









Arches: A Cycle-Level Hardware Simulation Framework for Exploring Massively Parallel Ray Tracing Architectures

J. Haydel¹  G. Bhokare¹  K. Zeng¹  P. Hong¹  S. Kondguli²  B. Budge²  E. Brunvand¹  C. Yuksel¹ 

¹University of Utah, USA

²Meta Reality Labs, USA

Abstract

We introduce *Arches*, a hardware simulation framework designed to explore and evaluate massively parallel ray-tracing architectures. Operating at the cycle level, *Arches* captures detailed performance metrics, including computational throughput, on-chip data movement across processors, caches, and off-chip communication via an accurate memory system model. The framework is modular, allowing flexible configuration and interconnection of processor cores, caches, and custom hardware units, enabling easy exploration of diverse hardware architectures. *Arches* supports high-performance parallel execution, simulating complex ray tracing workloads to image completion. It leverages the GNU toolchain, allowing users to write C++ software targeting both the simulated architecture and native execution for debugging, including support for custom instructions to control specialized hardware components. The framework provides comprehensive performance instrumentation, offering insights into time-varying statistics across all modules and identifying performance bottlenecks. Our evaluations demonstrate that *Arches* delivers performance estimates closely matching real hardware, offering faster and more accurate simulations than existing open-source hardware simulators. Its modularity also makes it a valuable tool for exploring alternative parallel computing strategies for high-performance ray tracing, and its extensibility enables adaptation for other workloads or general-purpose computation.

CCS Concepts

• *Computing methodologies* → *Graphics processors; Ray tracing;*

1. Introduction

A cycle-level hardware simulator is an essential component of architecture research, because manufacturing custom computing hardware is both expensive and time-consuming. Transistor-level designs require many person hours and high levels of expertise to develop. Using a hardware simulator avoids this cost and complexity, allowing quick testing and iteration without the need for transistor-level design process at each iteration. It also opens the door for testing fictional components and incomplete designs while providing power and performance estimates not afforded by higher-level functional simulations.

Despite these advantages, developing a reliable and flexible hardware simulator is still a challenging task. This challenge is exacerbated when targeting massively parallel ray tracing architectures. Notably, ray tracing systems can be either compute bound or memory bound depending on scene, rendering algorithm, and specifics of the hardware architecture, requiring a cycle-level simulator to correctly identify bottlenecks. Additionally, not all rays have the same cost, and ray tracing performance and bottlenecks can shift during the course of rendering a frame. This requires a fast simulator that can efficiently handle a large number of comput-

ing units and process the entire rendering task to completion. This is needed not only for evaluating the performance during different stages of rendering, but also for verifying that the final rendered image is correctly generated. Furthermore, custom hardware units are important for high-performance ray intersection tests and traversals through the scene hierarchy. Therefore, a flexible simulator design is needed to experiment with different resource distributions and memory configurations.

Existing hardware simulators are unable to deliver most of these features. In our experience, they typically scale poorly to a large number of computing units. Simulation performance is paramount for quick iteration as a single frame of interesting ray-tracing work can easily take hours to simulate on existing simulators. Most of them target specific hardware designs and lack a modular interface that would allow easy redesign of the simulated architecture.

We present *Arches*, a new cycle-level hardware simulator infrastructure that was designed to meet the needs of massively parallel hardware ray tracing architecture research. We discuss the design decisions we followed for developing *Arches* and its tool set, and provide evaluations showing the performance, flexibility, and accuracy of our hardware simulator.

Arches is specifically designed for efficiently simulating a large number of hardware units via parallel computation. The units are abstracted as different types of modules that can communicate with each other through interconnects. For each cycle, the modules execute their custom functions, processing their inputs and generating their outputs to be passed onto the other modules. Different hardware architectures can be simulated by developing the necessary modules and specifying their connections using a high-level programming interface in C++. We demonstrate the flexibility of this modular structure by presenting results with various hardware architectures we simulated using Arches.

We use a new cycle-level simulation approach that splits the computation of a cycle into two phases. It allows computing all simulated modules in parallel, avoiding data hazards (and the overhead for managing potential data hazards of parallel computation) by processing message sending and receiving between modules in two separate passes. This offers an efficient parallel execution model that is crucial for simulating a large number of modules without making any assumptions on how the modules operate, including their latencies and dependencies on other modules.

Arches is integrated with the GNU toolchain, allowing users to use C++ to develop both the software that is targeted to run on the simulated hardware architecture as well as the architecture models themselves. This includes custom instructions that can be added to the instruction set for controlling custom hardware units. This toolchain also allows compiling the same software to run natively on existing CPUs without the hardware simulator for software development and debugging purposes.

Arches also includes flexible instrumentation for measuring the performance statistics of all modules. These can be collected as aggregate values or time-varying measurements through the course of a simulation. This helps in identifying bottlenecks that can shift during rendering.

We provide experiments comparing Arches to alternative simulators and to actual hardware with a similar architecture. Our results show that Arches delivers significantly faster hardware simulation and can reasonably estimate the performance statistics collected from real hardware.

We release Arches and its source code, including its documentation, scripts, and example hardware architecture configurations: <https://github.com/Utah-Graphics-Lab/arches>

2. Prior work

Simulators have been a crucial component in computer architecture research for decades, enabling researchers to explore innovative designs for various hardware components within a computer system without having to fabricate them in silicon or map them to gate-level computing surfaces such as field programmable gate arrays (FPGAs). Hardware behavior can be simulated at varying levels of detail and complexity from functional and trace-driven to cycle-level and execution-driven simulations.

CPU simulators have undergone significant evolution, transitioning from simple functional and trace-driven simulators [BA97]

to more complex cycle-level [AR13a] and execution-driven simulators [BBB*11], capable of executing full operating systems [BBB*11, AR13b]. These simulators offer a great way to model CPU behavior at varying degrees of accuracy and complexity. However, their focus on modeling complex CPU-level behavior and their choice of ISAs limits their extensibility to modeling GPU behavior involving thousands of cores, dedicated fixed functional units, and graphics-specific rendering algorithms.

