after each simulated cycle. Although relaxing this synchronization requirement increases simulator speed [24, 26], it introduces errors in predicted hardware performance.

3.2 Functional Simulation

While the cycle-accurate simulator provides great detail about the behavior of a new architecture, it is, of course, much slower than the hardware it simulates. SimTRaX also includes a functional simulator to accelerate the development of applications that benchmark the architecture being designed.

The SimTRaX functional simulator takes the form of a standalone executable that is specifically customized for a given application. Essentially, the application source is compiled to a native executable, using functional implementations of any custom instructions through the same API function calls that would generate the custom assembly. This work flow is shown in Figure 2. Note that certain custom instructions may not have a meaningful analog in the functional simulator, like a modification to caching behavior, but typically they would affect only the cycle-accurate performance of the simulated architecture.

The functional simulator also takes advantage of several parallel simulation threads, although each executes a single instance of the application as if it were a TP. The SPMD programming model enables seamless execution within both simulators without tailoring the application code to either functional or cycle-accurate simulator.

4 IMPORTANCE OF ACCURATE DRAM MODELING

The main memory of a simulated architecture typically consists of DRAM, which can be a primary consumer of both energy and time in data-bound applications, so modeling it accurately is crucial. Due to the internal structure and makeup of DRAM circuits, its operation and performance is far more complex than SRAM (on-chip caches). In addition to being dynamic, and thus needing periodic refresh operations, the latency and energy consumption of individual accesses can vary widely based on the patterns and addresses of other recent accesses [3, 12].

One important DRAM modeling issue is that the internal structure of DRAM chips is designed to support cache refill operations: the minimum amount of data transferred on a typical memory channel for one access is one cache line of (typically) 64 bytes. Internally to the DRAM chips, every cache-line-sized access fetches an entire *row* of data (typically 8KB) from one of the low-level internal memory arrays. The rows of data accessed in the set of DRAM chips make up what is called the DRAM *row buffer*. The process of reading from the low-level circuit array into the row buffer is destructive, and the static row buffer must be written back to the DRAM array in each chip to restore the data.

An important behavior of the row buffer is that because it is fast static memory, if the next cache refill access is also within that row buffer, known as a *row buffer hit*, then access to that data is dramatically faster and more energy efficient than if the access requires opening a new row. In a sense, the row buffer acts like a cache located inside the DRAM chips.

Typical DRAM systems contain many of the physical row buffers, each capable of storing one open row. DRAM is divided up into banks, each storing a certain block of memory (a range of addresses). Each bank represents many rows of data, and can keep one row open at a time in its physical row buffer.

As a first-order approximation, DRAM performance (both in terms of latency and power) is determined by the row buffer hit rate. This is a function of many interacting systems:

- The access patterns generated by the application
- The on-chip data caches filtering those accesses
- The physical configuration of the DRAM system: number of channels, number of DIMMs per channel, number of banks, size of a row, etc...
- Address mapping policy: the mapping of addresses to different regions (channel, bank, row) of DRAM
- Scheduling policy: the memory controller can enqueue many requests and issue them out-of-order, in an attempt to increase row buffer hit rates

The interactions of each of these systems must be precisely modeled to capture the intricacies of DRAM performance.

When simulating memory-bound applications, particularly with thousands of threads, an accurate memory model is of key importance and can have drastic impact on the results. SimTRaX reports detailed memory access statistics to help the user tune and evaluate their algorithmic/architectural innovations.

5 LLVM INTEGRATION

SimTRaX incorporates the LLVM toolchain [19] to facilitate developing applications used to benchmark new hardware units and estimate their benefits. The use of LLVM allows extending the instruction set architecture (ISA) with relative ease. Applications are written in a high-level language, such as C++, and then compiled

