Nvidia GPU Inspired SIMD Simulation

Ryan Walton
walton.ry@northeastern.edu

***Abstract*—In this paper a new SIMD hardware acceleration simulator will be introduced that was developed as a final project for the EECE 7352 class at Northeastern University, in the fall semester of 2024. The simulator lends features and nomenclature from existing GPU microarchitectures, and uses an Nvidia GPU comiler and binary tools to develop a set of benchmark kernels for the simulator. A brief description of example GPU microarchitecture features will be introduced and explanation on how they are integrated into the simulator will be provided. A variety of simple benchmark kernels are used to demonstrate the simulator and benchmark its performance under different microarchitecture configurations. Finally, a cycle-by-cycle illustration of the simulator components is composed into image and .GIF files for ease of visualization of the simulator microarchicture.**

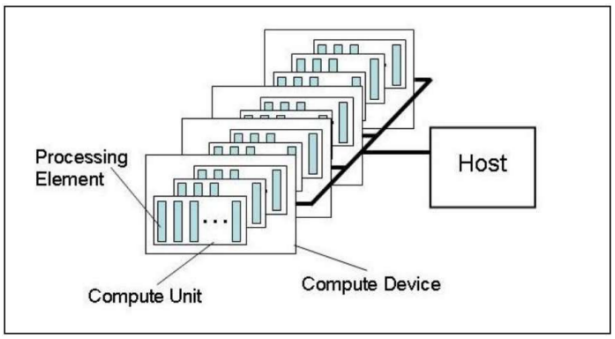

Fig. 1: OpenCL Platform Model [1]

I. Introduction

With the prevalence of GPUs in today's general purpose compute world, it is advantageous for a computer engineer to understand GPU microarchitecture. Such knowledge allows an engineer to understand the workloads that GPUs can be effectively applied to, and yields insight into how best to develop algorithms to run on these massively parallel compute devices.

One way to become familiar with microarchitecture of a computing device is to develop, study, or use a high level simulated model of the device. This paper will, describe a new simulator was written from scratch in python to simulate a simple GPU like SIMD hardware accelerator, which was a reasonably scoped task for a semester long project aimed at learning about GPU microarchitacture.

Existing popular and useful GPU simulators exist. One example is the Accel-Sim simulation framework [6], which provides trace and execution driven simulation of Nvidia Kepler, Pascal, Volta, and Turing microarchitectures for device code expressed in either virtual ISA (vISA, i.e. Nvidia's Parallel Thread Execution, or PTX, [7] ISA) instructions or machine ISA (mISA) instructions. Since this was an educational based effort, a new simulator framework was developed from scratch. Like Accel-Sim however, this simulator framework performs execution driven simulation on a subset of mISA instructions for defined for Nvidia GPUs.

II. GPU Microarchiture Background

Since the goal of the project was to learn about GPU microarchitecture, many microarchictural features present in Nvidia GPUs were incorporated into the simulator design. Nvidia has historically publicly released whitepapers outlining some design features of their GPU architectures. For example, the "NVIDIA TURING GPU ARCHITECTURE" [3] mentions that the Turing GPU architecture has independent integer and floating point datapaths. However, more detailed insight into the microarchitectural features of Nvidia GPUs is not publicly disclosed by the manufacturer, and it is necessary to microbenchmark the hardware in order to back out implementation specifics. [4]

Since this simulation project does not aim to accurately model existing GPU architectures, rather model an arbitrary example SIMD accelerator, concrete knowledge of low level Nvidia GPU microarchitecture is not required. Instead, high level characteristics of Nvidia GPUs are used as a template for the example SIMD accelerator.