The development of GPU simulators has branched into two distinct paths. Where one branch focused on simulating GPGPUs, with notable examples including GemDriod [CNYS*14], GPGPU-Sim [KSAR20], gem5-gpu [PHO*15], and gem5-Aladin [SXS*16]. The other branch focused on graphics rendering simulators, which evolved with advances in graphics rendering techniques.

Early graphics rendering simulators supported OpenGL-based rasterization for immediate mode rendering [dBGR*06, SLS04] and tile-based rendering [APX13]. Later simulators, such as GLTraceSim [SCHBS17], enabled the simulation of GPU behavior using traces generated by 3D graphics libraries. More recent simulators can support Vulkan-based full-system simulations for hybrid tile-based rendering techniques [GA19, TSS*23]. These simulators are heavily dependent on the underlying graphics APIs, inadvertently tying architectural explorations to the limitations of specific APIs.

Raytracing simulators are relatively new, with earlier variants providing the ability to model dedicated ray-tracing accelerators [SGB*18]. More recent ones, such as Vulkan-Sim [SCL*22], offer support for popular raytracing APIs. However, similar to their rasterizer counterparts, reliance of these simulators on graphics APIs curbs their continued usage, as they significantly limit the introduction of new instructions and hardware components within the architectural design. In contrast, Arches provides a flexible framework for introducing new instructions and hardware components and simulating their cycle-level behavior, making it an attractive solution for researchers seeking to explore innovative ray-tracing architectures. Arches also executes the instructions it simulates during execution stage instead of relying on program traces for data and control dependency.

In addition to processing behavior, modeling memory access behavior at the DRAM level is essential for retaining high simulator accuracy. DRAMSim [LYR*20] and USIMM [CBS*12] are popular simulators that model DRAM behavior, but they support only a subset of DRAM standards. Ramulator [LTB*24] provides cycle-level simulation support for a variety of DRAM standards with an option to easily incorporate newer standards. Arches integrates Ramulator as a module, providing it with a flexible and reliable ability to model memory access behavior.

3. Modular Hardware Simulation Framework

Arches uses a modular simulation framework, where each hardware component is implemented as a module that communicates with other modules through interconnects. There are two important reasons for this modular design:

1. **Performance:** By representing a hardware architecture as a collection of modules that operate every cycle to perform their custom tasks, this allows Arches to execute all modules in parallel, offering a significant performance boost over hardware simulators that are single-threaded [SCL*22] or parallelize poorly [SGB*18].
2. **Flexibility:** Most computing architectures use similar components, though the number of components and how they are connected vary. But, for research purposes it is also desirable to be able to posit new and different modules that are outside the norm. Our modular design provides the flexibility of easily representing different hardware architectures, including custom components, as modules.

In this section, we present the details of this modular framework and how Arches parallelizes the computation.

3.1. Modules

A module in Arches can represent any hardware component. It can be an entire general-purpose processor core or a part of a processor core, such as the floating-point unit. Caches and custom special-purpose units are also represented as modules.

A module can be as small as a fixed-function hardware unit that performs a simple task or as large as a thread multiprocessor with many cores. Using larger modules allows us to simulate hardware functionality at a higher level, offering better simulation performance. Having smaller modules provides the flexibility of reusing them with different architecture designs and sharing them among multiple modules, such as a floating-point division unit shared by multiple processor cores.

The simulator treats all modules the same way by calling their execution functions for each cycle. Modules are responsible for performing their functionalities when prompted, processing their inputs and generating their outputs/requests by communicating with the other modules through interconnects.

To maintain cycle accuracy, modules produce their outputs on the cycle in which they would be ready for consumption by another module. If a module takes multiple cycles to complete an action, a *latency FIFO* is used to delay the results until the correct global cycle. If needed, all modules can access the global cycle count, although currently only the DRAM module (see Section 4.3) does this because it is able to run at a non-integer ratio to the rest of the modules in the simulated system.

3.2. Interconnects

Modules are connected to each other via interconnects meant to model, at a high level, the physical interconnects present in a hardware instantiation. These connections are set up while specifying the hardware architecture that will be simulated. These interconnects are not switch-level models of actual interconnects, but behavior models for simulating and instrumenting the communications among modules.

The interconnects serve two main purposes:

1. They provide arbitration for communication between modules and an estimate of interconnection delay through that network.

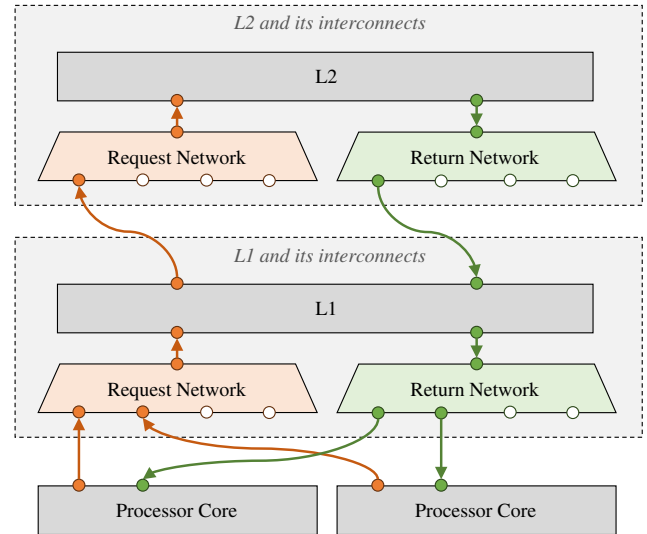


Figure 1: A diagram of interconnects between processor core modules, an L1 module, and an L2 cache module. The colored circles are the ports of the interconnects. The arrows show the connections from sink ports and towards the source ports.

2. They allow parallel simulation of modules by eliminating potential data hazards during data/message passing between modules.

Modules are connected to *ports* on either end of interconnects. Transactions flow from the *source* end towards the *sink* end. Modules connected to *source* ports can only write to these ports, and module connected to *sink* ports can only read from them. The module reading from a sink port can clear it to acknowledge that the transaction has been accepted.

The interconnect passes data from its source ports to its sink ports. Thus, only the interconnect itself can read from source ports and write to sink ports.

Each interconnect is owned and operated by one module. It allows the owner module to communicate with one or more other modules. Each one of these other modules is connected to a unique *port* of the interconnect. This is critical for avoiding data hazards during parallel simulation.

