15–418 – Final Project Report

THEO KROENING and ETHAN LU*, Carnegie Mellon University, USA

1 SUMMARY

We extended Professor Railing's CADSS framework to model computation on GPUs. By employing a modular and configurable design, we were able to study the impact of different hardware configurations and program characteristics on performance, including block scheduling, warp scheduling, barriers, and thread divergence.

2 BACKGROUND

Parallel programming models for GPU's like NVIDIA's CUDA provide a high level abstraction for parallel programming. Though programmers explicitly define the "layout" for computation by assigning threads to blocks and blocks to grids, most details about how instructions are scheduled onto hardware remain opaque by design. Programmers cannot make assumptions about how threads are grouped into warps, how they are scheduled, or how they handle divergent control flow. Because these details are proprietary, many are also not exposed by profiling tools like NVIDIA's Nsight. However, these are all details that can significantly affect program performance, so it is important to be able to reason about them at a high level.

The goal of our project, then, is to gain insights into how different designs — both in terms of the hardware layout and the mapping of work onto the hardware — impact the performance of programs. In particular, our project will investigate the following questions:

(1) **How does performance scale across multiple processors?** The "compute" area of GPUs is typically divided into several processors, each of which interleaves the computation of warps within a thread block. We will refer to these basic units as *streaming multiprocessors* throughout this report as this is the terminology used by NVIDIA [1]. We will investigate how performance scales as we run the same amount of work on one streaming multiprocessor versus when we do so on multiple.
(2) **How does inter-warp parallelism affect performance?** Modern GPUs can interleave the instruction streams of several warps to tolerate pipeline and memory stalls [1]. We will investigate to what extent this capability allows performance to scale within a single multiprocessor as we add more warps.
(3) **How does synchroniztion affect performance?** *Barrier* instructions, for example CUDA's `__syncthreads()`, cause a warp's instruction stream to stall until all threads in the block have reached the specified program point. We will investigate how these instructions affect performance.
(4) **How does thread divergence affect performance?** Modern GPUs execute instructions in a *SIMT* (single instruction, multiple thread) fashion. So, threads are grouped into "warps" (the NVIDIA term [1]) that must execute in lockstep. Divergent control flow within warps affects performance because not all threads can be actively performing computation, which limits our ability to exploit a GPU's parallelism. In this paper we will explore how a simulator can be used to measure thread divergence for different programs, and investigate the impact of different per-warp scheduling techniques on occupancy and performance.

Authors' address: Theo Kroening, tkroenin@andrew.cmu.edu; Ethan Lu, eblu@andrew.cmu.edu, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

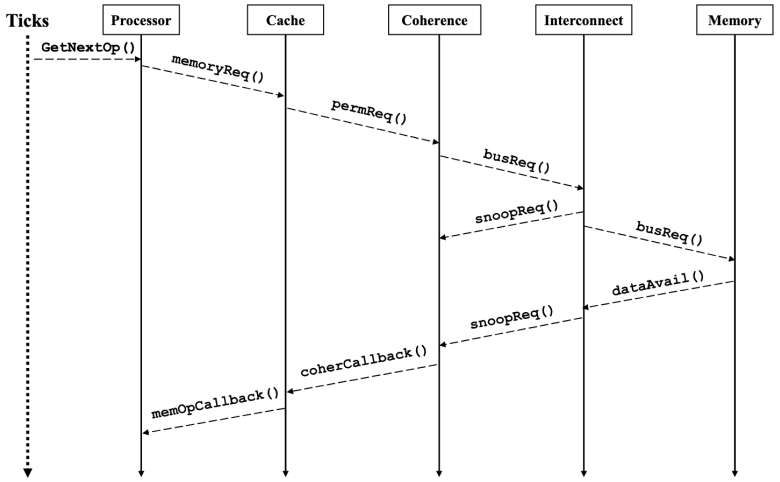


Fig. 1. Architecture of the original CADSS Simulation framework [6].

Taken together, we believe that these questions cover most of the key mechanisms GPUs use to achieve parallelism. We therefore hope to offer answers to each question through a series of simulation studies.

3 APPROACH

3.1 Programming Framework

Rather than implement a new computer architecture simulation framework from scratch, we based our simulator on **CADSS** (**C**omputer **A**rchitecture **D**esign **S**imulator for **S**tudents). CADSS is a trace-driven, open source simulator framework built by Professor Railing for the introductory computer architecture course at CMU (15–346) [6]. In 15–346, students put together a simulator that integrates multiple components (cache, memory, interconnect, processor, branch predictor, etc.) to enable unified simulation across a range of modular components.

Figure 1 shows the architecture of the original simulator. As the diagram shows, a key feature is that it integrates multiple components that affect the performance of computation. For our project, we replace the the `processor` component with a `gpu` component. From there, we retain only the `memory` component. All other components (cache, coherence, interconnect) are effectively stripped out because they are out of scope for this project.

CADSS is a CMake project primarily written in C, but with support for extensions written in C++. We wrote all of our extensions in C++. So, our project has no specific target machine. Note that if shared objects are used, then they should be recompiled for the target machine.

3.2 Our Extensions

In Section 2 we discussed the goals of our project. We will now discuss how our extensions to CADSS help us tackle each question. Recall that we wanted to (a) measure how performance scales with parallelism across warps within SMs and across SMs, (b) quantify the effects of synchronization, and (c) study the effects of thread divergence and workload imbalance within warps.

To effectively investigate each component, our goal was to isolate each component and hold all other variables constant. So, for components (a) and (b), we do not model any thread divergence or workload imbalance. The instruction streams executed by each warp are therefore identical. Note

that this setup does *not* require the simulator to perform computation (i.e. actually execute the instructions on real data) — because we are only interested in total runtime, trace files need only to specify the types of instruction being executed in addition to the distribution of memory accesses. This matches the setup used in the original CADSS framework.

However, this trace file format is not suitable for investigating component (c). If the trace file does not specify different input data for each thread, then they cannot diverge and there cannot be any workload imbalance. One workaround would be to explicitly encode divergent instruction streams for each warp in the trace files. However, hardcoding the instruction stream for each warp like this would not allow us investigate the effects of different techniques for handling branches and thread divergence. So, we decided the simulator must support per-thread computation based on variable input data.

So, we have seen that the requirements for components (a) and (b) differ significantly for the requirements for component (c). Component (c) requires significant changes to the trace file format and simulator to support computation. Because (c) is only concerned with modeling thread divergence, the considerations for components (a) and (b) (memory latency, multiple SMs, multiple warps) are not relevant to (c). Similarly, (a) and (b) do not need to model thread divergence and can use a much simpler trace file format and simulator design.

We therefore decided to split our project into two simulators, each of which is based on a simple 5-stage in-order processor we developed at the beginning of this project:

- **Scaling Simulator.** This simulator assumes no divergence. We will use it to measure how performance scales across multiple SMs, multiple warps on an SM, and barriers. We also assume that memory is constant latency.
- **Divergence Simulator.** This simulator explicitly models computation to track thread divergence. We will use this to investigate the performance impacts of thread divergence, as well as the benefits of different methods of handling branches. This simulator assumes that all instructions have the same runtime, save for those that cause stalls due to data dependencies or control hazards.

We will discuss the design of each simulator in the subsections that follow.

3.3 Scaling Simulator Design

3.3.1 Overall design of single SM. . The overall GPU SM design of our simulator is a 6-stage pipelined 32 wide SIMD processor with warp scheduling. The SM contains a set number (configurable, 64 by default) of warp slots that can be used to store warp contexts and schedule warps. In other words, the SM is able to schedule up to 64 warps at a time by default, and any additional warps will be stored in a queue that is pulled from when one of the 64 current warps finishes executing.

