Adam Zuckerman
CS370A
Paper 3

Parallelism in NVIDIA's Cuda Architecture

Compared to CPUs, GPUs offer a superior level of parallel computing made possible by a parallel architecture of thousands of cores working in tandem to process large sets of instructions, data, and control of concurrent threads. Parallel computing through a GPU such as Nvidia's CUDA architecture revolutionized industries requiring advanced machine learning algorithms. The next generation of modern datacenters are currently being equipped with Nvidia's Blackwell architecture utilizing 5th generation Tensor cores designed to train AI algorithms on trillion-parameter models. This new architecture still utilizes the CUDA core programming libraries which set the foundation for advancements in parallel computing that we have observed. However, there is a tradeoff in use cases. Tensor cores execute multiple instructions per clock cycle and CUDA cores execute a single operation per clock cycle. The tradeoff is in speed versus accuracy. Tensor cores are preferred for LLM (Large Language Models). Whereas CUDA cores are preferred for general purpose parallel computing and rendering of 3D graphics.

To understand how we arrived at Tensor core technology it is essential to understand the key concepts behind CUDA core parallel computing which are instruction level parallelism, thread level parallelism, and data parallelism. This paper will examine the CUDA microarchitecture of the Fermi streaming multiprocessors from which the newer CUDA microarchitectures of the Ampere system were built on.

Both the CPU and GPU work in tandem to process instructions and data sets. The

CPU generally handles serial input and is designed to handle instructions of more complex arithmetic. When the CPU encounters large data sets that need highly parallelized processing the CPU hands off the parallelized instructions to the GPU. The difference is in the instruction model supported by the microarchitecture. The CPU works from the MIMD (Multiple Instructions Multiple Dataset) whereas the GPU is enhanced for ILP (instruction level parallelism) with the SIMD (Single Instruction Multiple Data) model. SIMD can boil down to the adage, “for this range of data, perform this operation”. (Cook, pg. 29)

The CUDA GPU is essentially a cluster of SIMD vectors. However, a more accurate classification is the SIMT instruction model (Single Instruction Multiple Thread). As the cores of a SM (streaming multiprocessor) are capable of more than one type of instruction in flight. To identify how CUDA architecture enables parallel computing, we must discuss how the top-down control structures transform data sets through parallel execution of the same instruction type at a time.

At the top of the CUDA architecture, we have the Giga-Thread global scheduler which organizes workflow to the SM and assigns the number of warps and blocks allocated based on the capacity of the hardware. In the legacy Fermi architecture, a SM contains 32 SIMD cores from which a set of instructions are run in parallel to preform transformations on the currently loaded datasets. (Faber, pg. 88) In addition to the cores of the SM, in the die there is also an instruction cache, warp scheduler, registers, SFU (complex arithmetic unit), L1 cache, and various texture caches for rendering graphics. The SM is allocated blocks of instructions, and each block of instructions contains multiple iterations of warps. This architecture leads to grids of blocks executing instructions concurrently across the

cores of the SM. Instructions cascade down from the kernel to global scheduler and are stored in the instruction caches then to warp scheduler sitting at the top of the control structure in the SM. Each SM contains two warp schedulers that are responsible for maintaining parallelism in the streaming microprocessor through controlling which warp within the block is to execute in the current clock cycle. Subsequent organization of grids, blocks, and context switching between warps to specific instructions begin.

In the Fermi GPU architecture, there are 24k active threads that can be allocated and organized amongst grids, blocks, and warps. (Cook, pg.25) It is the programmer's responsibility to achieve maximum utilization of the GPU's resources through efficiently allocating blocks and warps to the application's workload. Latency issues arise when warps assigned data dependent instructions are stalled waiting for memory retrieval of the critical parameters to resolve the dependency issue.

As programmers we can schedule multiple warps per block to hide both data/arithmetic latency and increase TLP (Thread Level Parallelism). This allows the SM to select more warps from the ready queue to implement the next set of instructions while the stalled threads are waiting for the current memory fetch. However, increasing TLP alone will not result in the application running at higher speeds. The warp scheduler must also increase ILP (Instruction Level Parallelism) to put the highest number of warps executing instructions inflight. To increase ILP we must select sets of instructions that have been determined by the warp scheduler to be data independent. Instructions that are data independent use less of the GPU's shared resources. This type of instruction does not require a memory fetch outside the processor's local memory stores. Scheduling critical

process maintains parallelism as instructions that contain conditional logic that branches into different paths causing a condition called warp divergence. The warp scheduler prioritizes instructions that already have parameters in local storage and instructions where conditional statements follow a similar path. (Farber, pg.94)

The goal is to organize the ready queue so that the number of warps currently assigned to data dependent instructions are at a lower position in the ready queue to keep instruction stalls at minimum. This tactic lowers the amount of time that data dependent instructions are in the ready queue to minimize resource loss and allows for multiple concurrent execution of data independent instructions sets per clock cycle. In this way, NVIDIA GPUs use a form of superscalar execution similar to the way that CPUs use branch prediction algorithms to fetch the next set of relevant instructions. Thereby increasing the amount of memory available in the processor's register through data reuse.