If the owner module is using the interconnect to receive transactions from other modules, it is called a *request network*. The owner module is connected to its sink ports, and the other modules are connected to the source ports. Conversely, if the owner module is using the interconnect to send transactions, it is a *return network*. Then, the owner is connected to the source ports, and the other modules are connected to its sink ports.

Figure 1 shows an example L1 cache module connected to multiple processor core modules and an L2 cache module via interconnects. The L1 cache owns both a request network and a return network. Each core is connected to one source port of the request network and one sink port of the return network. Cores can send request transactions through the request network by writing to that network's source ports. The cache then uses this interconnect to ac-

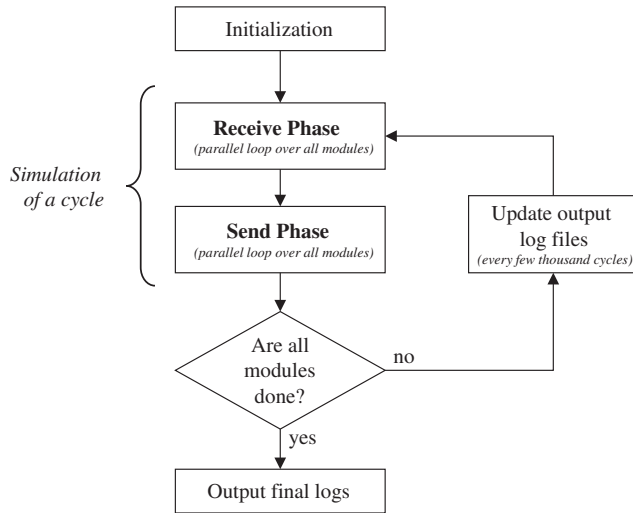


Figure 2: Cycle-level hardware simulation flowchart of Arches, starting with initialization and ending with outputting the final log files. A two-phase cycle simulation loop runs, where each phase is a parallel loop over all modules.

cept one of these request transactions. The interconnect clears the port to send an accept signal to the requesting core. When the cache is ready to return the data requested by the core, it sends a return transaction through the return network by writing to its source port. This interconnect passes the transaction to the corresponding sink port that is connected to the same core. When the core reads this transaction, the port is cleared. The L2 cache operates in an identical fashion, servicing multiple L1 cache modules.

An interconnect can have multiple source and sink ports. For instance, a cache might contain multiple banks that need to be fairly shared by multiple processor cores. To facilitate this, its request network can have multiple source ports that are connected to other modules and multiple sink ports, each corresponding to a different bank, but connected to the same cache module.

An interconnect can simultaneously receive multiple requests on different input ports. When that happens, the interconnect must decide which request(s) to accept. The ports for the accepted requests are cleared to signal the requesting modules that their requests are accepted. The other ports are kept unchanged.

The interconnects can simulate various forms of arbitration. A common one is a crossbar switch that implements round-robin arbitration over the input ports. These crossbars work by keeping a mask of all pending requests that would like to access a shared module. Each cycle, a sink arbiter selects the highest priority source from the set. Each time a source is selected, it becomes the lowest priority index, and the source after it (in round-robin order) becomes the highest priority. Interconnects can also have pipelines that can be used for simulating arbitration latency.

3.3. Simulation Cycle

Figure 2 shows the flowchart of cycle-level hardware simulation with Arches. Each cycle is split into two phases: *receive* and *send*.

In the receive phase, modules can receive their transactions from all sink ports to which they are connected. If a module owns a receive network, it must first run it to handle the arbitration. Then, it can receive the transaction from that interconnect. The modules scan the byte arrays for pending request transactions. When a pending request is found, it can either be acknowledged by clearing the byte or left on the sink port to be processed on the next cycle. Once the request is read from the sink port, the corresponding byte is cleared, representing that the request was acknowledged. Then, the module can perform its internal functions.

If a module takes more than a single cycle to complete, a *latency FIFO* is used to delay the result until the correct cycle. If a module is fully pipelined, it can generate a result on every cycle into the latency FIFO. If not, the module is responsible for entering results only in the appropriate slots in the latency FIFO.

After all modules have completed their receive phase, the simulator synchronizes the simulation threads and begins executing the send phase.

During the send phase of a cycle, modules can write their request transactions to the source ports to which they are connected. Since no two modules may share a port, this operation remains thread-safe. While sending a request, modules also write a single corresponding byte to indicate that there is a request pending on that source port. Modules that own a return network must also run that interconnect at the end of their send phase operations.

When a receiving module finishes processing a request, it may optionally want to send a response. For instance, in the case of a load instruction issued to a cache module, the cache must return the requested data to the processor that issues the load request. In this case, the same series of steps is carried out on the return network. During the send phase, the sending unit writes the response to the corresponding source port and sets the byte to represent a pending response. During the next receive phase, the receiving unit checks the byte to see if a response is pending, and if so, it may read the corresponding data and clear the byte, representing an acknowledgment.

While this approach is simple, we find that it works very well in practice, since the receiving module needs to read the data associated with requests only when they are acknowledged. All other requests incur only a single byte of memory traffic, and due to the send/receive phases, updating the state of the byte array does not require the use of atomics or mutexes for parallelizing the execution of the modules during the simulation.

Having two phases per cycle requires all simulation threads to synchronize twice per cycle. In practice, these incur a non-trivial overhead in the simulation. Even using a highly optimized threading library like TBB (thread building blocks), the synchronization overhead is still considerable. In practice, however, we find that when simulating massively parallel architectures like those described in this paper, the amount of simulation work that needs to be done per cycle renders the synchronization overhead marginal.

3.4. Instrumentation

One of the most important features of any hardware simulator is the ability to glean useful information about the activity within the simulated hardware. To this end, Arches provides extensive logging for all modules.

Modules can output custom information about their operations. For example, processor cores can record the issue and stall cycles, along with the stall type (i.e. pipeline or data hazard) and the instruction that caused the stall. Similarly, caches can record statistics about hit rates, occupancy, and stalls.

Arches provides an easy way to aggregate these statistics. The user can choose to output per-module statistics or average them across a set of modules. For example, it is possible to output separate statistics for each processor core, average statistics for a collection of cores, or average statistics across all cores of the entire simulated architecture.