The processor consists of the following stages, and we use queues to pass instructions from one stage to another stage. This is simplified and omits some implementation details.

- **Fetch**: In this stage, we use the warp scheduler to select the next warp we want to run and get it's instruction pointer. We also update the register scoreboard, which is used to prevent read after write conflicts at the fetch stage.
- **Decode**: This stage is currently has nothing inside it as nothing in decode can effect ticks in our model.
- **Execute**: Same as decode, we do not need to worry about this stage as we are implementing a software simulation.

- **memory_rising**: In this stage, we check to see if the current request is a memory load. If it is a load we need to kick off a memory operation and move the current instruction into a vector of entries waiting for memory data. If it is not a load, we pass it to the memory falling stage. One interesting feature in our memory is that it allows instructions following a load to retire earlier as long as there is no data hazard, meaning that our memory is non blocking.
- **memory_falling**: In this stage, we check to see if any waiting instruction has received data from memory. If any waiting instruction has received data, it will be added to a buffer which will pass it to the write back stage as soon as the pipeline stage is free.
- **write_back**: If a instruction reaches this stage, it is about to retire and commit to architectural state. Here, we update the register scoreboard, and we can successfully retire the instruction.

3.3.2 Warp scheduler. : We used a simple but seemingly very effective warp scheduler (see results) to schedule our warps. In summary, the scheduler will loop through all the warps inside the 64 warp slots of the SM and select the first one that can issue a instruction (ie is not stalled or waiting to resolve a conflict). This means that our warp scheduler will always prioritize the warp in the first slot inside the SM and will only schedule other warps in higher slots in the event that the warp in the first slot can not be scheduled. Ignoring barriers, we keep track of what can be scheduled by keeping a register scoreboard for each warp that indicates if we will get a conflict. Since our processor is in-order, the only conflict we need to worry about is read after write.

3.3.3 Barriers. : Our scaling simulator also supports barriers. This complicates our warp scheduler because barriers work very differently than just hazard detection as not only do all the warps currently residing inside the 64 slots of our SM need hit the barrier, we also need any additional warps in the waiting queue to also hit the barrier. Meaning that we will need to take a warps that hit a barrier (and therefore are stalling) out of the 64 slots in the SM and replace it with warps in the waiting queue that have not hit the barrier yet.

Since we are swapping warps, it adds an additional layer of complexity because we need to ensure that all instructions of a warp retires and commits to the architectural state (scoreboard) before we swap it. This can be done by stalling the warp until its scoreboard is clean, and then we issue the barrier instruction and swap it out for a warp in the waiting queue that hasn't hit the barrier yet. The stalling is one source of overhead introduced by barrier. The other source of overhead is that all warps are synchronized after a barrier, meaning that if one warp stalls due to a hazard right after the barrier, all warps after it will also stall immediately.

3.3.4 Multiple SMs. Our scaling simulator also supports multiple SMs. Each SM can only run 1 thread block at a time. We map blocks onto SMs dynamically by having each SM take one thread block during initialization. If an SM finishes computation, it can take another thread block as long as there are still thread blocks remaining (specified by `remainingBlocks` in `gpu.cpp`). Computation only finishes when there are no more remaining blocks and all SMs have finished their thread block.

3.4 Divergence Simulator Design

3.4.1 Trace File Format and Transpiler. To model the impacts of thread divergence and workload imbalance across a range of programs easily, we chose to have the divergence simulator perform computation — that is, actually execute the instructions that make up the kernels that it simulates. To achieve this, we modified the trace reader component to read a dialect of PTX rather than the

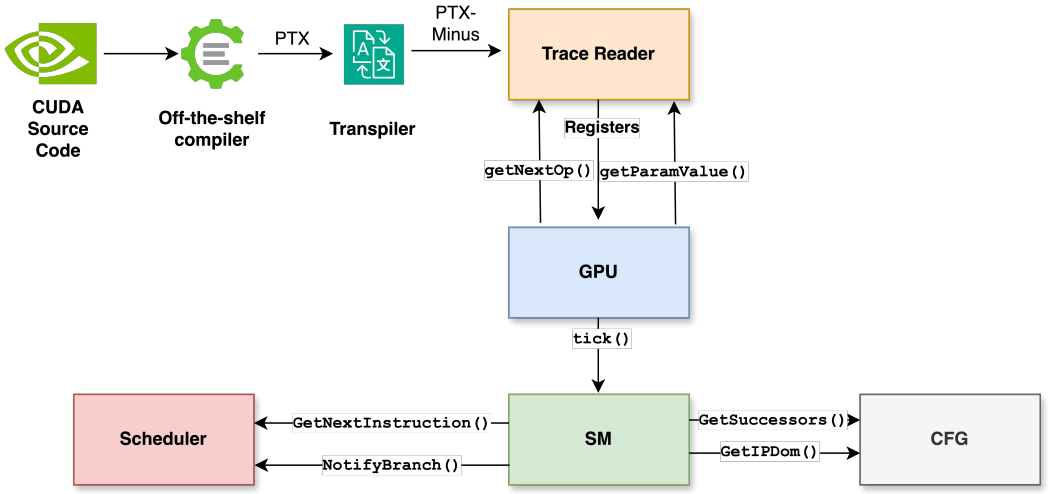


Fig. 2. Overview of the Divergence Simulator. An off-the-shelf compiler is used to generate PTX from C++ CUDA source code, then a transpiler translates the PTX into a more restricted language we call "PTX-Minus". To support different scheduling techniques, the `Scheduler` class is abstract and supports multiple implementations. The CFG is used to to get information about successors and dominators.

existing CADSS trace file format. Though PTX is not actually an ISA that executes on hardware, a similar approach is used by existing open source projects [3], and it is a reasonable enough proxy for our purposes.

Figure 2 shows the specifics of this approach. We use an off-the-shelf compiler to convert CUDA source code to PTX, then use a simple transpiler to a restricted subset of PTX. Inspired by the "PTXPlus" of GPGPUSim [3], we dub our language "PTX-Minus" because it implements a very limited set of instructions and has a very simple syntax. Appendix B shows some examples of transpiled programs.

Due to time constraints, we only guarantee coverage on the test programs bundled with this project. Still, this approach has the advantage that it is relatively easy to add new trace files: rather than trying to collect a trace from a real GPU, you simply compile CUDA source code, specify the program inputs, then put everything through the transpiler. If the simulator does not yet support an instruction, it is relatively easy to add support for it either by adding it explicitly or by breaking it down into simpler instructions. For example, the fused multiply add instruction (`mad`) is broken down into a multiply followed by an add for simplicity.

3.4.2 Computation. With this new trace file format in hand, computation is relatively straightforward. The trace reader reads the input data from the top of the trace file, `malloc`'s space for it, and passes the start addresses to the GPU. Instructions are then "executed" simply by interpreting them at runtime during the "write back" stage of our five-stage pipeline. Since the example programs only write to "global" memory and not shared memory, memory reads and writes are supported simply by reading from and writing to the buffers allocated by the trace reader. Because all the types are explicitly encoded in the trace file, the address computation in the compiled PTX code works out of the box for the programs covered in this report!

The key advantage of doing computation is that we know the exact values of the predicate registers for branch instructions, so we can track thread divergence. After going through a divergent

branch, a lane mask is attached to subsequent instructions by the *Scheduler* component. The masked lanes are then simply ignored for those instructions by the interpreter.