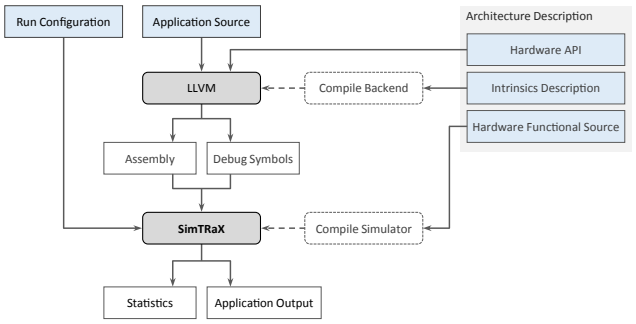


Figure 3: Work flow when using SimTRaX in cycle-accurate mode. Three components rely on the *architecture description* during compilation: LLVM backend, application assembly which relies on this backend, and SimTRaX executable. Blue color highlights inputs.

```
Time Profile                    100.00%
Main                            100.00%
| Shade                          47.57%
| | BVH::Intersect               31.16%
| | | Tri::Intersect             13.80%
| | | | Vec::Cross                2.15%
| | | | Vec::operator-            1.38%
| | | | Vec::Dot                  1.08%
| | | | Vec::Dot                  1.08%
| | | | Vec::operator-            0.92%
| | | | Vec::operator-            0.92%
...
```

```
FPU Energy Profile              100.00%
Main                            100.00%
| Shade                          51.10%
| | RandomReflection             15.61%
| | | Vec::Normal                11.48%
| | | | Vec::Length               7.40%
| | | | | sqrt                    7.27%
| | | | Vec::operator*=           0.07%
| | | sqrt                       3.20%
| | | Orthogonal                 0.22%
| | | | Vec::Cross               0.22%
...
```

Figure 4: An example profiler output for the same benchmark, showing both the computation time (left) and energy use of different functions (right).

to the modified ISA to serve as benchmarks for the new architecture design. In addition, we also include debugging and profiling features within the simulator.

Each custom hardware unit (including the shared memory hierarchy) may be supported by an API to expose it to the application. Because the API abstracts away details of the target simulator, protects the developer from maintaining multiple application copies, one for each simulation type. When compiling as the standalone functional simulator, each API call translates into a function call to the appropriate implementation of the "instruction." Figure 2 shows the work flow for the functional operation. Since the LLVM component is such an integral part of the simulator, it is important that the simulated processor cores are based upon assembly language that is supported by LLVM.

When the application source is compiled for the cycle-accurate simulator, an assembly file is produced for input to the simulator. Custom instructions are exposed by API calls which invoke compiler intrinsics that generate the appropriate assembly instruction. Figure 3 shows the work flow for the cycle-accurate operation. We rely on `clang` [19] for the frontend to compile an application written in a higher-level language like C++ into LLVM intermediate representation (IR). This IR is then fed into the LLVM backend implementing the ISA with any additional custom instructions, and LLVM generates an assembly file along with optional debug symbols. Finally, at execution time, the compiled binary is loaded into the instruction memory of the simulator and execution begins with the program counter for each TP set at the first instruction of the application.

5.1 Extending the ISA

Exploring the usefulness of new functional units often requires supporting them directly within the instruction set. The LLVM toolchain allows the ISA to remain fluid through straightforward extensions while avoiding the need to modify binaries. Although the architectures simulated using SimTRaX mostly rely on modified MIPS ISA [11], any other instruction set supported by the LLVM toolchain can be implemented and integrated into the simulator.

Adding a new instruction to the ISA is a fairly straightforward process, once the API and intended functionality are implemented in the simulator. The designer must define the intrinsic and its inputs and outputs such that they match the API. The intrinsic must emit an assembly instruction that matches what the SimTRaX assembler expects. Finally, the LLVM backend must be recompiled to incorporate these changes before use, labeled as *Compile Backend* in Figure 3. Incorporating the LLVM toolchain within a design exploration simulator allows to prototype and test new functional units quickly and easily.

5.2 Debugging and Profiling

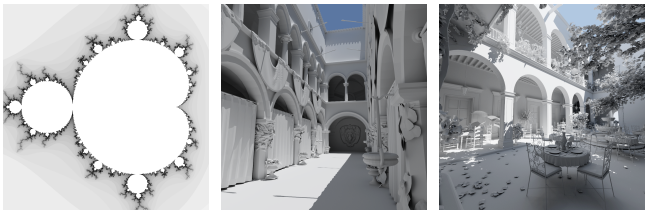
The LLVM toolchain can embed within the assembly of the compiled application. SimTRaX parses and interprets DWARF [7] debugging symbols, and includes a built-in debugger and profiler that operate on the full cycle-accurate state of the simulated machine.