Statistics can also be aggregated over time (represented as clock cycles), including output of the statistics for a specified range of cycles. This flexibility makes it easy to add detail to the output where needed and to consolidate data where aggregate information is preferred.

4. Implementation

We have implemented various modules, including general-purpose processor cores, various cache models, interconnect types, and special-purpose units for ray tracing. They form a sufficient set for simulating different hardware architectures for ray tracing. We discuss their details in this section.

4.1. Processor Cores

We have implemented a general-purpose processor core module that is capable of executing RISC-V instructions. This processor core has a simple 5-stage pipeline, where instructions are fetched, decoded, and issued in order, but may retire out of order. In this pipeline, dependencies are checked using a scoreboard and, optionally, multi-threading can be enabled to allow a processor core to switch to a new thread when data hazards are detected.

For resource sharing, a core (or a set of cores) keeps a table that maps instruction types to modules. Other entries may point to shared resources that require sending requests to execute but may need to communicate with a shared functional unit (implemented as a separate module), e.g. to execute a floating-point divide instruction. The number, latencies, and initiation intervals of these pipelines are configurable, along with instruction-module pairings. Null entries in this table correspond to an instruction type that can be executed directly by the core, e.g. a core may be able to execute an add instructions natively,

Since the simulator is designed to simulate hardware architectures that contain thousands of cores, it must be relatively cheap to decode and execute each instruction within the processor core module. To efficiently simulate each instruction, we implement them as separate functions that operate on a register file and a program counter. The corresponding function of an instruction is stored in

a lookup table. When a function is executed, it performs its operation and then updates the register file program counter state. For some instructions, such as load and store, this function generates a request that is passed to another module, such as the lowest level cache module of the memory system.

In addition to this simple processor core, we have implemented a module for a generalized shared functional unit (SFU). Since the ISA (instruction set architecture) implementation takes care of the functionality of instructions and the interconnects handle resource sharing, these SFUs only need to simulate the timing of their pipelines. When an instruction is received by an SFU, it inserts the request into a latency queue with a variable initiation interval. When the request leaves the queue, a write-back transaction is sent back to the core to indicate that the destination register has become available.

4.2. Interconnect Implementation

In a tiled multi-core architecture, there are thousands of shared resources that communicate amongst each other and require arbitration at every cycle. Therefore, it is important that the communication and arbitration operation of selecting a port is efficient to execute.

The interconnect design supports various policies, though our implementation currently only includes round-robin arbitrated crossbars. A naive implementation of a round-robin arbiter might keep an array of requests and loop over the entire array every cycle, looking for the next request. This is wasteful not only in terms of memory footprint, but also computational cost. Instead, we choose to simulate the arbiters by keeping a bit mask of all requesting inputs. This minimizes the storage cost and allows selection of the highest priority index in constant time with only a handful of micro instructions. This optimization allows us to simulate the operation of thousands of interconnects per cycle, enabling every module to fairly arbitrate over its inputs.

In addition to the crossbars used by shared units to arbitrate over their inputs, there is also a standalone crossbar module that can be used to form a partitioned memory hierarchy. These work by mapping different parts of the address space to different memory partitions. This allows each partition, generally composed of several slices of cache and a single memory controller, to work independently. In order to facilitate this, the crossbar module provides address translation between global memory addresses and the addresses within a partition, in addition to routing requests to and from the partitions. In practice, this means that a partitioned memory system can be formed by a simple rearrangement of modules to communicate through this chip-wide crossbar. Additionally, since these partitions are simulated as separate modules they benefit from the same parallelism as the rest of the simulator.

4.3. Memory Hierarchy

An accurate model of a complex memory system is one of the most important components of Arches. Since ray tracing and many other high-performance workloads tend to be memory-bound, an accurate and flexible memory system is crucial to identifying bottlenecks so that we can iteratively improve the architecture design.

Like our compute model, the memory system is designed to be highly modular and configurable, allowing us to experiment with various configurations. The core building blocks of most memory systems are caches and DRAM (Dynamic Random Access Memory). To this end, the simulator implements configurable modules that simulate each of these components. The register file description, another critical part of the memory system, is contained within the processor core module.

These modules can be connected in different ways to form various memory hierarchies. For example, a cache hierarchy can be constructed by connecting multiple levels of caches with different sizes, organizations, replacement policies, features, latencies, and bandwidths.

Each of the cache and DRAM modules are highly configurable. The cache module models a set-associative cache that supports configurable associativity, latency, and banking, as well as several write, allocation, and replacement policies. In addition, we allow for independent allocation and fill granularity, and implement configurable miss status holding registers (MSHRs), which play a key role in high-performance memory systems by allowing caches to track multiple outstanding misses without stalling.

One of the most important and complicated parts of the memory system is DRAM. The behavior of DRAM systems is complex, and the latency and bandwidth characteristics of these systems are highly dependent on both their configuration and the data access pattern at run time. To accurately model this behavior, we choose to integrate Ramulator [LTB*24], an open source cycle-level DRAM simulator. Ramulator supports various DRAM configurations, including different DDR, GDDR, and HBM standards as well as internal behaviors of DRAM such as bank sizes and configurations, the difference in latency based on open vs. closed DRAM pages, and row buffer size. To allow Ramulator to interface with the rest of our simulator, we implement a DRAM wrapper module that converts incoming requests to Ramulator's memory requests. Then, we use Ramulator to model the latency of the memory request, while the corresponding data for the request is filled and then returned to the requesting unit on the cycle that the request completes. Because the DRAM model is integrated in Arches as its own module, it inherits all of the modularity and multi-threading benefits of Arches. Additionally, we allow the DRAM model to be clocked at an arbitrary clock ratio with respect to the processor clock in order to simulate differing memory and core clocks.

4.4. Special-Purpose Units for Ray Tracing

The modular design of the simulator also makes it easy to add special-purpose hardware. In order to simulate a ray-tracing architecture, we have implemented a fixed-function ray traversal (RT) core module.

