GPU architecture: Hardware/Software approach

Mohamed Elsayed, an MSc student at DIT (Deggendorf Institute of technology) - Germany

***Abstract*—This paper provides an introduction to general purpose graphical processing units (GPGPUs) architecture where it covers three different perspectives. First, the paper covers the underlying execution model and the basic hardware structure of GPGPUs and their compute cores. Second, it examines memory architecture of modern GPUs and how GPU kernels utilize memory for computation. This section also explores the memory hierarchy and different types of memory that can be used in software kernel design. Finally, the paper discuss GPU programming, illustrating this through CUDA programming model. This specific part will cover how GPU kernels can be designed, how to leverage hardware architecture knowledge into designing an efficient GPU CUDA kernels and how to measure its performance.**

***Index Terms*— CUDA programming model, Dynamic scheduling, GPGPU, Heterogeneous systems, kernel design, Multithreading computation, Parallel algorithms, Parallel programming, SIMD execution model and SIMT management model.**

I. INTRODUCTION

Graphics processing units (GPUs) in modern systems aren't standalone devices; they are combined with central processing units (CPUs) to represent a single efficient system capable of providing higher instruction throughput and memory bandwidth i.e. frontier supercomputer [1].

There are 2 possible configurations for such heterogeneous systems [2]. The first is a discrete configuration, where each GPU and CPU in the system has its own separate physical memory space and communication between a host (CPU) and device (GPU) is conducted explicitly through a system bus, such as PCI Express (PCIe) bus. The second configuration is known as a system on chip (SOC). In this configuration, both the GPU and CPU share the same physical memory space, known as unified memory architecture (UMA). This approach simplifies the programming model as the memory coherency is maintained by the hardware. System on chip (SOC) examples include the Qualcomm snapdragon mobile GPU that is integrated into mobile phones and NVIDIA Jetson Nano development board which features a CPU of ARM Cortex-A57 with four cores and 512 CUDA cores (execution units – known as streaming processors (SPs)) based on the Maxwell architecture that has a unified memory architecture support from NVIDIA which can be exploited by using CUDA programming model [3] [4].

The primary design purpose of GPUs is to execute thousands of threads in parallel resulting in higher instruction throughput than CPUs alone, hence GPUs are specialized hardware devices that should be used for high parallel computations and scientific simulations that include heavy data processing.

The paper is divided into four sections. Section I is going to cover an overview about GPGPU architecture and the underlying models which is divided into 2 models for different design purposes. The first model is Single Instruction Multiple Thread (SIMT) management model, and section I is going to show how this model manages threads scheduling, and the second model is single instruction multiple data (SIMD) execution model responsible for the actual multi-thread execution. Section II covers the memory hierarchy and memory architecture in GPUs and how to use it efficiently in GPU kernel design. Section III discusses the CUDA programming model, CUDA kernels design such as reduction sum kernel and how it can be optimized using the GPU architecture knowledge for multithreading behavior. It will also cover optimization techniques to consider when designing these kernels along with how to analyze a certain GPU kernel for bottlenecks. Section IV is a brief introduction for applications that use GPUs for their operations and what benchmarks available in the HPC industry.

II. GPGPU EXECUTION MODEL

GPU execution model follows Single Instruction Multiple Data (SIMD) model and the Single Instruction Multiple Threads (SIMT) model. SIMT is a model to manage and schedule threads where these threads are organized into groups. The naming convention for such grouping of threads is different from a vendor to another; for instance, NVIDIA calls such these groups as warps while AMD calls them wavefronts. A warp or a wavefront is the hardware scheduling unit. This indicates that at any instant in time a scheduler is going to select a warp of threads for execution where all threads in this warp is going to execute in a SIMD way; same instruction is executed by all threads in a warp but on different portions of data [5].

The pipeline in Fig. 1 is a generic view for a GPGPU microarchitecture as mentioned in [2].

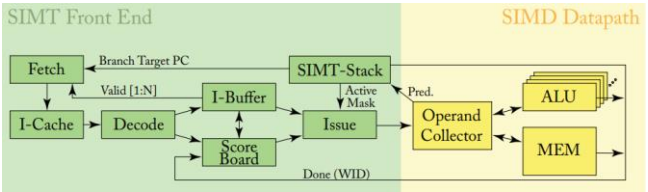

Fig. 1: Microarchitecture of a generic GPGPU core [2]

The pipeline is divided into 2 parts as described in [2]. The first part is SIMT front end that manages and schedules the threads and is divided further into 3 scheduling loops:

- *Instruction fetch loop*: Covers the following blocks
  - Fetch, I-Cache, Decode and Instruction buffer (I-Buffer)
- *Instruction issue loop*: Covers the following blocks
  - Instruction buffer, Score board, SIMT-STACK and Instruction issue block.
- *Register access loop*: Covers the following blocks
  - Memory, operand collector and execution units (ALUs).