The SimTRaX profiler has perfect run time execution information and does not rely on time sampling. This allows the user to see, with single cycle accuracy, the resources spent in various regions of the program's source code. The profiler can also annotate source code with any other information available to the simulator, like the energy spent by hardware units, the number of cache misses, the number of arbitration conflicts, etc. Fig. 4 shows an example of the profiler's output, customized to show two different metrics: on the left, a typical time profile, and on the right, a profile of the energy spent in floating point arithmetic units. This simple example reveals that vector operations, denoted by `Vec::*`, consume most time, but floating point square root consumes most energy because the fixed-function hardware unit is extremely power consumptive.

The profiler can expose, at the application source code level, the exact source of various behaviors on the chip, providing researchers more insight about where to target their efforts. As a result, SimTRaX can provide a more complete understanding of the impact aspects of the hardware have on performance when evaluating new architectures.

6 EVALUATION

We evaluate SimTRaX performance when simulating the TRaX architecture [30, 31]. Selected graphics benchmarks, shown in Figure 5, evaluate the effects of accessing memory (excluding TM's



(a) Mandelbrot (b) Path tracing the "Crytek Sponza" scene (c) Path tracing the "San Miguel" scene

Figure 5: Representative output images generated using Mandelbrot and path tracing benchmark applications.

local store) on simulator performance. The Mandelbrot benchmark includes a large percentage of computation, where the memory subsystem is accessed only for initial parameters and image output. Path tracing, a popular algorithm that generates photo-realistic images of virtual scenes [14], relies on iterating over tree data structures per pixel and thus can access memory rather incoherently. Smaller scenes, like Crytek Sponza (262 thousand triangles), generate more coherent accesses and thus higher cache hit rates than larger scenes like San Miguel (10.5 million triangles).

The output image resolution is 1024×1024, and path tracing benchmark uses maximum ray depth of three. Benchmarked TRaX configuration combines 32-wide TMs into a chip with 128 to 2048 total TPs running at 1GHz. Each TM contains shared execution units, and a 32KB L1 data cache. The L1 data cache from each TM is assigned to one of four global 512KB L2 data caches. They in turn connect to the DRAM memory controller set up as 8 channel GDDR5 for a maximum of 512GB/s bandwidth. The performance of the simulated architecture is measured in frames per second (FPS), which is the inverse of total time taken to generate the image. The simulator is evaluated on a hexa-core Intel i7-5820K CPU running at 3.3GHz and 32GB of DDR4 memory.

6.1 Architecture Flexibility

SimTRaX was used by researchers to explore dedicated graphics co-processor architectures [17, 18, 29–31] targeting path tracing algorithm. Based on the data flow, this work can be segmented into three distinct types of architectures.

TRaX, Figure 6a, relies on a more familiar memory layout: TMs are split between four L2 data caches which share access to DRAM. The researchers add a few global registers and an atomic increment instruction, which returns incremented value: `ATOMIC_INC destReg srcGblReg`.

STRaTA, Figure 6b, connects all TMs to a single smaller L2 data cache and a shared on-chip hardware ray queue. STRaTA also modifies the ISA to allow a master TP in each TM to schedule a ray queue for the entire TM and to enable each TP to read / write ray data from the assigned queue. A ray is read into dedicated TP registers and the success is stored into the `destReg` register: `READ_RAY destReg`.

Dual streaming architecture, Figure 6c, connects all TMs to several specialized shared on-chip units designed to work with both ray and scene data streams. The software and the hardware are designed concurrently so as to prevent TMs from accessing memory randomly by prefetching necessary data from DRAM prior to scheduling execution. The dual streaming architecture also modifies the ISA to read / write rays into the ray stream (stored in DRAM), fetch scene stream from on-chip buffer and atomically update ray hit records. The updater loads necessary data stored in TP's local store address given by `srcAddrR` register value, compares with and potentially overwrites data stored in main memory at the address given by `destAddrR` register value: `UPDATE_HR destAddrR srcAddrR`.