This RT core performs bounding volume hierarchy (BVH) traversal and triangle intersection. Our RT core module is implemented using C++ templates to support different types of BVHs, node formats, and primitive encodings. It works by managing the traversal state of each ray in hardware and issuing memory requests to advance the ray traversal. Each request corresponds to a traversal step, loading either a node or primitive data from the memory

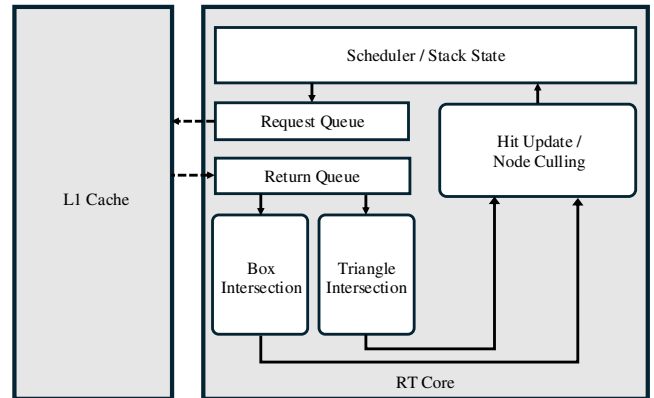


Figure 3: The general structure and workflow of our RT core implementation, connected to an L1 cache. It includes a ray scheduler fed by a custom instruction, request and return queues that manage L1 communication, box and triangle intersections, and a specialized logic for handling hit updates and culling.

hierarchy. Since multiple rays can be resident in the traversal core at a given time, these memory requests can be queued up in the memory system to hide latency and keep the memory system fed with requests.

Figure 3 shows the general structure and workflow of our RT core. At each cycle, an RT core attempts to schedule work by selecting a ray in a first-come-first-serve (FCFS) manner and popping the next node or primitive from the ray's traversal stack. When the data for a given traversal step returns from the memory system, the RT core sends the data to the corresponding box or primitive intersection pipeline along with the ray data. Once the intersections are evaluated, the traversal state of the ray is updated, and the ray is appended to the scheduling queue for the next traversal step.

A ray's traversal state is represented as a short stack backed by a restart trail [Lai10]. The restart trail stores the part of the tree that has been visited, and the short stack stores the top entries. When the stack runs out of space, we restart traversal again from the root, masking off the part of the tree that is already visited, using the restart trail. This reduces the state required by each ray significantly, allowing for processing more rays with less area.

A ray traversal is initiated by a processor core, using a custom instruction we added to the RISC-V ISA. Once a ray's traversal completes, the hit record is returned to the originating processor core. Adding these RT cores to an existing architecture is as easy as attaching the RT core module to a port in the memory hierarchy, adding an instruction to pass a ray to the RT core module, and updating the instruction table to send those ray instructions to the RT core at run time.

5. Programming, Execution, and Tools

Arches simulates the execution of user software on the specified hardware architecture. The hardware architecture is defined in C++ by creating the necessary modules (Section 3.1) and connecting

them as needed through our interconnection and arbitration modules (Section 3.2). Custom modules with arbitrary functionalities and connections can also be added, since the execution mode of Arches does not need to know what any module does, but only how they are connected.

The user software can also be written in C++. Since we have implemented a general-purpose processor core module that can load and execute RISC-V instructions, the user software can be compiled using this instruction set and the simulator can directly run this compiled executable. More specifically, we use RISC-V ELF (executable linkable format) for the user software. Therefore, we can utilize all the tooling available for RISC-V, including relatively mature C/C++ compilers of the GNU toolchain. Thus, programming the hardware to be simulated is as easy as writing C or C++ code and cross-compiling to target RISC-V.

This process for developing the user software has several benefits. First of all, user software can be easily debugged by compiling it natively and using a debugger on the CPU, instead of running the software on Arches. Additionally, RISC-V makes it easy to extend the ISA to support custom hardware. RISC-V has a relatively simple encoding and includes reserved opcode space specifically for architecture-specific instructions. Once an opcode encoding is defined, instructions can be added to the GNU RISC-V assembler and invoked from C/C++ code using inline assembly. In the simulator, these instructions can be decoded by adding a new instruction implementation to the corresponding opcode in the instruction table. The custom instruction can then be invoked from the code running on the simulated hardware through inline assembly. When compiling the user software natively for debugging purposes, the custom instructions are automatically replaced by function calls and the corresponding software implementation can handle the necessary functionality while running the user software natively on the CPU.

The execution of Arches begins and ends with the user software. During the initialization of the simulation, the binary for the user software is copied to a specified memory location in DRAM and the program counters of all processor cores are initialized to the entry point. Thus, the simulation begins by directly executing the user software.

During the course of the simulation, Arches periodically updates the output text-based log files that record the statistics captured during the simulation, enabling us to track the progress of the simulation. This is particularly helpful for early detection of any configuration errors or terminating tests that clearly perform below expectations. When all processor modules reach the end of the user program, Arches terminates and outputs the final statistics to the log files.

These log files can then be parsed with our python scripts to produce visualizations of the desired statistics, such as bandwidth usage and cache hit rates over time. This can be crucial for understanding the behavior of the simulated architecture during the course of an entire simulation, such as rendering a frame to completion, where bottlenecks can shift over time.

Table 1: Simulated NVIDIA RTX 2080 Hardware configuration.

TPs	64	NVIDIA
TMs	46	NVIDIA
Clock Rate	1515 MHz	NVIDIA
Memory Partitions	8	NVIDIA
Memory Type	GDDR6 14 GT/s	NVIDIA
Controller Latency	100 cycles	Vulkan-Sim
L2 Size	4MiB	NVIDIA
L2 Assoc	16-way	Vulkan-Sim
L2 Slices	32	Vulkan-Sim
L2 Latency	160 cycles	Vulkan-Sim
L1 Size	64KiB	NVIDIA
L1 Assoc	32-way	guess
L1 Banks	4	NVIDIA
L1 Latency	20 cycles	Vulkan-Sim
Line size	128B	NVIDIA
Fill granularity	32B	NVIDIA
RT Cores	46	NVIDIA
Max Rays	64	guess
BVH Build	Top down SBVH	guess
Node Format	Compressed	Vulkan-Sim
	Wide BVH6 (64B)	
Triangle Format	Uncompressed (64B)	Vulkan-Sim
Stack	Short Stack / Restart Trail	Vulkan-Sim
Node Pipe Latency	3 cycles	guess
Triangle Pipe Latency	22 cycles	guess

6. Case Studies

We have implemented and simulated different hardware architectures for ray tracing using Arches. These include a version of actual hardware that mimics the ray tracing components of NVIDIA's RTX 2080 [gtx] and different ray tracing hardware architectures proposed in prior research work such as, TRaX [SKKB09], STRaTA [KSS*13], and Dual Streaming [SGK*17]. In addition, we show the interoperability of these architectures by combining the ray scheduling units from STRaTA and Dual-Streaming with the traversal cores and the memory system from the RTX 2080.