3.4.3 Scheduler. A key factor determining thread divergence is how the instruction scheduler handles divergent branches. To explore the performance of different implementations, the Scheduler provides an abstract base class, on top of which different schedulers can be built. Currently, we provide two schedulers: `NaiveScheduler` and `ReconvergenceScheduler`. Each only needs to implement the interface shown in Figure 2: the `GetNextInstruction()` method grabs the next instruction to be executed, and the `NotifyBranch()` method notifies the scheduler of a branch instruction and the new lane mask so that it can update its internal state.

We will now discuss each of the schedulers provided by our framework. The *Naive* scheduler handles divergent branches simply by pushing the if and else branches onto a stack, along with their corresponding lane masks. The top entry on the stack is then executed until it has completely finished executing. Then, we pop from the stack and resume execution from the new top entry. If all threads have finished, the simulation exits.

The *Reconvergence* scheduler implements the algorithm initially presented by Fung et al. [2], and reproduced by Aamodt et al. [1] in their textbook. Each stack entry holds three things: a reconvergence point, a `NextPC` and a lane mask. The algorithm then proceeds as follows:

- Execution starts with an entry where all warps are running.
- When a branch is encountered. Change the `NextPC` of the top entry to be the reconvergence point of the branch. Push two new entries with the same reconvergence point, but different `NextPC`'s and lane masks for the if and else branches.
- When execution hits the reconvergence point for the entry on the top of the stack, we pop from the stack and resume execution from the new top entry.

As described by [1], we choose the reconvergence point to be the *Immediate Post Dominator* of the basic block in the program's control flow graph. This is the functionality provided by the `CFG` module – it constructs a control flow graph from the program, and computes the immediate post dominator for each node by first computing the post dominators, then doing a BFS to find the nearest successor that is a post-dominator. The algorithms for post-dominators and immediate post-dominators were adapted from pseudocode provided in a lecture by Mahlke [5].

As an example, Figure 3 shows the control flow graph our simulator generates for the `collatz` example program. Note that the diamond starting with the node `LBB02` corresponds to the if/else block in the program's loop. When the reconvergence scheduler is notified of the branch there, it notices that the threads can reconverge at `LBB05`, since that is the immediate post-dominator of `LBB02`.

4 RESULTS

4.1 Scaling Simulator Studies

4.1.1 Relevant variables.

- **Memory load delay**: While we have used the API for memory accesses provided by CADSS, we have modified `cache.c` such that all memory loads take a constant amount of time (by default, 100 cycles). This can be configured inside `cache.c` by changing the `count` variable in `pendingRequest`.
- **Warp slots**: every SM has a fixed number of execution contexts to hold register files for warps (configurable, defaults to 64). Additional warps that do not fit into this space are buffered in a queue and are eventually inserted into a free warp slot once one frees up.

4.1.2 Warp scheduler analysis.

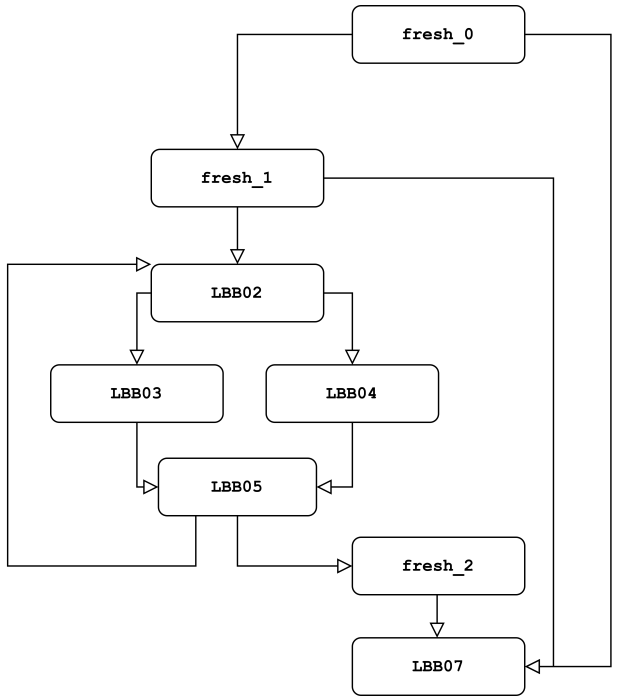


Fig. 3. Control Flow Graph generated from the `collatz` example program by our simulator.

- **Table 1**: `nostall` is a trace that consists only of ALU ops with no register dependencies between instructions. Hence, we never stall and the importance of warp scheduling is diminished. This means that we should expect the number of ticks elapsed to scale linearly with the number of threads, since only one warp can run at any given time.
- **Table 2**: Meanwhile, `stallMax` is the same trace as `nostall` but every ALU instruction has a register dependency on the previous instruction, meaning that we must stall at every instruction to avoid read-after-write hazards. Since we stall, warp scheduling can kick in and mask latency. We see this effect as there are minimal tick increases as we start increasing the number of threads. We also see that runtime begins to *increase* linearly once we reach 128 threads (4 warps). This is because read-after-write hazards only stall for 6 cycles in a 6 staged pipelined processor, meaning that we only need 6 warps at a time to fully hide latency and any more will not help.
- **Table 3**: `Mixed.trace` is a short trace of size 4 that contains an ALU op, a load, followed by another ALU op with a register dependency to the load, and ends with a store. Since memory loads take 100 cycles, we see that a single warp/thread should take over 100 ticks to complete. Without warp scheduling, we should expect the total number of ticks to double as we double the number of threads. However, with our warp scheduler, we see that ticks barely increase as we increase threads until we reach 2048 threads (64 warps). As we exceed 2048 threads (64 warps), we see a rapid increase in execution time because the GPU SM can only store the context of 64 warps at a time by default, meaning that any additional warps must wait for a current running warp to complete before they can run.
- **Table 4**: `HeavyStall.trace` is a trace that contains many load instructions followed by an ALU instrution with a register dependency to the load. So, the trace stalls a lot and

can therefore benefit significantly from warp scheduling. As we can see, there is a very minimal increase in ticks as we double the number of warps until we reach 2048 total threads (64 warps). After we exceed 64 warps, we have filled up all the warp slots on our SM and therefore remaining warps can only be run when more execution contexts become available.

- **Table 5**: Here, we try to run `mixed.trace` with 32×64 threads (64 warps) as we try to vary the number of warp slots/warp contexts a single SM can hold. As expected, as we increase the number of warp slots, our performance increases as we have more warps we can switch into when we encounter stalls. However, we see that there is no change between 64 and 256 slots because we have only 64 warps worth of threads.

Threads	Ticks
32	44
64	82
128	158
256	310
512	614
1024	1222

Table 1. A table showing how ticks change based on thread count when running nostall.trace on a single SM with 64 warp slots. We can disregard memory latency because nostall has no memory operations

Threads	Ticks
32	192
64	193
128	195
256	384
512	762
1024	1333

Table 2. A table showing how ticks change based on thread count when running stallMax.trace on a single SM with 64 warp slots. We can disregard memory latency because stallMax has no memory operations

4.1.3 Barrier analysis.

- **Table 6**: the `heavystall_barrier` trace modifies `HeavyStall` to include a barrier between every instruction. As we might expect, runtime and overhead compared to `HeavyStall` increase as we increase the number of threads. This makes sense because we need to ensure that all warps have reached the barrier before we can execute instructions past the barrier.
- **Trace of a simple barrier**: Appendix C contains a simple barrier trace as well as the simulation walkthrough of the barrier trace. Notice that the both warps are synchronized after we hit the barrier.