As a basis of nomenclature for parallel computation devices, comparison will be made between the OpenCL 1.2 [1] specification and Nvidia terminology. Figure 1 shows the OpenCL Platform Model illustration as printed in [1]. A similar illustration is provided in Figure 2, which is provided in an article on Nvidia's website titled "CUDA Refresher: The CUDA Programming Model" [5]. Three terms of note are pointed out in the OpenCL figure: compute device, compute unit, and processing element. These terms are defined in the OpenCL 1.2 [1] specification, but a brief overview and CUDA equivalent terminology mapping will be provided: A **compute device** is typically a single GPU card, or a single GPU chip for cards with multiple devices. A **compute unit** is defined as the unit upon which a **work group** executes, where a work group is a collection of **work items** that share local memory and support work group barriers. In Nvidia terms, a compute unit is most well represented by a **Streaming Multiprocessor (SM)**, upon which a **thread block** executes, which is a collection of **threads** that share local memory and support work group barriers. Finally, a **processing element** is the basic unit of compute hardware in the OpenCL model, which corresponds to an Nvidia **CUDA Core**.

Additional detail on the microarchecture of Nvidia SMs is available, but has no direct OpenCL 1.2 analog (The OpenCL 3.0 [9] specification defines a "sub-group" which

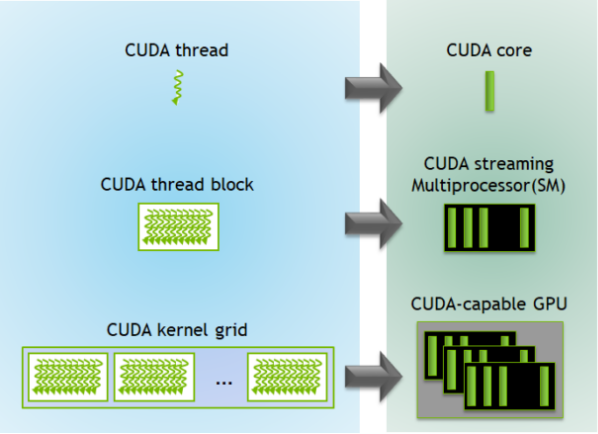

Fig. 2: CUDA Programming Model [5]

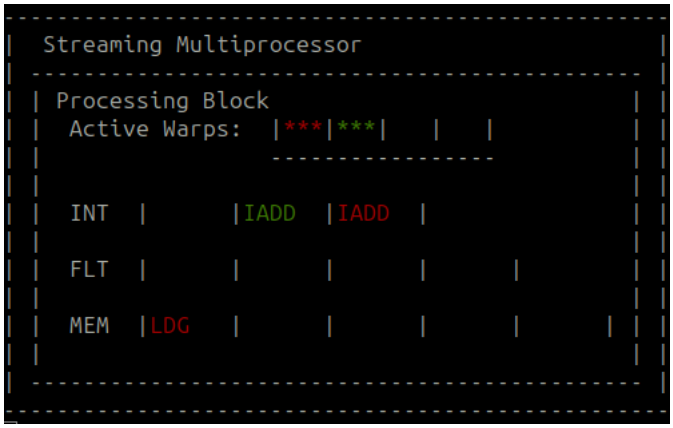

Fig. 3: Example SIMD Simulation State Illustration

can be related to an Nvidia warp, but that will not be discussed further here). The Turing architecture whitepaper [3] details partitioning the SM into four processing blocks, each with a warp scheduler, dispatch unit, 16FP32 cores, 16INT32 cores, and two Tensor Cores. The four processing blocks each share a common shared memory resource and an L1 data cache. The warp scheduler and dispatch unit deal with another collection of threads called a warp in a manner known as Single Instruction Multiple Thread (SIMT) [2]. In control flow sparse code, the threads in a warp will frequently execute the same instruction on different data, maximizing the utilization of arithmetic and floating point logic elements with minimal instruction decoding and scheduling overhead. In Nvidia GPUs, the warp size is 32 elements [2].

III. Custom SIMD Model Simulator Implementation

Given the brief overview of GPU microarchitecture terminology and implementation, the SIMD model that is simulated implements a single compute unit or SM, that has a configurable number of processing blocks, each with a separate integer, floating point, and load/store datapath. Details of assumptions and limitations of the model are described in the following sections.