6.1. Comparisons to Actual Hardware

To validate the simulator, we compare it against actual hardware. This is a challenging task, because most details of actual hardware are not publicly disclosed, and reverse-engineering hardware is out of scope for this work.

Fortunately, Vulkan-Sim [SCL*22] provides a configuration for NVIDIA Turing GPU architecture [tur18], though it is unlikely to match the actual hardware exactly (as also indicated by our tests). We use this configuration with the specifications of NVIDIA RTX 2080, forming the set of parameters shown in Table 1. Notice that while a few of them are from the official specifications, we have also guessed some parameters, such as the associativity of the L1 caches (which is not supported in Vulkan-Sim) and the configuration details of the RT cores.

Table 4: L2 Bandwidth measurements on an NVIDIA RTX 2080 and simulation results with Arches.

		Primary		Secondary		Tertiary	
		Value	Ratio	Value	Ratio	Value	Ratio
Crytek Sponza	RTX 2080	9.4%	-	42.2%	-	51.4%	-
	Arches	12.9%	1.37	49.0%	1.16	52.5%	1.02
Intel Sponza	RTX 2080	13.9%	-	37.8%	-	39.5%	-
	Arches	16.5%	1.19	38.5%	1.02	39.5%	1.00
San Miguel	RTX 2080	15.2%	-	29.7%	-	28.0%	-
	Arches	33.0%	2.17	39.3%	1.32	34.0%	1.22

Table 5: DRAM Bandwidth measurements on an NVIDIA RTX 2080 and simulation results with Arches.

		Primary		Secondary		Tertiary	
		Value	Ratio	Value	Ratio	Value	Ratio
Crytek Sponza	RTX 2080	20.4%	-	21.4%	-	20.4%	-
	Arches	23.3%	1.14	33.6%	1.57	37.1%	1.82
Intel Sponza	RTX 2080	32.5%	-	41.3%	-	42.5%	-
	Arches	33.5%	1.03	38.6%	0.93	39.5%	0.93
San Miguel	RTX 2080	31.1%	-	39.0%	-	40.2%	-
	Arches	34.0%	1.09	40.7%	1.04	41.7%	1.04

In Table 4 and Table 5, we show the bandwidth numbers, as percentages of maximum bandwidth, for the L2 cache and the DRAM. These are also elevated, as compared to real hardware. This indicates that the simulated hardware is driving more total L2 traffic than real hardware, but the DRAM numbers indicate that this traffic is absorbed by the L2 cache in all cases except for the secondary and tertiary rays in the Crytek Sponza scene. This seems to indicate differing behavior on incoherent rays with shallow traversal depths, pointing to differences in stack state management. With primary rays, we see a very similar DRAM traffic. Since primary rays are unlikely to cause enough cache pressure to evict nodes and primitives before last use, this indicates the size of BVH is in the correct ballpark. Additionally, the extra traffic from secondary rays in the San Miguel scene is absorbed entirely by the L2 cache, indicating more access to the BVH nodes, which is possibly due to a higher number of restarts on deep traversal paths.

Overall, despite the differences between the actual and simulated hardware and data structures, we can see that Arches can provide reasonably close estimates to actual hardware. The unknowns in these tests prohibit validating the implementation of all modules in Arches. However, all results are still relatively close to actual hardware, validating that the overall simulation system is able to provide a reasonable estimate, even in the presence of variations in detail.

6.2. Ray Tracing Architecture Research Experiments

The modular structure of Arches makes experimenting with different hardware configurations easier. We demonstrate this with three different variants of the TRaX [SKKB09] ray tracing hardware architecture shown in Table 6 with ray tracing performance numbers (in million rays per second) for the Intel Sponza scene.

Table 6: Million rays per second results of simulations with different versions of the TRaX architecture in the Intel Sponza scene.

	Primary	Secondary	Tertiary
TRaX with Software Traversal	625	463	353
TRaX with RT Cores	1026	576	365
TRaX with RT Cores & Compression	2092	828	605

The first version of TRaX handles ray traversal in software, as in the original work [SKKB09]. The second version replaces the software traversal with RT cores, configured similar to the experiments above. As one would expect, the RT cores provide a substantial performance improvement for primary rays that exhibit a relatively coherent memory access pattern. Yet, its improvement on secondary rays is relatively modest and almost non-existent for tertiary rays, as the ray traversal becomes more memory bound with the less coherent memory access these ray distributions produce. The third version of TRaX uses slightly different RT cores that can handle BVH compression. This reduces the scene data size and offers a significant performance boost across all ray distributions.

While the details of this experiment are out of the scope of this paper, these experiments illustrate the ease with which these architecture models can be modified. More specifically, introducing RT cores involves attaching RT core modules to the processor core modules and the memory hierarchy, adding a ray traversal instruction to the ISA, and replacing the software traversal in the user code with this instruction. Introducing BVH compression was even easier, as our RT core implementation supports various node and primitive formats.

We have also used the same RT cores for augmenting the STRaTA [KSS*13] and dual streaming [SGK*17] architectures. The latency hiding and compression capabilities provided by these traversal cores provided a significant boost in performance but revealed new bottlenecks in the process. Specifically, the ray stream traffic for the dual streaming architecture was shown to incur a significant cost which is not constant during the course of rendering a frame. As can be seen in Figure 6, writing rays for the ray stream has a significant cost earlier in rendering, but, then, this cost almost disappears. Reading the rays for the ray stream, however, gradually increases and then decreases near the end of rendering. Looking at the total DRAM traffic, we can see that the DRAM writes are dominated by writing the rays, except for at the very end of the frame, when the frame buffer is written. DRAM reads, on the other hand, are also padded by the scene data traffic, which is more significant in the very beginning and near the end in this test. The time-varying data we can export from Arches allows us to produce such graphs and understand the shifting costs of different operations during rendering.