4.1.4 Multiple SMs.

- **Figure 4**: Here, we investigate how performance scales as we add additional streaming multiprocessors. Since every SM can only run 1 thread block at a time, we expect the number of ticks to be inversely proportional to the number of SM cores initially. Once we reach 16

Threads	Ticks
32	113
64	115
128	124
256	132
512	158
1024	205
2048	314
4096	570
8192	1082
16384	2106
32768	8250
65536	20442

Table 3. A table showing how ticks change based on thread count when running mixed.trace on a single SM with 64 warp slots. We assume that memory loads have a 100 tick latency.

Threads	Ticks
32	1627
128	1629
512	1633
2048	1753
8192	6683
32768	26159

Table 4. A table showing how ticks change based on thread count when running heavystall.trace on a single SM with 64 warp slots. We assume that memory loads have a 400 tick latency.

Warp Slots	Ticks
1	6854
4	1730
16	494
64	314
256	314

Table 5. A table showing how ticks change based on warp slots of a SM when running mixed.trace on a single SM when using 2048 threads (64 warps). We assume that memory loads have a 100 tick latency.

cores, however, we stop seeing a decrease in ticks as 16 thread blocks can only map to 16 cores at most; any extra cores will be idle.

4.2 Divergence Simulator Studies

We will base our analysis in this section on the results for two example programs, which we introduce here:

- **`binsearch`**: Given an array of search keys and a sorted array of elements to search through, try to find search keys in parallel using a single CUDA kernel. As was discussed in the relevant written assignment, a major issue in this kernel is thread divergence because the

Threads	Ticks
32	1651
128	1671
512	1906
2048	2698
8192	9298
32768	35650

Table 6. A table showing how ticks change based on thread count when running heavystall_barrier.trace on a single SM with 64 warp slots. We assume that memory loads have a 400 tick latency.

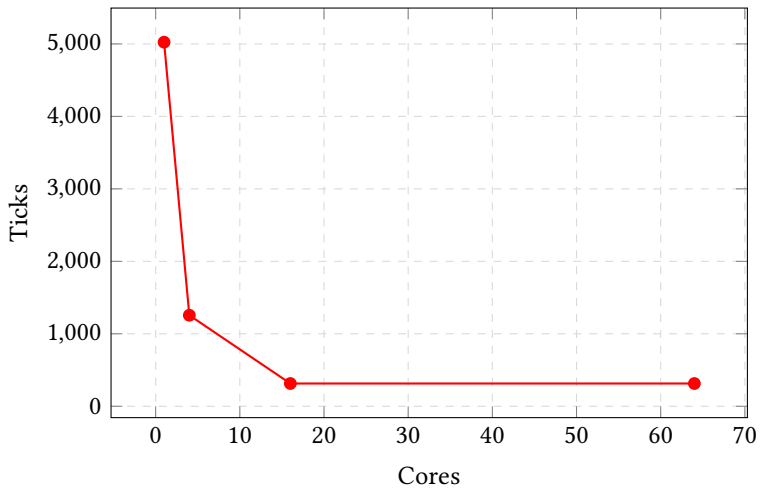


Fig. 4. Chart depicting ticks vs cores when we run 16 thread blocks with block size of 2048 threads (64 warps). Each SM has 64 warp slots and we assume memory loads have 100 ticks of latency.

elements in the search array are completely independent and could lie anywhere in the target array.

- **`collatz`**: The Collatz conjecture states that for any $n \geq 1$ you can repeatedly apply the following operation:

$$f(n) = \begin{cases} 3n+1 & \text{If } n \text{ is odd} \\ n/2 & \text{If } n \text{ is even} \end{cases}$$

  Then you will eventually reach 1. For our example program, we collect the first 32 numbers that require at most 20 iterations to converge to one, then sort them by ascending order of computational intensity.

The full CUDA C++ source code for each program is included in Appendix A.

4.2.1 Measuring Thread Divergence. One of the key goals of the divergence simulator was to quantify thread divergence within a single warp across a range of different programs. To demonstrate this capability of the simulator, we modify it to print the active lane mask for each instruction that it executes. This then gives us an accurate picture of how many threads are active during each point of the simulated program. We can visualize this output in two different ways:

(1) Total Active Threads over Time – at each tick, how many threads in the warp are active? This is just a single line where the x-axis is the instruction count and the y-axis is the total number of active threads.
(2) Heatmap. Here the x-axis is still instruction count, but the y-axis has 32 lanes, one for each thread in the warp. Each cell then is colored if the thread is active at that time step, and white otherwise. This format is useful for visualizing what specific threads are doing over the course of the program rather than the aggregate representation provided by the first format.

An example of the heatmap representation is shown in Figure 5. We can see there is a vast difference in thread divergence. We can see that the binary search kernel is very divergent because of the amount of "white" on the chart — for the majority of the program, the majority of threads are inactive. Meanwhile, the Collatz kernel suffers from divergence to a much lesser extent, since most of the chart is colored.

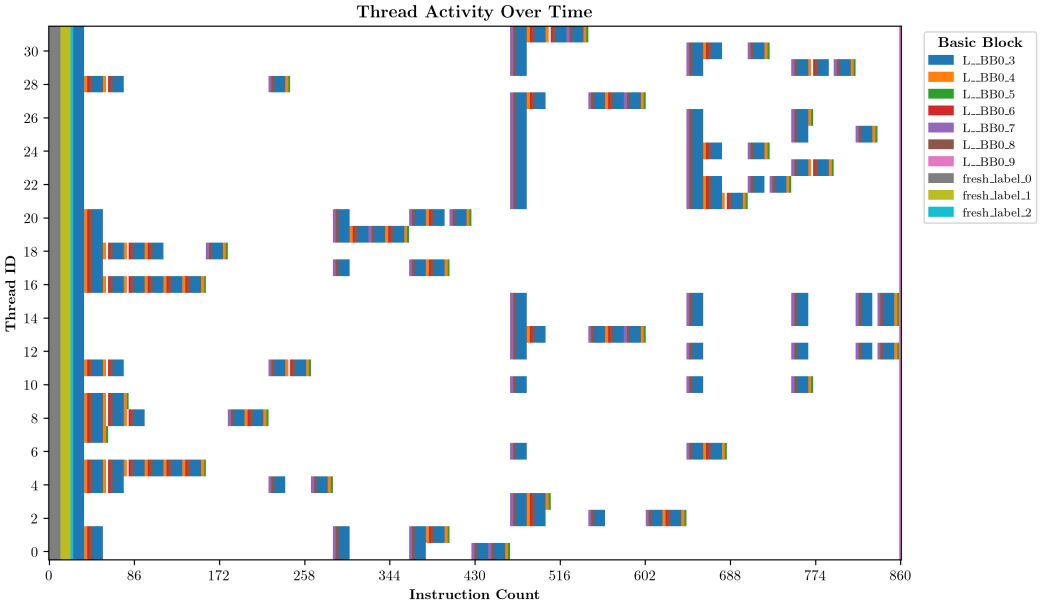
Since these results were collected using the reconvergence scheduler, the control flow graphs for each program can lend insight to why there is such a dramatic difference in divergence. Figure 6 shows the CFG for `binsearch`, and Figure 3 shows the CFG for `collatz`. As noted earlier, the CFG for Collatz has a reconvergence point at `LBB05` that allows all active threads to resume execution in lockstep together at the end of each loop iteration. To see this, note the colors in Figure 5. Each divergent section is marked by a strip of blue – this is exactly the if-else statement in the CFG (LBB02). This is followed by a strip of orange corresponding to the if branch (`LBB03`), followed by a *complementary* (non-overlapping) strip of green corresponding the else branch. The threads then converge at reconvergence point marked by a strip of red (`LBB05`). Meanwhile, the control flow graph generated by our simulator for `binsearch` is a mess. Here, the blocks corresponding to the loop are those below `LBB03`. Note that, since the algorithm is based on immediate post-dominators, there is no reconvergence point for the branches in the loop other than `LBB09`, which is the end of the program anyway. The group of active threads grows smaller and smaller until each individually terminates (marked by the small strip of green). The only time all threads come together again is at the end of the program (the return statement), marked by the strip of pink on the right of the chart.