The last topic we must discuss is how the memory architecture of the CUDA system enables DLP (Data Level Parallelism). Another approach to achieving better performance through parallelism is to transform data through data decomposition. Decomposition involves breaking the arrays of a data set into sections and allocating blocks to the SMs so that each thread is handling one data point of the array. We prioritize the distribution of data first and the transformations on the data second.

A real-world example of a data parallelism problem is how blocks and grids are assigned to a HD photo. For a 2560x1440 image we would decompose the 2D image into an X and Y axis creating rows and columns of arrays. When allocating blocks to the rows we need to ensure that the blocks are assigned in multiples of the warp size which is 32. If it is

initially not a multiple you will need to pad the rows with extra blocks until you reach a multiple of 32. If we choose a thread size per block of 256, we will need 10 blocks to complete the top row of the HD picture for a total of 2560 threads with each thread assigned to a data point or pixel within the row. There will be a total of 1440 rows across the Y axis. 10 x 1440 gives us the total number of blocks assigned to the grid. Which is 14,400 blocks with 256 threads per each block totaling 3.6 million threads assigned at a 1 to 1 ratio for each pixel. This example shows how we can utilize a grid structure of blocks to create a 2D data set achieving a parallel data structure of indices. (Cook, pg. 84-85) As CUDA cores are designed for graphics rendering, we can also create a 3D data structure of X, Y, and Z coordinates to render a 3D structure. This concept is heavily utilized by video games when creating 3D worlds.

When working on a parallel data set, we need to achieve atomic read and write operations on the data set so that every processor and thread sees the same data. For atomic read and write operations within the SM we utilize the L1 cache which all cores share and see the same set of data. When performing atomic read and writes between SMs we utilize the L2 cache as shared memory storage local to the SMs. To ensure atomic read and writes, threads that are accessing the same memory address in the cache will only preform one fetch for the shared data. The shared data is then broadcasted across the threads to avoid concurrent read and writes to the same memory address. Data coalescence allows for parallel instruction execution on multiple datasets. Atomicity in read and rights helps to coalesce memory access into one memory transaction helping avoid race conditions within the memory stores. (Farber, pg. 125) Naturally memory stores

that are closer to the cores are the fastest memory fetches and as you move outward into the global memory the fetch speeds progressively get slower.

Circling back to the way we achieve ILP through the scheduling of data independent instructions, we schedule instructions so that the ones that are in flight are instructions that share common parameters that can be stored closer to the processor. The further the location in memory the parameter of an instruction is the more clock cycles the processor takes to fetch. If parameters aren't stored locally the programmer designing the application will need to consider the most efficient way to retrieve information from the global stores. Instructions and parameters that require a global fetch are loaded into memory through from the top down from the DRAM into the processor's global store and then to the L2 cache. The number of round-trip fetches is to be minimized through coalescence of instructions and data from the global stores. Where the L2 shines is in its ability to coalesce memory from the global stores in the 128-byte cache size. (Farber, pg.126) The L2 cache will implement a least recently used model to flush the memory storage of data that is rarely accessed to avoid bottlenecks in the memory bandwidth, once the LRU instructions are flushed from the L2 cache writes are preformed to the global memory stores.

Nvidia’s CUDA architecture enables parallelism across data, instruction, and threads. CUDA architecture has been specifically engineered to allow software engineers to transform sets of data in parallel at unprecedented speeds in modern computing. In today’s Ampere architecture the amount of SM the number of SM has increased greatly from the 15 SM and 32 CUDA core system that was used in 2010. Today’s graphics cards

feature 45 SM and 128 CUDA cores allowing for truly next-generation parallel computing. In addition to the CUDA cores that enable general purpose computing the Ampere architecture also has Tensor cores capable of complex matrix operations that LLMs use in artificial intelligence. We are now approaching a milestone in microchip design with Nvidia's Blackwell architecture allowing for LLMs in the trillions of parameters and the transmission of data across the chip's die at a rate of 10 terabytes per second! The concepts of parallel programming from Fermi 's SIMT architecture to today's Ampere and Blackwell architecture remain the foundations in achieving high speed parallel processing.

Bibliography

Cook, S. (2014). *Cuda programming: A developer's guide to parallel computing with gpus*. Morgan Kaufmann.

Farber, R. (2014). *Cuda Application Design and Development*. Morgan Kaufmann.

AceCloud. (2023, October 4). All you need to know about SIMD and how GPUs employ it. *AceCloud*. https://acecloud.ai/resources/blog/all-about-simd-and-how-gpus-employ-it/

AceCloud. (2024, July 24). CUDA cores vs Tensor cores – which one is right for machine learning. *AceCloud*. https://acecloud.ai/resources/blog/cuda-cores-vs-tensor-cores/

NVIDIA Blackwell architecture. (n.d.). NVIDIA. Retrieved October 17, 2024, from https://www.nvidia.com/en-us/data-center/technologies/blackwell-architecture/