The flexibility of SimTRaX and the ease of modifying the ISA enabled simulating performance of these three different sets of co-processor architectures. The authors also rely on detailed performance and energy reports generated by SimTRaX to evaluate their proposed designs.

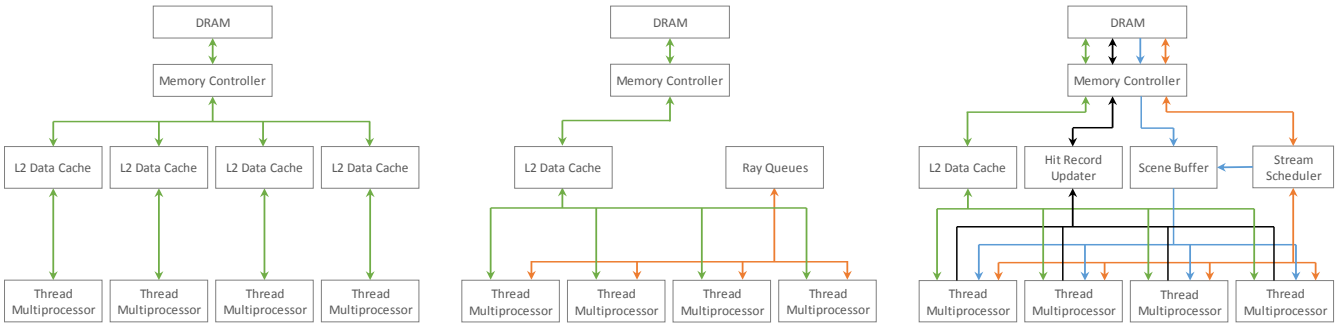
6.2 DRAM

To compare the effects of simulating DRAM accurately, consider simulated performance executing Path Tracing on a TRaX architecture with varying number of TPs, shown in Figure 7. This test includes four different models for DRAM accesses. *USIMM* simulates DRAM behavior correctly. The *Free Memory* model makes all memory accesses take one cycle, bypassing L2 data caches and DRAM memory controller altogether. This model is useful for measuring the compute-bound limit to the performance of the simulated architecture. The *100 cycles* test sets DRAM access latency to 100 cycles for each access and limits maximum bandwidth per cycle without simulating read/write queues per channel. The *Correct Avg Latency* test similarly uses a constant latency for DRAM accesses, which is manually set to match the average access latency generated by *USIMM* simulations (60 - 470 cycle read latencies). The simulated performance plateaus for the San Miguel scene at 512 TPs when simulating DRAM accurately because the configuration is DRAM bandwidth-limited. However, without accurate DRAM simulation, this behavior cannot be accurately captured, and the simulated performance results scale almost linearly with the number of TPs. It is clear that the accuracy of simulating the DRAM dynamic behavior can have a profound effect on reported results of the simulator.

6.3 Simulator Performance

Although cycle-accurate simulators enable highly accurate and detailed evaluation of proposed architectures, they are considered slow. However, the combination of architectural choices, using multi-threading and general improvements in processor performance help make cycle-accurate simulations feasible.

Figure 8 shows SimTRaX performance measured in millions of simulated instructions per second (MSIPS) as a function of number of TPs while using four simulator threads. SimTRaX simulates at a rate of 0.33 to 7.38 MSIPS depending on how frequently the workload and hardware communicate with chip-wide units. This performance is good enough to enable co-processors with thousands of threads and new hardware units, and benchmark applications running to completion. The performance dips for both Crytek Sponza and San Miguel scenes around 1024 TPs because those configurations become DRAM bandwidth-limited. This results in higher



(a) TRaX [30, 31] **(b) STRaTA [17, 18]** **(c) Dual Streaming [29]**

Figure 6: Different dedicated graphics co-processor architectures rely on SimTRaX to evaluate their designs. Lines connecting units illustrate data flow colored based on type: (orange) ray data, (blue) scene data, (black) hit records, and (green) other data.