Of course, another factor is the algorithmic properties of each program. In the naive binary search kernel, elements make it through the branch depending on whether they are smaller or greater than a common pivot value in the middle of the target array. Assuming the search keys are uniformly distributed, half of the threads will become inactive at each iteration. We can visually confirm this using the "total active threads" chart shown in Figure 7. Note how the number of active threads roughly halves at each step, followed by another spike, at which point the halving pattern continues again. The spike at the end of the program corresponds to the to the threads reconverging at the return statement (the pink strip at `LBB09` in Figure 5).

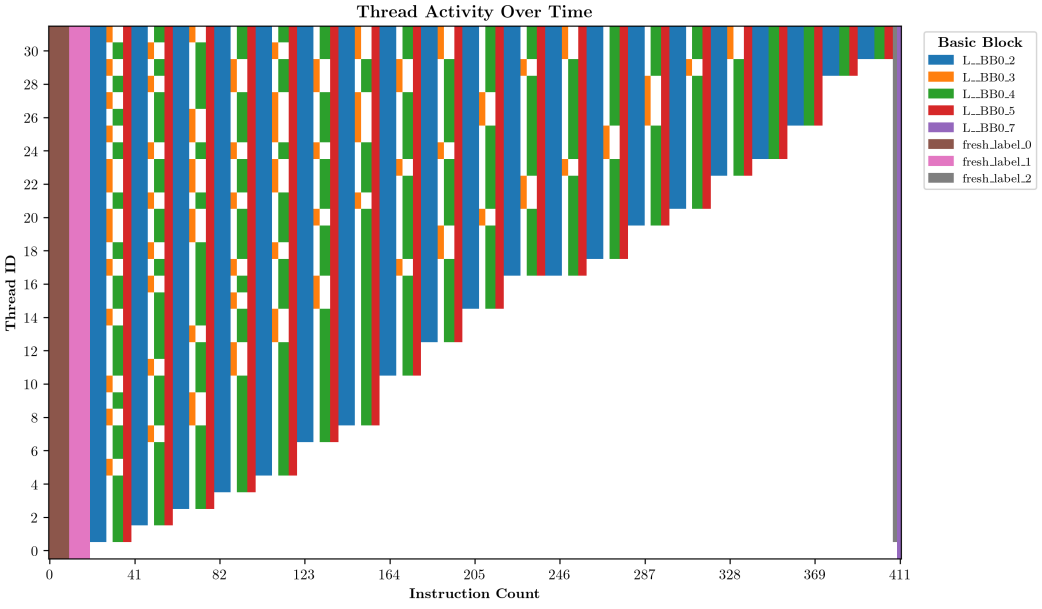
Meanwhile, `collatz` is explicitly written so that adjacent threads have similar amounts of work. The reason we see the total number of active threads decreasing over time in Figure 5 is because higher thread IDs have more difficult problems (more Collatz iterations) by construction, yielding the triangle shape we see on the chart.

4.2.2 Exploring Divergence Management Strategies. Our simulator design also allows us to explore different scheduling strategies. In particular, our simulator supports a *Reconvergence Scheduler* and a *Naive Scheduler* that does not attempt to look for reconvergence points (Section 3.4).

Figure 8 shows how thread activity varies over time using the naive scheduler using the heatmap representation. Comparing this to Figure 5 (the results for the Reconvergence Scheduler) yields interesting results. We see that the chart for `binsearch` is almost identical. The only difference is that the unified pink strip at which the threads reconverged has now been broken up and put



(a) `binsearch`



(b) `collatz`

Fig. 5. Heatmap representation for thread activity for one warp executing the `binsearch` and `collatz` example programs using the **Reconvergence Scheduler**. In each chart, the x-axis is instruction count, and the y-axis is the thread ID. A colored cell indicates that the thread is active for the relevant instruction, and a blank cell indicates that thread is inactive. The specific color of each colored cell corresponds to the basic block the instruction belongs to.

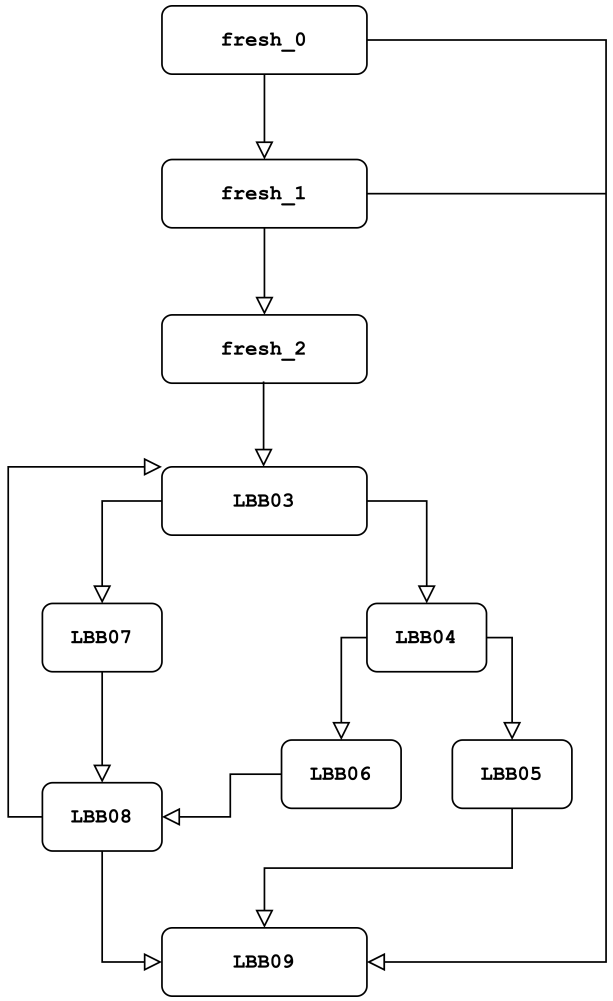


Fig. 6. Control Flow Graph generated by our simulator for the `binsearch` program.

at the end of the last bar for each thread. This result aligns with our expectations — as we noted, the CFG produced by our simulator does not allow any opportunities for reconvergence besides at the end of the program, so the per thread activity profile looks similar to the naive scheduler. Meanwhile, the results for `collatz` are dramtically different. From the amount of white on the chart, we can see that most threads are inactive for the majority of the program. Where threads *are* active, many of them serialize almost all of their computation into one fat bar. So, this is a case where the reconvergence scheduler clearly comes out on top because it is able to exploit the reconvergence point at `LBB05` (Figure 3).