A. Single SM

Rather than distribute workload across multiple SMs, this model only instantiates a single SM.

B. Instruction Latency

The instructions that run through the different datapaths are assumed to have a fixed latency, depending on the datapath each is executed on. For example, all integer operations might have a latency of 3 clocks while each floating point operation might have a latency of 4 clocks. The latency for each of the 3 datapaths is configurable.

C. Memory Access

Memory access is assumed to be instantaneous (minus the latency incurred by the memory datapath pipeline).

D. Kernel Compilation and Launch

Like a standard CUDA workflow, the simulator supports launching an arbitrary kernel on the simulated hardware with arbitrary input parameters. The kernel is first compiled with the nvidia compiler `nvcc`. Then, the kernel binary is extracted from the compiled object file using `cuobjdump`. Then, the kernel binary is saved as SASS instructions using the `nvdisasm` tool. The simulator framework then parses the instructions, allows the user to design kernel parameters, copies kernel parameters to the simulator memory, and assigns constant memory pointers to the kernel parameters which are referenced by the compiled instructions.

E. Instruction Issue and Hazards

The SIMD simulation model does not support out of order instruction issue. Depending on a configurable flag, the model either issues a single instruction from a single warp or it can issue multiple instructions in a single cycle, provided they target different datapaths and are from different warps.

The SIMD simulation model supports checking of RAW data hazards in the instruction issue logic. Instructions will not be issued for a warp if source operands of the instruction will be changed by instructions still in the pipeline.

F. ASCII Simulation State Illustration

The simulator provides cycle-by-cycle state illustration in the form of printed ASCII characters to the terminal. An example illustration is show in Figure 3. In this illustration, a SM has been configured with a single processing block. There are 4 slots for warps that can be executed in an interleaved and simulataneous fashion, but only 2 slots are filled with active warps. The integer datapath has a latency of 3 cycles, the floating point datapath has a latency of 4 cycles, and the memory datapath has a latency of 5 cycles. The red warp has an `IADD` instruction that is about to complete in the integer datapath and an `LDG` instruction that was just entered into the memory datapath. The green warp has an instruction in the 2nd cycle of the integer datapath. The floating point datapath is idle.

IV. Simulator Validation and Benchmarking

A. Validation

Simulation validation is important to verify conclusions reached are accurate. Many simulators will have a large unit test suite to run on a variety of different workloads and configurations, and compare simulated results to some truth results in order to determine simulator validity. A full unit test suite was not developed for this simulator. However, a test is included that tests 160 different simulation configurations on three separate kernels: a vector multiply kernel, a matrix transpose kernel, and a matrix multiply kernel. The 160 simulation configuration vary 4 parameters:

1) : Enable or disable the multiple instruction per cycle flag, that allows instructions from different warps to be issued to different datapaths in the same cycle.

2) : Warp interleave factor, the number of warps that can be simulataneously scheduled on a given processing block.

3) : Warp width, the number of threads that comprise a warp.

4) : Number of processing blocks.

The vector multiply kernel takes two input arrays, and stores the element wise floating point product of values at corresponding indices in a corresponding index of one of the input arrays. This is compared with the python vector multiply implementation and confirmed to not differ at each index. The matrix transpose and matrix multiply kernels do as their names suggest, and are also compared with a python implementation for verification.