The pipeline starts by having a warp scheduler, similar to Fig. 2, which selects a group of threads (warp) that has enough space in the instruction buffer, then the instruction fetch loop is going to fetch an instruction from the warp's code by checking the program counter and check the instruction cache access to fetch the next instruction i.e. Warp 8, instruction 11 [6]; there might be a cache miss in some instructions. After the instruction is fetched successfully, it's saved in the instruction cache then decoded and saved in the instruction buffer.

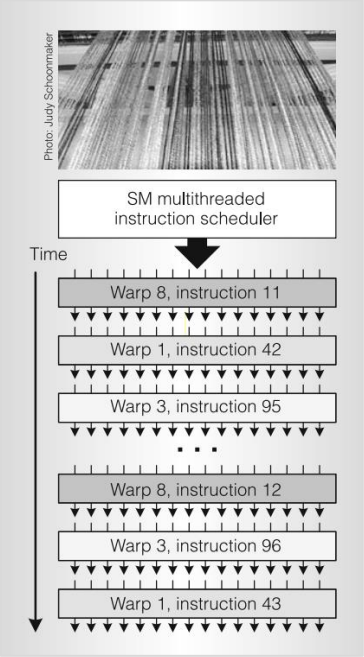

Fig. 2: Single-instruction, multiple-thread (SIMT) warp scheduling [6]

Another scheduler is going to access the instruction buffer and choose an instruction that doesn't have any major dependencies. The data dependency between instructions for the same warp can be checked using a scoreboard implementation, similar to the CPU implementation to check for data hazards. After choosing the instruction for a warp, the instruction is issued for execution and the register access loop gets the required data from memory and start the execution where all the threads in the warp are going to execute the same instruction at the same time; guaranteed by the SIMD execution model where the execution units in Fig. 1 are simply ALUs.

In GPU kernel design and parallel programming, some threads might diverge in the execution due to a conditional branch statement where the condition is dependent on the thread index in the block/grid. In this scenario, the most common approach in GPUs to handle thread divergence is to serialize the execution, as described in [7], by using *SIMT-STACK* which provides an execution mask where each bit in this mask can be of value 1, which indicates that a certain thread is going to execute the current instruction in the pipeline, or value 0 which indicates that the current thread is waiting for other threads to execute the current instruction in the pipeline.

Consider the example in Fig. 3.a taken from [2], it shows a computational graph of a certain code snippet where at statement A, a SIMT mask of all ones is generated; which means that all threads inside a warp are going to execute this instruction. This example assumes that a warp consists of 4 threads only; this doesn't have to be the case in different GPUs.

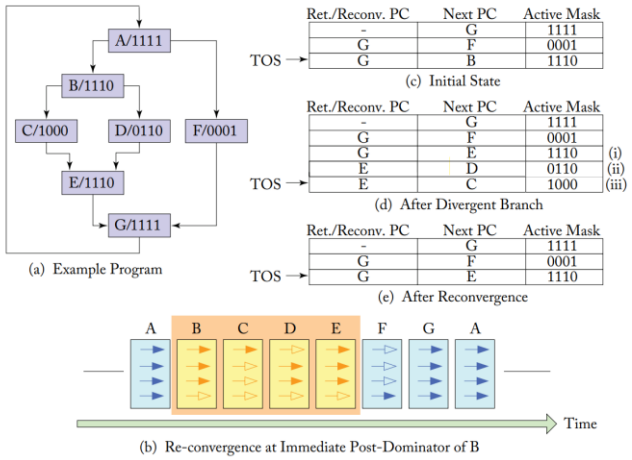

Fig. 3: SIMT stack operation - serializing execution of threads under thread divergence condition [2] [7]

After statement A, there is a divergence where 3 threads are going to execute statement B and only 1 thread is going to execute statement F and hence the SIMT mask bits change according to the thread index in the grid. In Fig. 3.b, it shows how thread execution serialization happens across time where the graph faces multiple thread divergence and convergence points during the GPU kernel lifetime. A stack based implementation is used to hold the convergence points. Whenever there is a thread divergence, each entry in the stack holds the re-convergence instruction, the address of next instruction to be executed and the active mask for thread execution status.