A natural follow up question is: how do these different strategies affect performance? Figure 9 shows the runtime of each strategy on both traces, measured in simulator ticks. We can see that there is a minimal improvement for `binsearch`, but a dramatic improvement for `collatz`. This matches up with our analysis of the heatmap charts. We see a small improvement on `binsearch` because the threads are able to reconverge at the last basic block to execute it once altogether

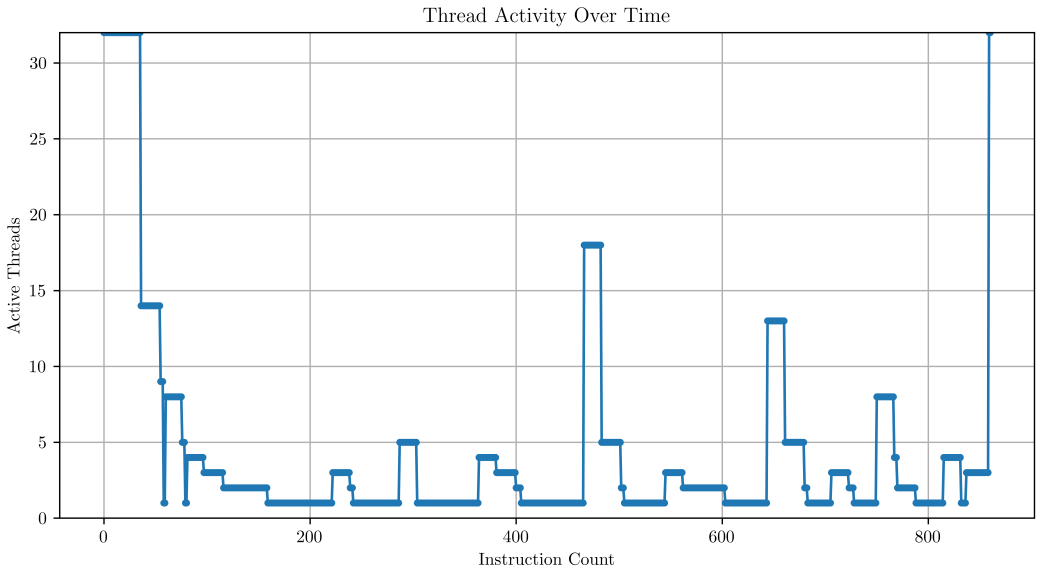


Fig. 7. Total Number of Active Threads over time for the `binsearch` example program. The x-axis represents the total number of instructions processed, and the y-axis represents the total number of active threads at each instruction.

rather than several times. However, the improvement is only small because there are no other opportunities for reconvergence. Meanwhile, the improvement on `collatz` is dramatic – 8259 vs 1009 ticks (over an 8× speedup!). This is because, without reconvergence, threads often execute instructions with few other threads, or on their own. Note that this also means the effects of costly instructions like branches (which stall the pipeline due to being control hazards) are magnified because they are executed more than they need to be. These charts therefore demonstrate that a scheduler that manages divergence effectively is crucial to performance for programs where there are opportunities for reconvergence.

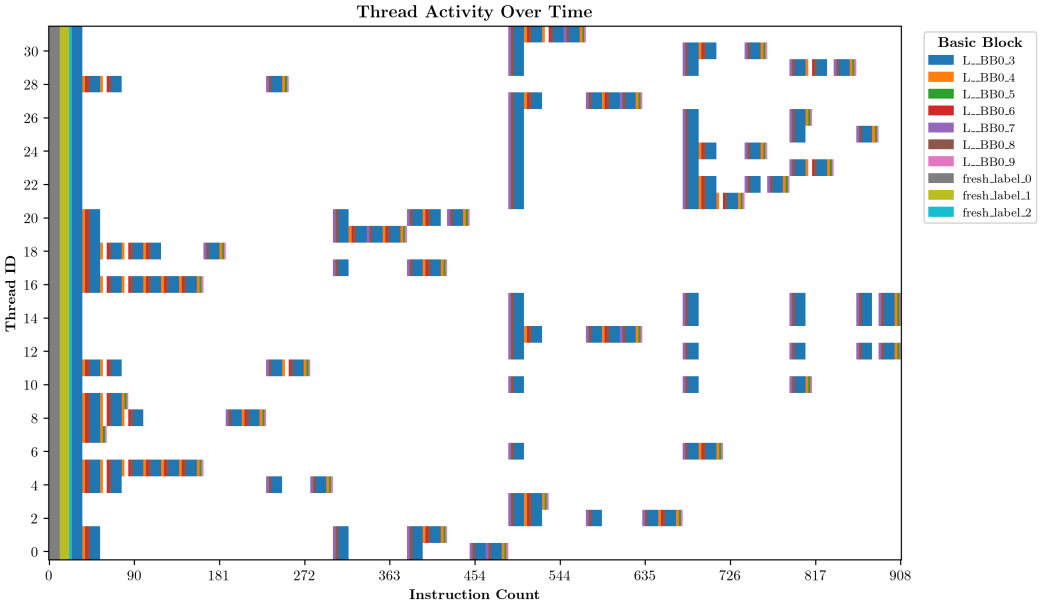
4.2.3 Summary. In this section we have used heatmap and total active thread charts to demonstrate that our simulator effectively captures the divergence and control flow properties of our example programs. We have also shown that our framework allows comparison of different strategies for managing divergent control flow within warps.

5 REFERENCES USED

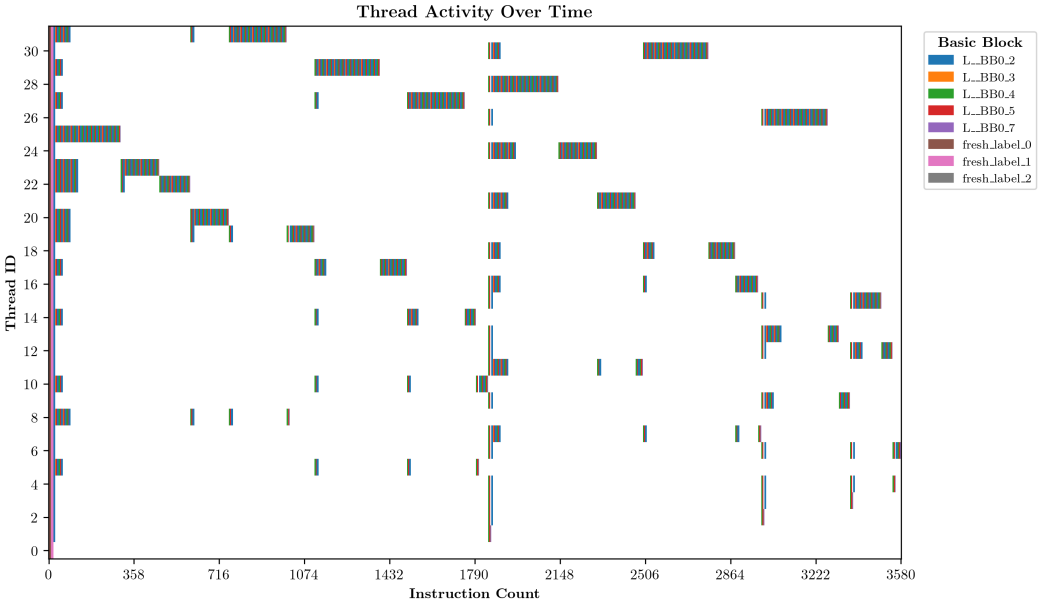
In this section, we briefly list the works we found helpful in completing this project:

- The textbook by Aamodt et al. [1] was very helpful in understanding the reconvergence stack used in the Reconvergence Scheduler.
- The lecture slides by Mahlke [5] provided pseudocode for computing post-dominators and immediate post-dominators.
- The paper on CADSS [6] was very useful in understanding how the simulator framework used in this project.