The three kernels are shown below. The matrix transpose and matrix multiply kernels both have the matrix dimension defined at compile time, while the vector multiplication kernel can be defined at runtime due to the design of the kernel.

```
__global__ void vecmul( float* buf1, float* buf2 )
{
    auto idx = threadIdx.x + blockDim.x * blockIdx.x;
    buf1[idx] = buf1[idx] * buf2[idx];
}

constexpr uint32_t mat_dim = 64;

__global__ void mattranspose( float* buf1, float* buf2 )
{
    auto idx = threadIdx.x + blockDim.x * blockIdx.x;
    auto idx_i = idx / mat_dim;
    auto idx_j = idx % mat_dim;
    buf2[ idx_j * mat_dim + idx_i ] = buf1[ idx_i * mat_dim + idx_j ];
}

__global__ void matmul( float* buf1, float* buf2, float* buf3 )
{
    auto idx = threadIdx.x + blockDim.x * blockIdx.x;
    auto idx_i = idx / mat_dim;
    auto idx_j = idx % mat_dim;
    float sum = 0;
    for( size_t k=0; k<mat_dim; k++ )
    {
        sum += buf2[ k * mat_dim + idx_j ] * buf1[ idx_i * mat_dim + k ];
    }
    buf3[ idx_i * mat_dim + idx_j ] = sum;
}
```

B. Warp Interleaving and Multi Issue

One of the applications of this simulator is to visualize the merit of certain GPU microarchitectural features. Two features that are easy to visualize are cycle level warp interleaving and multiple instruction issue per cycle to different datapaths. The cycle level warp interleaving allows for latency hiding of the datapaths, that would otherwise cause stalls to avoid hazards, by saturating the hardware with independent instructions from different warps. The multiple instruction issue per cycle allows for potential of efficiently using more than just one datapath at a time, when instructions executed target a mix of datapaths. Figure 4 shows a snapshot of the case in which only a single warp is able to execute on a processing block at a time, causing lots of stalls as data hazards are addressed. Figure 5 shows a snapshot of the case in which 6 warps are able to execute on a processing block at a time. The hardware utilization efficiency is much higher since independent instructions can be issued one after another, and in some cases simultaneously when the instructions target different datapaths.

Fig. 4: Single Warp Active

For the A sweep of various simultaneous warp scheduling values was performed with multiple instruction issue per cycle enabled and disabled on the three kernels that were introduced in the simulator validation section. The results are shown for the vector multiplication, matrix transpose, and matrix multiply kernels in Figures 6, 7, and 8, respectively. The vector multiplication kernel was executed on two vectors with 8192 32-bit floating point elements, and the matrix transpose and multiplication kernels were both executed on 64x64 square matrices. The benchmarking was performed with 2 processing blocks in the SM, a warp width of 32, an integer datapath latency of 2 cycles, a floating point datapath latency of 3 cycles, and a memory datapath latency of 3 cycles.

For each of the kernels, it can be seen that increasing the number of warps scheduled simultaneously decreased the execution time of the benchmark, with diminishing returns after about 3 simultaneous warps. Multiple issue also significantly improves performance, especially when 3 or more warps are scheduled simultaneously. The multiple issue enabling improved performance drastically in the matrix multiply benchmark compared to the other benchmarks. This is because the compiler interleaved loads and fused multiply addition floating point operations in the machine code when it unrolled the for loop inside of the kernel. This allowed for a near doubling in IPC when compared to the single issue case.

C. Number of Processing Blocks

The number of processing blocks in a SM directly impacts SM performance and size. The number of processing elements in an SM varies linearly with the number of processing blocks, for a given processing block configuration. However, increasing number of processing blocks causing issues with