6.3. Simulation Performance

The performance of a cycle-based simulator can be measured by the number of cycles it simulates per second. Obviously, this depends on not only the complexity of the hardware architecture that is being simulated but also the performance of the computer that

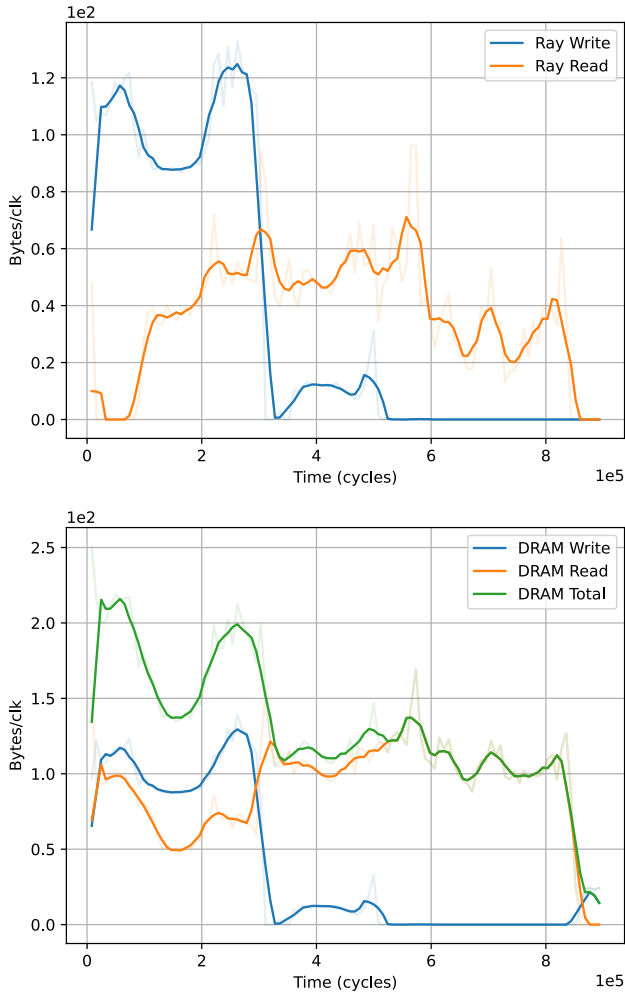


Figure 6: The ray stream traffic and total dram read traffic over time for dual streaming. We see an initial spike in write traffic from filling the ray queues and then sustained read traffic from draining them. A large portion of the total DRAM traffic is occupied by the ray streams.

runs the simulation. For our performance results we used an Intel i9-13900K CPU at clocked at 5.80GHz with 24 cores (32 threads).

As can be seen in Table 7, while simulating our approximated version of the NVIDIA RTX 2080 architecture, arches achieved an average of 19,380 cycles per second using 32 threads. This simulation speed does not depend on the rendered scene or the ray distribution, but does depend on the number and types of modules simulated.

In comparison, Vulkan-Sim achieved an average of 273 cycles per second while simulating the same architecture. Thus, Arches achieves about 71 times faster simulations than Vulkan-Sim for this architecture.

Table 7: Simulated cycles per second when simulating the approximate NVIDIA RTX 2080 architecture on arches and vulkan-sim using various numbers of threads.

Simulation threads	1	2	4	8	16	32
Arches	2,590	5,380	8,810	12,650	16,130	19,380
Vulkan Sim	273	-	-	-	-	-

7. Conclusion

We have presented Arches, a new cycle-level hardware simulation framework, designed for efficiently handling massively parallel ray tracing architectures. Arches uses a novel two-phase computation for simulating each cycle, allowing parallel simulation without data hazards. Its modular design simplifies the process of making substantial changes to a hardware architecture, allowing quicker iterations for research explorations. Its integration with the GNU toolchain allows writing software to run on the simulated hardware using C++, including custom instructions for controlling custom hardware units, and permits debugging natively on the CPU. Its comprehensive instrumentation provides detailed statistics that can be aggregated or turned into a time-varying sequence for deeper exploration of the results.

Our comparisons to actual GPU hardware shows that Arches produces reliable results while also deliver superior performance, as compared to the closest alternative for simulating fixed function ray tracing hardware.

References

- [APX13] ARNAU J.-M., PARCERISA J.-M., XEKALAKIS P.: Teapot: a toolset for evaluating performance, power and image quality on mobile graphics systems. ICS '13, Association for Computing Machinery, p. 37–46. URL: <https://doi.org/10.1145/2464996.2464999>, doi:10.1145/2464996.2464999. 2
- [AR13a] ARDESTANI E. K., RENAULT J.: Esesc: A fast multicore simulator using time-based sampling. HPCA '13, IEEE Computer Society, p. 448–459. URL: <https://doi.org/10.1109/HPCA.2013.6522340>, doi:10.1109/HPCA.2013.6522340. 2
- [AR13b] ARDESTANI E. K., RENAULT J.: Esesc: A fast multicore simulator using time-based sampling. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)* (USA, 2013), HPCA '13, IEEE Computer Society, p. 448–459. URL: <https://doi.org/10.1109/HPCA.2013.6522340>, doi:10.1109/HPCA.2013.6522340. 2
- [BA97] BURGER D., AUSTIN T. M.: The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News* 25, 3 (June 1997), 13–25. URL: <http://www.simplescalar.com>, doi:10.1145/268806.268810. 2
- [BBB*11] BINKERT N., BECKMANN B., BLACK G., REINHARDT S. K., SAIDI A., BASU A., HESTNESS J., HOWER D. R., KRISHNA T., SARDASHTI S., SEN R., SEWELL K., SHOAB M., VAISH N., HILL M. D., WOOD D. A.: The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. URL: <https://doi.org/10.1145/2024716.2024718>, doi:10.1145/2024716.2024718. 2
- [CBS*12] CHATTERJEE N., BALASUBRAMONIAN R., SHEVGOOR M., PUGSLEY S. H., UDIPI A. N., SHAFIEE A., SUDAN K., AWASTHI M., CHISHTI Z. A.: Usimm: the utah simulated memory module. URL: <https://api.semanticscholar.org/CorpusID:14712582>. 2