Modern GPUs implement the handling for thread divergence differently where the stack approach is not used. This is known as stack-less convergence mechanism. NVIDIA Volta architecture, as described in [8] and [2], implements stack-less convergence barrier where it depends on hardware registers to store information to track which thread within a warp is going to participate in a certain convergence barrier and this knowledge is used by the hardware warp scheduler.

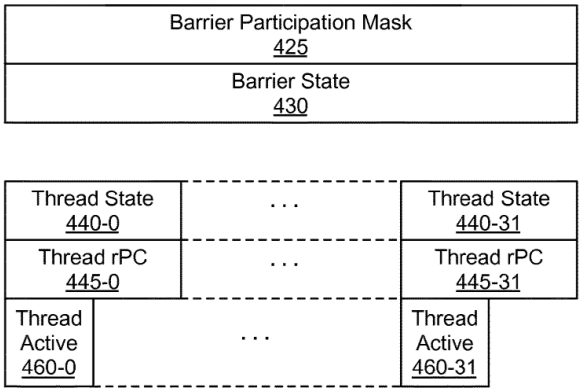

Fig. 4: Illustration of convergence barrier state. [8] [2]

The stack-less approach, as shown in Fig. 4, is divided into 4 registers and 1 mask, as mentioned in [2] and [8], which gets updated in the runtime for each warp of threads and used by the hardware warp scheduler:

- *Barrier Participation Mask*:
  - A convergence barrier participation mask is used to track which thread within a given warp is participating in a given convergence barrier.
  - Threads tracked by a given barrier participation mask will wait for each other to reach a common point (Re-converge together).
  - The size of this mask is dependent on number of threads within a warp; i.e. for a warp of 32 threads, the size will be 32 bits mask.
  - There might be more than one barrier participation masks within a warp as threads might diverge multiple times.
- *Barrier state*:
  - This state tracks which threads have arrived at a given convergence barrier.
- *Thread state*:
  - This state tracks, for each thread within a warp, whether the thread is ready to execute or not.
  - It has only 3 states:
    - *Blocked state*; where a thread is blocked at a convergence barrier.
    - *Yielded state*; it's used to enable other threads in a warp to progress past the convergence barrier (Used in a situation where a deadlock might have happened).
    - *Ready to execute state*: A thread is ready to start execution.
- *Thread rPC*:
  - It's used to track the address of the next instruction to execute for an idle thread (A thread that's not active in the current warp of threads).
- *Thread Active*:
  - This state is updated by the hardware scheduler to indicate whether the thread within this warp is active or not.

As described in [2], the hardware scheduler selects a warp for execution. For an architecture that supports a warp size of 32 threads, the *barrier participation mask* is going to be 32 bits where each bit indicates whether this thread is going to participate in the convergence barrier or not. The *barrier participation mask* is used by warp scheduler to stop threads at a specific convergence barrier location. It's initialized using a special *ADD instruction* where all active threads within a warp are going to execute this *ADD instruction* and hence the corresponding bit for each thread in the convergence barrier is going to be 1; if there is a divergence, some of these bits are going to be 0 as the *ADD instruction* wasn't executed. If there is multiple divergence points within a warp, there will be multiple barrier participation mask per warp.

After a divergence, the address of next PC to execute will differ between some of the threads within a given warp; the scheduler is going to select a subset of threads with a common PC and update the *thread active field* which is just a 32 bits mask that specifies whether a thread is going to be active or not to execute the current instruction. This behavior is called *warp split*, and it gives a new degree of freedom within a warp where the scheduler is free to switch between groups of diverged threads. *Warp split* can be stopped by executing a *WAIT instruction* when it reaches a convergence barrier.

The *WAIT instruction* adds the threads in the warp split to barrier state register to change thread state to blocked state. Once all threads in the barrier participation mask have executed *WAIT instruction*, the thread scheduler switches all threads from the warp split to active and update the *Thread Active register*. The following section covers memory model and integrates with the execution model knowledge to design efficient GPU kernels.

III. GPGPU MEMORY MODEL

This section covers the GPGPU memory model from the programming point of view and how GPU kernels can use different memory spaces with different memory access patterns that has great effect on the kernel performance.