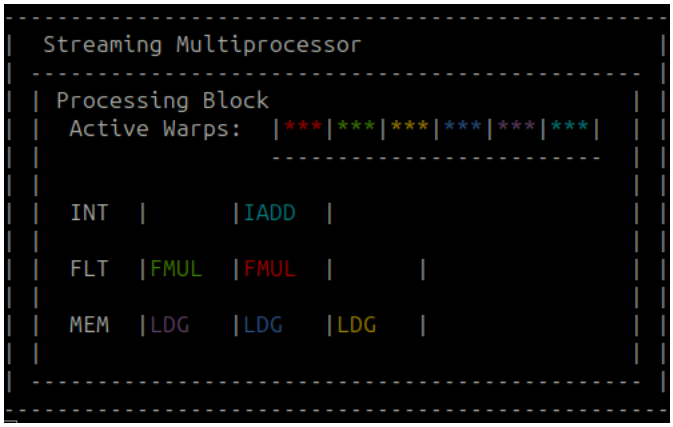

Fig. 5: Many Warps Active

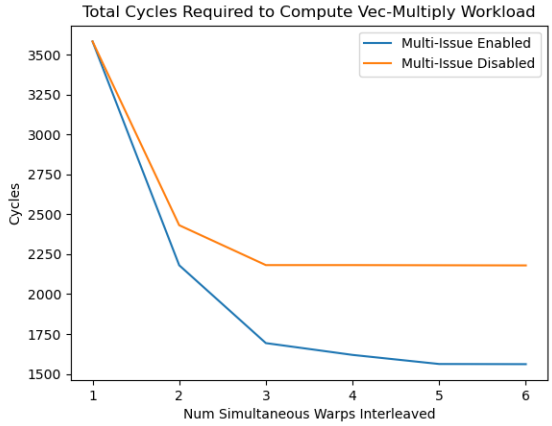

Fig. 6: Vec-Multiply Computation Time With Different Interleave and Multi-Issue Parameters

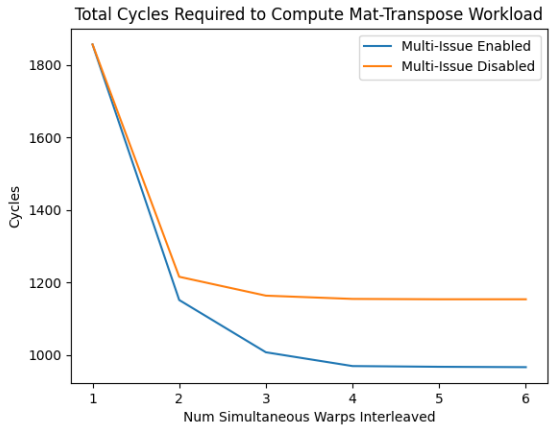

Fig. 7: Mat-Transpose Computation Time With Different Interleave and Multi-Issue Parameters

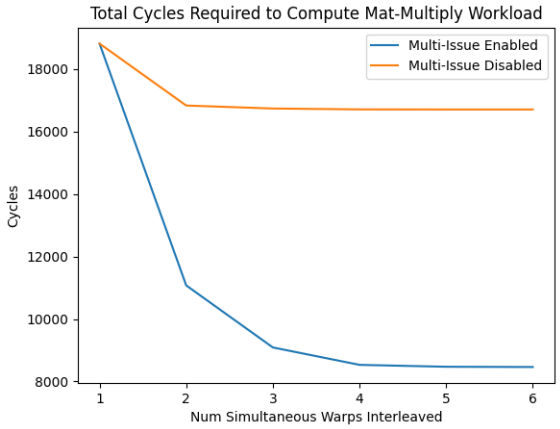

Fig. 8: Mat-Multiply Computation Time With Different Interleave and Multi-Issue Parameters

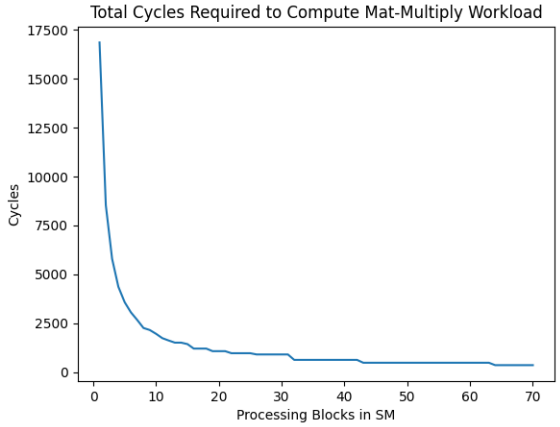

Fig. 9: Mat-Multiply Computation Time With Varying Number of Processing Blocks

memory systems etc. to arise. A fictitious test that arbitrarily swept the number of processing blocks in an SM was performed, assuming no change in memory access time etc. The result is shown in Figure 9. The figure clearly shows Amdahl's inversely proportional relationship between speedup and parallelization. Also, steps in the graph are visible as the slope becomes closer to zero. These steps are due to work load batching losses: when the number of warps needed to execute is not a multiple of the number of processing blocks, at some point the processing blocks will experience varying levels of saturation, causing slight overall performance decrease.