- [CNYS*14] CHIDAMBARAM NACHIAPPAN N., YEDLAPALLI P., SOUNDARARAJAN N., KANDEMIR M. T., SIVASUBRAMANIAM A., DAS C. R.: Gemdroid: a framework to evaluate mobile platforms. *SIGMETRICS Perform. Eval. Rev.* 42, 1 (June 2014), 355–366. URL: <https://doi.org/10.1145/2637364.2591973>, doi:10.1145/2637364.2591973. 2
- [dBGR*06] DEL BARRIO V., GONZALEZ C., ROCA J., FERNANDEZ A., E E.: Attila: a cycle-level execution-driven simulator for modern gpu architectures. In *2006 IEEE International Symposium on Performance Analysis of Systems and Software* (2006), pp. 231–241. doi:10.1109/ISPASS.2006.1620807. 2
- [GA19] GUBRAN A. A., AAMODT T. M.: Emerald: Graphics modeling for soc systems. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)* (2019), pp. 169–182. 2
- [gtx] <https://www.nvidia.com/en-gb/geforce/20-series/>. 7
- [KSAR20] KHAIRY M., SHEN Z., AAMODT T. M., ROGERS T. G.: Accel-sim: An extensible simulation framework for validated gpu modeling. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (2020), pp. 473–486. doi:10.1109/ISCA45697.2020.00047. 2
- [KSS*13] KOPTA D., SHKURKO K., SPJUT J., BRUNVAND E., DAVIS A.: An energy and bandwidth efficient ray tracing architecture. In *Proceedings of the 5th High-Performance Graphics Conference* (New York, NY, USA, 2013), HPG '13, ACM, pp. 121–128. URL: <http://doi.acm.org/10.1145/2492045.2492058>, doi:10.1145/2492045.2492058. 7, 9
- [Lai10] LAINE S.: Restart trail for stackless bvh traversal. In *Proceedings of the Conference on High Performance Graphics* (Goslar, DEU, 2010), HPG '10, Eurographics Association, p. 107–111. 6
- [LTB*24] LUO H., TUGRUL Y. C., BOSTANC F. N., OLGUN A., YAGLK A. G., MUTLU O.: Ramulator 2.0: A modern, modular, and extensible dram simulator. *IEEE Computer Architecture Letters* 23, 01 (Jan. 2024), 112–116. URL: <https://doi.ieeecomputersociety.org/10.1109/LCA.2023.3333759>, doi:10.1109/LCA.2023.3333759. 2, 6
- [LYR*20] LI S., YANG Z., REDDY D., SRIVASTAVA A., JACOB B.: Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters* 19, 2 (2020), 106–109. doi:10.1109/LCA.2020.2973991. 2
- [PHO*15] POWER J., HESTNESS J., ORR M. S., HILL M. D., WOOD D. A.: gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters* 14, 1 (2015), 34–36. doi:10.1109/LCA.2014.2299539. 2
- [SCHBS17] SEMBRANT A., CARLSON T. E., HAGERSTEN E., BLACK-SCHAFER D.: A graphics tracing framework for exploring cpu+gpu memory systems. In *2017 IEEE International Symposium on Workload Characterization (IISWC)* (2017), pp. 54–65. doi:10.1109/IISWC.2017.8167756. 2
- [SCL*22] SAED M., CHOU Y. H., LIU L., NOWICKI T., AAMODT T. M.: Vulkan-sim: A gpu architecture simulator for ray tracing. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2022), pp. 263–281. doi:10.1109/MICRO56248.2022.00027. 2, 3, 7
- [SGB*18] SHKURKO K., GRANT T., BRUNVAND E., KOPTA D., SPJUT J., VASIOU E., MALLET I., YUKSEL C.: SimTRaX: Simulation infrastructure for exploring thousands of cores. In *Proceedings of the 2018 Great Lakes Symposium on VLSI* (New York, NY, USA, 2018), GLSVLSI '18, Association for Computing Machinery, p. 503–506. URL: <https://doi.org/10.1145/3194554.3194650>, doi:10.1145/3194554.3194650. 2, 3
- [SGK*17] SHKURKO K., GRANT T., KOPTA D., MALLET A., YUKSEL C., BRUNVAND E.: Dual streaming for hardware-accelerated ray tracing. In *Proceedings of High Performance Graphics* (New York, NY, USA, 2017), HPG '17, ACM, pp. 12:1–12:11. URL: <http://doi.acm.org/10.1145/3105762.3105771>, doi:10.1145/3105762.3105771. 7, 9
- [SKKB09] SPJUT J., KENSLE A., KOPTA D., BRUNVAND E.: Trax: A multicore hardware architecture for real-time ray tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28, 12 (2009), 1802–1815. doi:10.1109/TCAD.2009.2028981. 7, 9
- [SLS04] SHEAFFER J. W., LUEBKE D., SKADRON K.: A flexible simulation framework for graphics architectures. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (New York, NY, USA, 2004), HWWS '04, Association for Computing Machinery, p. 85–94. URL: <https://doi.org/10.1145/1058129.1058142>, doi:10.1145/1058129.1058142. 2
- [SXS*16] SHAO Y. S., XI S. L., SRINIVASAN V., WEI G.-Y., BROOKS D.: Co-designing accelerators and soc interfaces using gem5-aladdin. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016), pp. 1–12. doi:10.1109/MICRO.2016.7783751. 2
- [TSS*23] TINE B., SAXENA V., SRIVATSAN S., SIMPSON J. R., ALZAMMAR F., COOPER L., KIM H.: Skybox: Open-source graphic rendering on programmable risc-v gpus. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (New York, NY, USA, 2023), ASPLOS 2023, Association for Computing Machinery, p. 616–630. URL: <https://doi.org/10.1145/3582016.3582024>, doi:10.1145/3582016.3582024. 2
- [tur18] NVIDIA Turing GPU Architecture. Tech. Rep. WP-09183-001_v01, NVIDIA, 2018. 7