A full bibliography is included at the end of this document.



(a) `binsearch`



(b) `collatz`

Fig. 8. Heatmap representation for thread activity for one warp executing the `binsearch` and `collatz` example programs using the **Naive Scheduler**. In each chart, the x-axis is instruction count, and the y-axis is the thread ID. A colored cell indicates that the thread is active for the relevant instruction, and a blank cell indicates that thread is inactive. The specific color of each colored cell corresponds to the basic block the instruction belongs to.

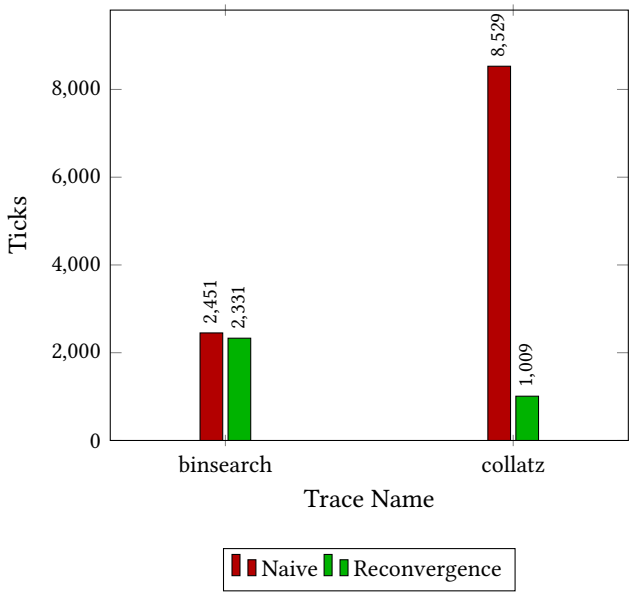


Fig. 9. Runtime in Ticks for the *Naive* and *Reconvergence* strategies on the `binsearch` and `collatz` traces.

6 LIST OF WORK DONE BY EACH STUDENT

Both authors contributed equally to the base for the Scaling and Divergence simulators, namely the simple five stage in-order pipeline implemented at the beginning of the project.

6.1 Ethan

Ethan implemented all of the components and report sections related to the *Scaling Simulator*. In particular, he extended the simulator to support:

- Warp scheduling within SMs to tolerate latency and pipeline stalls. Allows SM to simulate runs with arbitrary many warps and arbitrarly many warp-slots.
- Multiple SMs, and scheduling of warps onto them. Can run arbitrarly many blocks with any possible block size that is divisble by threads_per_warp
- Barrier Instructions, and the associated traces and results.
- Incorporated CADSS memory component to simulator

6.2 Theo

Theo implemented all of the components, results, and report sections related to the *Divergence Simulator*. In particular, he:

- Extended the trace reader to support a new trace file format supporting computation, referred to as “PTX-Minus” in this report.
- Extended the simulator to support the new instructions, and support computation via interpretation of PTX-Minus instructions. Also added support for predicated execution.
- Implemented two scheduling components to explore different strategies for managing thread divergence, including the “Naive” scheduler and the reconvergence scheduler.
- Implemented the supporting “CFG” module used by the scheduling components. Implemented code to build CFGs, compute post-dominators, and immediate post-dominators.

6.3 Distribution of Work

Based on the above distribution we believe both authors have contributed equally to this work.

A EXAMPLE PROGRAMS

A.1 Binary Search Kernel

This parallel binary search kernel is taken from Written Assignment 2. A key learning objective was that the kernel could suffer from thread divergence. We confirm this hypothesis in our report!

```
__global__ void cudaBinarySearch(int n, int m, int *S, int *A, int *R) {
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i >= n) return;

    int search = S[i];
    int left = 0, right = m - 1, middle = 0;

    R[i] = -1; // In case not found

    // Binary Search
    while (left <= right) {
        middle = left + (right - left) / 2;

        if (search < A[middle]) {
            right = middle - 1;
        }
        else if (search > A[middle]) {
            left = middle + 1;
        } else {
            R[i] = middle;
            break;
        }
    }
}
```

A.2 Collatz Kernel

This Collatz kernel computes the number of iterations needed for each element to converge in parallel.

```
__global__ void collatz(int* A, int *result, int n) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    if (tid >= n) { return; }
    result[tid] = 0;

    int x = A[tid];
    while (x != 1) {
        if (x % 2 == 0) {
            x /= 2;
        } else {
            x = 3 * x + 1;
        }
        result[tid] += 1;
    }
}
```

B TRANSPILED "PTX-MINUS" PROGRAMS

After compiling the programs shown in Appendix A, the user can specify values for the parameter at the top of the file. The traces below show the result of running this through the transpiler. Note the simplified syntax and slightly modified instructions versus regular PTX.

B.1 trace_binsearch.txt

```
int _param_0 32
int _param_1 64
int* _param_2 8928 9385 4950 6247 11255 15290 2469 11094 14127 13579 1043 12060 166
     5382 166 166 15290 9385 14371 10388 9249 2983 2117 1275 2278 287 1043 5382 11501
     1263 2278 6447
int* _param_3 166 287 750 1043 1263 1275 1577 1865 2117 2278 2280 2469 2660 2983 3058
     3396 3773 4938 4950 5224 5382 5461 6032 6247 6398 6447 6646 7179 8319 8785 8802
     8866 8928 9033 9053 9204 9249 9385 9400 9698 9971 10247 10388 10513 10794 10802
     10920 11094 11202 11255 11271 11501 11870 12060 12266 13579 13729 13933 14127 14349
      14371 14442 14977 15290
int* _param_4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
---
ldparam.u32 %r13 _param_0
ldparam.u32 %r12 _param_1
ldparam.u64 %rd3 _param_2
ldparam.u64 %rd4 _param_3
ldparam.u64 %rd5 _param_4
mov.u32 %r14 %ctaid.x
mov.u32 %r15 %ntid.x
mov.u32 %r16 %tid.x
mul.s32 %r1 %r15 %r14
add.s32 %r1 %r1 %r16
setp.ge.s32 %p1 %r1 %r13
@%p1 bra $L__BB0_9
cvta.global.u64 %rd6 %rd5
cvta.global.u64 %rd7 %rd3
mul.wide.s32 %rd8 %r1 4
add.s64 %rd9 %rd7 %rd8
ld.global.u32 %r2 %rd9
add.s64 %rd1 %rd6 %rd8
mov.u32 %r17 -1
st.global.u32 %rd1 %r17
setp.lt.s32 %p2 %r12 1
@%p2 bra $L__BB0_9
add.s32 %r23 %r12 -1
cvta.global.u64 %rd2 %rd4
mov.u32 %r24 0
$L__BB0_3:
sub.s32 %r19 %r23 %r24
shr.u32 %r20 %r19 31
add.s32 %r21 %r19 %r20
```

```
shr.s32 %r22 %r21 1
add.s32 %r6 %r22 %r24
mul.wide.s32 %rd10 %r6 4
add.s64 %rd11 %rd2 %rd10
ld.global.u32 %r7 %rd11
setp.lt.s32 %p3 %r2 %r7
@%p3 bra $L__BB0_7
bra.uni $L__BB0_4
$L__BB0_7:
add.s32 %r23 %r6 -1
bra.uni $L__BB0_8
$L__BB0_4:
setp.gt.s32 %p4 %r2 %r7
@%p4 bra $L__BB0_6
bra.uni $L__BB0_5
$L__BB0_6:
add.s32 %r24 %r6 1
$L__BB0_8:
setp.ge.s32 %p5 %r23 %r24
@%p5 bra $L__BB0_3
bra.uni $L__BB0_9
$L__BB0_5:
st.global.u32 %rd1 %r6
$L__BB0_9:
ret
```