V. Simulator Limitations

While the simulator that was developed is somewhat capable, several limitations exist that haven't yet been addresses.

A. Control Flow

Throughout the paper, the simulator has been described as a SIMD machine simulator rather than a SIMT machine simulator. This is because control flow is not yet implemented.

Control flow would allow the simulator to parse and execute predication instructions, branches, conditional branches, etc. This would allow the simulator to perform algorithms whose execution depends on the value of the data. It would also allow the simulator to execute algorithms with more arbitrary sizing. For example, the matrix transpose and multiply kernels that developed and tested on the simulator required that the matrix size was known at compile time. If control flow was supported in the simulator, the kernel could be designed with a matrix size that was defined at runtime instead. There exist several papers that dive into the possible microarchitecture of Nvidia GPUs that allow for robust control flow, for example "Control Flow Management in Modern GPUs" [8], a very recent paper that propose Hanoi, an efficient mechanism for control flow in GPUs.

B. Memory Hierarchy

The memory hierarchy is one the things that makes GPUs so competitive for high performance computing tasks. There are lots of nuance in GPU microarchitecture in the memory hierarchy, including special shared memory local to a SM, memory access coalescing, and banking. In order to get the best hardware utilization efficiency on a GPU, it is very important to understand the memory hierarchy and efficient memory access patterns, so it is unfortunate that this topic was not able to be investigated in this project.

C. Single Compute Unit / SM

As mentioned before, the simulator only supports simulating a single compute unit / SM. It would be relatively easy to extend the simulator to having multiple SMs once the memory hierarchy limitations were addressed.

D. Subset of All Instructions

Only a subset of all of the Nvidia SASS instructions were implemented for this simulator. Implementation of all instructions in the ISA would require significantly more work, and potential for reverse engineering of those that are undocumented.

VI. Conclusion

GPUs are a hot topic today for a variety of industries including AI, cryptography, and scientific computing. Understanding GPU microarchitecture can better equip a computer engineer to tackle today's problems. Implementing a simulator like this provided opportunity for a deep dive into existing literature on GPU microarchitecture and hands-on understanding of the challenges associated with GPU design.

References

[1] A. Munshi, Ed., *The opencl specification*, version 1.2, Khronos OpenCL Working Group, 2012.

[2] Y. Lin and V. Grover, "Using cuda warp-level primitives," 2018. [Online]. Available: https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/.

[3] *Nvidia turing gpu architecture*, WP-09183-001$_v$01, NVIDIA Corporation, 2018.

[4] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza, "Dissecting the nvidia turing T4 GPU via microbenchmarking," *CoRR*, vol. abs/1903.07486, 2019. arXiv: 1903.07486. [Online]. Available: http://arxiv.org/abs/1903.07486.

[5] P. Gupta, "Cuda refresher: The cuda programming model," 2020. [Online]. Available: https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/.

[6] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "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.

[7] *Ptx isa*, 8.5, NVIDIA Corporation, 2024. [Online]. Available: https://docs.nvidia.com/cuda/parallel-thread-execution/.

[8] M. A. Shoushtary, J. T. Murgadas, and A. Gonzalez, *Control flow management in modern gpus*, 2024. arXiv: 2407.02944 [cs.AR]. [Online]. Available: https://arxiv.org/abs/2407.02944.

[9] *The opencl specification*, version v3.0.17, Khronos OpenCL Working Group, 2024.