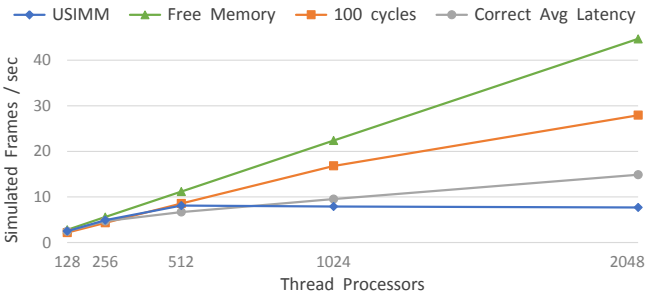


Figure 7: Effect of DRAM accuracy on simulated performance of path tracing on TRaX architecture for San Miguel scene.

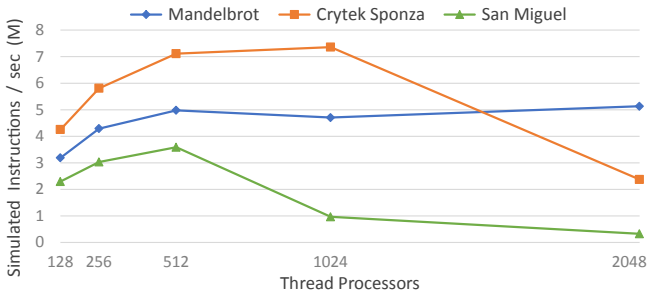


Figure 8: SimTRaX performance measured in millions of simulated instructions per second.

DRAM latencies and more simultaneous requests to the DRAM simulation component, requiring heavy utilization of software locks.

7 CONCLUSION AND FUTURE WORK

We have described SimTRaX, a simulator designed to explore highly parallel co-processor architectures with thousands of simple threads sharing access to expensive computation and memory resources. The simulation infrastructure provides several components important for quick exploration of possible architecture designs concurrently with targeted software modifications: combined cycle-accurate and functional simulation capability, flexibility in how functional units are connected, a highly accurate DRAM model, and integration of the LLVM toolchain for easy ISA extensions and compiling applications written in a high-level language.

The combination of LLVM debugging information and cycle-accurate system state can be used to generate a highly detailed profile with information beyond just execution time. For example, a user could generate a profile of energy spent in the various chip components during execution. This focus on massively parallel architecture, integration with LLVM, and the resulting ability to profile simulated code in detail on the simulated architecture is not available on other simulator platforms that we are aware of.

An interesting future direction is extending SimTRaX to support more sophisticated memory sharing mechanisms, which would enable SimTRaX to simulate more general architectures. It would also be interesting to explore how we can enable simulations to rely on task-based parallelism while relaxing how often simulation threads need to synchronize. To aid in evaluating radically different architectures faster, SimTRaX's modularity should extend further and enable a node-based graphical interface to configure how functional units are connected and shared. Finally, an automatic system can be used to find optimal architecture configurations by using gradient-descent-like algorithms to optimize a metric output by the simulator (like energy or performance).

ACKNOWLEDGMENTS

This material is supported in part by the National Science Foundation under Grant No. 1409129. The authors thank Solomon Boulos, Al Davis, Spencer Kellis, Andrew Kensler, Steve Parker, Paymon Saebi, Pete Shirley, and Utah Architecture group for discussions and simulator contributions. Crytek Sponza is from Frank Meinl at Crytek and Marko Dabrovic and San Miguel is from Guillermo Leal Laguno.

REFERENCES

[1] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*. 163–174.
[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Arch. News*