B.2 collatz.txt

```
int* _param_0 1 2 4 8 16 5 32 10 3 20 21 6 40 12 13 24 26 17 34 35 11 22 23 7 14 15 28
     29 30 9 18 19
int* _param_1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
      -1 -1 -1 -1 -1 -1 -1
int _param_2 32
---
ldparam.u64 %rd2 _param_0
ldparam.u64 %rd3 _param_1
ldparam.u32 %r9 _param_2
mov.u32 %r10 %ntid.x
mov.u32 %r11 %ctaid.x
mov.u32 %r12 %tid.x
mul.s32 %r1 %r10 %r11
add.s32 %r1 %r1 %r12
setp.ge.s32 %p1 %r1 %r9
@%p1 bra $L__BB0_7
cvta.global.u64 %rd4 %rd3
mul.wide.s32 %rd5 %r1 4
add.s64 %rd1 %rd4 %rd5
mov.u32 %r19 0
```

```
st.global.u32 %rd1 %r19
cvta.global.u64 %rd6 %rd2
add.s64 %rd7 %rd6 %rd5
ld.global.u32 %r20 %rd7
setp.eq.s32 %p2 %r20 1
@%p2 bra $L__BB0_7
$L__BB0_2:
mov.u32 %r3 %r19
and.b32 %r15 %r20 1
setp.eq.b32 %p3 %r15 1
mov.pred %p4 0
xor.pred %p5 %p3 %p4
not.pred %p6 %p5
@%p6 bra $L__BB0_4
bra.uni $L__BB0_3
$L__BB0_4:
shr.u32 %r16 %r20 31
add.s32 %r17 %r20 %r16
shr.s32 %r20 %r17 1
bra.uni $L__BB0_5
$L__BB0_3:
mul.s32 %r20 %r20 3
add.s32 %r20 %r20 1
$L__BB0_5:
add.s32 %r19 %r3 1
setp.ne.s32 %p7 %r20 1
@%p7 bra $L__BB0_2
add.s32 %r18 %r3 1
st.global.u32 %rd1 %r18
$L__BB0_7:
ret
```

C BARRIER TRACE

C.1 Trace File

```
A 7f23f3d92100 2, 1, 0 //PC of 0x555555576fb0
F 7f23f3d92f38 -1, -1, -1 //PC of 0x555555577ff0, F stands for barrier
A 7f23f3d92f38 2, 7, 1 //PC of 0x555555578020
```

C.2 Simulator output

```
Tick: 1
0 instructions waiting for memory
Current Instruction (Warp #0): 0x555555576fb0 //WARP 0 runs first alu instuction
register 2 is used

Tick: 2
0 instructions waiting for memory
Current Instruction (Warp #1): 0x555555576fb0 //WARP 1 runs first alu instruction
register 2 is used

Tick: 3
0 instructions waiting for memory
All warps are stalled.
Current Instruction (Warp #18446744073709551615): 0 //WARP 0 and 1 stalls until all
     instructions in pipeline retires and
updates architectural state

Tick: 4
0 instructions waiting for memory
All warps are stalled.
Current Instruction (Warp #18446744073709551615): 0

Tick: 5
0 instructions waiting for memory
All warps are stalled.
Current Instruction (Warp #18446744073709551615): 0

Tick: 6
0 instructions waiting for memory
stallCount: 1
Current Instruction (Warp #0): 0x555555577ff0 //All instructions from warp 0 retired so
      we can issue barrier, warp 0 is now stalled


Tick: 7
0 instructions waiting for memory
warp : 0 is in a barrier and can not execute
stallCount: 2
```

```
Current Instruction (Warp #1): 0x555555577ff0 //All instructions from warp 1 retired so
     we can issue barrier, both warps are stalled
Tick: 8
0 instructions waiting for memory
warp: 0 has finished!
Current Instruction (Warp #0): 0x555555578020 //Since both instructions reached barrier
     both both are now runnable
register 2 is used

Tick: 9
0 instructions waiting for memory
warp: 1 has finished!
Current Instruction (Warp #1): 0x555555578020
register 2 is used

Tick: 10
0 instructions waiting for memory
All warps are stalled.
Current Instruction (Warp #18446744073709551615): 0 //we are now waiting for
    instructions to retire through pipeline

Tick: 11
0 instructions waiting for memory
All warps are stalled.
Current Instruction (Warp #18446744073709551615): 0 //we are now waiting for
    instructions to retire through pipeline

Tick: 12
0 instructions waiting for memory
All warps are stalled.
Current Instruction (Warp #18446744073709551615): 0 //we are now waiting for
    instructions to retire through pipeline

Tick: 13
0 instructions waiting for memory
All warps are stalled.
Current Instruction (Warp #18446744073709551615): 0 //we are now waiting for
    instructions to retire through pipeline

Tick: 14
0 instructions waiting for memory
All warps are stalled.
Current Instruction (Warp #18446744073709551615): 0 //we are now waiting for
    instructions to retire through pipeline

Tick: 15
0 instructions waiting for memory
All warps are stalled.
```

```
Current Instruction (Warp #18446744073709551615): 0
Ticks - 15
```

REFERENCES

[1] Tor M Aamodt et al. *General-purpose graphics processor architectures*. Springer, 2018.
[2] Wilson WL Fung et al. "Dynamic warp formation and scheduling for efficient GPU control flow". In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE. 2007, pp. 407–420.
[3] *GPGPU-Sim 3.x Manual — gpgpu-sim.org*. http://gpgpu-sim.org. [Accessed 28-04-2025].
[4] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2017. ISBN: 9780128119051. URL: https://books.google.com/books?id=MBQFuAEACAAJ.
[5] Scott Mahlke. *EECS 483 Lecture 20: Control Flow II: Dominators, Loop Detection*. https://web.eecs.umich.edu/~mahlke/courses/483f06/lectures/483L20.pdf. [Accessed 28-04-2025]. 2006.
[6] Brian P Railing. "CADSS: Computer Architecture Design Simulator for Students". In: *Proceedings of the Workshop on Computer Architecture Education*. 2023, pp. 34–40.

Additional References

These are not academic papers, so they are more difficult to cite, we will paste their links here [4] [1].

1. https://forums.developer.nvidia.com/t/maximum-number-of-warps-and-warp-size-per-sm/234378
2. https://stackoverflow.com/questions/10460742/how-do-cuda-blocks-warps-threads-map-onto-cuda-cores
3. https://stackoverflow.com/questions/3087361/gpu-emulator-for-cuda-programming-without-the-hardware
4. https://stackoverflow.com/questions/15240432/does-syncthreads-synchronize-all-threads-in-the-grid
5. YouTube videos by https://www.youtube.com/@NotesByNick
6. Theos 18346 assignments
7. https://en.wikipedia.org/wiki/Thread_block_(CUDA_programming)
8. https://www.arccompute.io/arc-blog/gpu-101-memory-hierarchy
9. modal.com
10. godbolt.com
11. https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#
12. https://docs.nvidia.com/cuda/parallel-thread-execution/
13. ChatGPT

- Used ChatGPT to help us convert the data we gathered into charts and tables
- Ethan asked ChatGPT questions about C++ syntax (ex. How to resize a C++ vector, how to remove a element from a vector)

14. Ethan's 18447 Lab Diagram (modified significantly for diagram in poster)

Received April 29, 2025