39, 2 (Aug. 2011).
[3] E. Brunvand, D. Kopta, and N. Chatterjee. 2014. Why Graphics Programmers Need to Know About DRAM. In *ACM SIGGRAPH Courses*.
[4] D. Burger and T. Austin. 1997. *The Simplescalar Toolset, Version 2.0.* Technical Report TR-97-1342. University of Wisconsin-Madison.
[5] T. E. Carlson, W. Heirman, and L. Eeckhout. 2011. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-core Simulation. In *SC*.
[6] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. 2012. *USIMM: the Utah SImulated Memory Module.* Technical Report UUCS-12-02. Univ. of Utah.
[7] DWARF Debugging Information Format Committee. 2010. DWARF Debugging Information Format V. 4. http://www.dwarfstd.org/doc/DWARF4.pdf. (2010).
[8] Y. Fu and D. Wentzlaff. 2014. PriME: A parallel and distributed simulator for thousand-core chips. In *ISPASS*. IEEE, 116–125.
[9] W. WL Fung, I. Sham, G. Yuan, and T. M Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *IEEE/ACM Micro*. 407–420.
[10] V. Govindaraju, P. Djeu, K. Sankaralingam, M. Vernon, and W. R. Mark. 2008. Toward A Multicore Architecture for Real-time Ray-tracing. In *IEEE/ACM Micro*.
[11] Imagination Technologies. 2015. MIPS Architectures. http://www.imgtec.com/mips/architectures/. (2015).
[12] B. Jacob, S. W. Ng, and D. T. Wang. 2008. *Memory Systems - Cache, DRAM, Disk*. Elsevier.
[13] D. Johnson, M. Johnson, J. Kelm, W. Tuohy, S. Lumetta, and S. Patel. 2011. Rigel: A 1,024-Core Single-Chip Accelerator Architecture. *IEEE Micro* 31, 4 (July 2011), 30–41.
[14] J. T. Kajiya. 1986. The Rendering Equation. In *Proceedings of SIGGRAPH*. 143–150.
[15] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. 2009. Rigel: an architecture and scalable programming interface for a 1000-core accelerator. In *ISCA*.
[16] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. 2010. Cohesion: a hybrid memory model for accelerators. In *ISCA*.
[17] D. Kopta, K. Shkurko, J. Spjut, E. Brunvand, and A. Davis. 2013. An energy and bandwidth efficient ray tracing architecture. In *Proc. HPG*. 121–128.
[18] D. Kopta, K. Shkurko, J. Spjut, E. Brunvand, and A. Davis. 2015. Memory Considerations for Low Energy Ray Tracing. *Comp. Gr. Forum* 34, 1 (2015), 47–59.
[19] C. Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*. 1–2.
[20] W. Lee, Y. Shin, J. Lee, J. Kim, J. Nah, S. Jung, S. Lee, H. Park, and T. Han. 2013. SGRT: A mobile GPU architecture for real-time ray tracing. In *Proc. HPG*.
[21] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M Aamodt, and V. J. Reddi. 2013. GPUWattch: enabling energy optimizations in GPGPUs. *SIGARCH Comput. Arch. News* 41, 3 (2013).
[22] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. 2008. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE* 28, 2 (2008), 39–55.
[23] P. S Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.
[24] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. 2010. Graphite: A distributed parallel simulator for multicores. In *HPCA*. IEEE, 1–12.
[25] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. 2015. Gem5-GPU: A Heterogeneous CPU-GPU Simulator. *IEEE Computer Architecture Letters* 14, 1 (Jan 2015).
[26] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, N. Zheng, and S. Devadas. 2012. HORNET: A Cycle-Level Multicore Simulator. *IEEE Trans. on CAD* 31, 6 (2012), 890–903.
[27] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A Cycle Accurate Memory System Simulator. *IEEE Comput. Arch. Let.* 10, 1 (Jan. 2011), 16–19.
[28] D. Sanchez and C. Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-core Systems. In *ISCA*.
[29] K. Shkurko, T. Grant, D. Kopta, I. Mallett, C. Yuksel, and E. Brunvand. 2017. Dual Streaming for Hardware-Accelerated Ray Tracing. In *Proc. HPG*.
[30] J. Spjut, A. Kensler, D. Kopta, and E. Brunvand. 2009. TRaX: A Multicore Hardware Architecture for Real-Time Ray Tracing. *IEEE Trans on CAD* 28, 12 (2009).
[31] J. Spjut, D. Kopta, E. Brunvand, and A. Davis. 2012. A Mobile Accelerator Architecture for Ray Tracing. In *3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-3)*.
[32] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli. 2012. Multi2Sim: A Simulation Framework for CPU-GPU Computing. In *PACT*.
[33] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. 2005. DRAMsim: A Memory System Simulator. *SIGARCH Comput. Arch. News* 33, 4 (Nov. 2005).
[34] K. Yoongu, Y. Weikun, and M. Onur. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Computer Arch. Letters* 15, 1 (2016).