GPU memory is divided into local and global memory spaces as described in [3] [4] [2]. Fig. 5 shows an NVIDIA Tesla architecture which has multiple streaming multiprocessors clusters and within each cluster there are two streaming multiprocessors (SMs) which map to the SIMT core in the general GPU architecture in Fig. 1. Each SIMT core consists of multiple streaming processors (SPs) or known as CUDA cores; these simply map to the ALUs in the general GPU architecture in Fig. 1. In tesla architecture, it shows global off-chip memory spaces designed using DRAM technology that can be accessed by all the threads inside a grid of threads, however this memory bandwidth is small and requires lots of

time to read/write data. Each thread in the grid has its own local memory space i.e. local registers per thread, and each block of threads has a shared memory that can be accessed by all the threads within a block as shown in Fig. 6 i.e. a scratchpad memory [4].

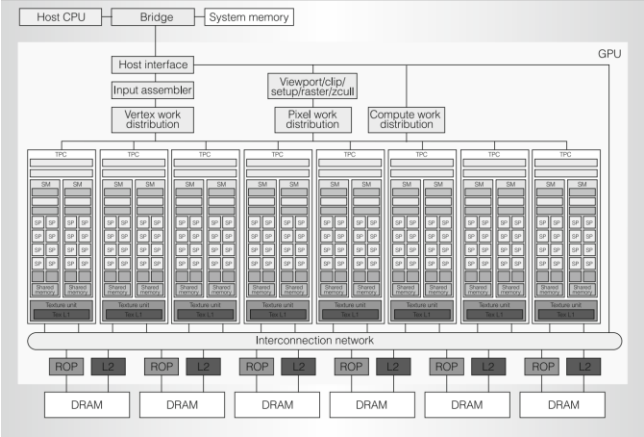

Fig. 5: NVIDIA Tesla GPU architecture details [6]

In order to start an operation on a GPU, the CPU runtime is going to explicitly copy all data needed for processing from CPU memory address space to a GPU memory address space, which normally is the global off-chip DRAM memory, and then the CPU runtime is going to invoke a GPU kernel, which is written by a certain programming model [4]. Finally, the work is distributed by launching a grid of threads where the grid is divided into blocks of threads and each block consists of certain number of warps i.e. 1024 threads per block maximum divided into groups of 32 threads per warp (architecture dependent).

The memory access time is always a bottleneck for a kernel execution. When designing a GPU kernel, one thing to keep in mind is that the fastest memory to access is the local memory per thread and then the shared memory per block as both of which are an on-chip type of memories. The memory hierarchy from NVIDIA GPU and CUDA programming model point of view is shown in Fig. 6.

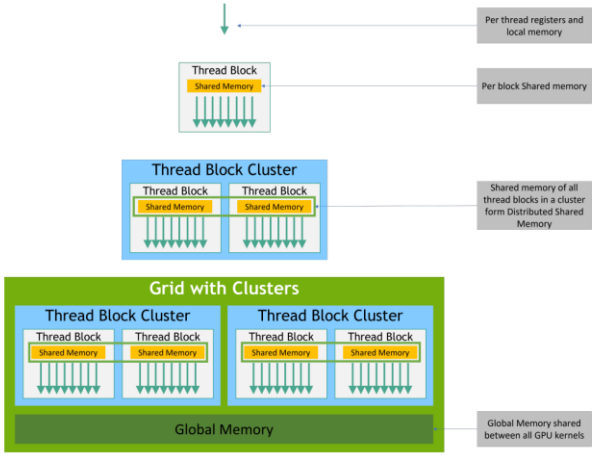

Fig. 6: Memory hierarchy from NVIDIA CUDA model point of view [4].

A memory model of unified memory architecture, as mentioned in [3], is also implemented in GPUs and shown in Fig. 7; NVIDIA implemented the hardware support for this feature starting from Pascal architecture as mentioned in [3].

Unified memory architecture (UMA) is a single memory address space that can be accessed by any GPU or CPU in the system. In order to work with managed memory, we can use CUDA programming model for NVIDIA GPUs; specifically *CudaMallocManaged API* [3].

Normally, when working with memory there is a page table where each entry in this tables is set internally by the driver for all pages that are covered by a certain memory allocation and that's how the system knows whether the pages are resident on the CPU or a GPU device side. Starting from NVIDIA Pascal architecture, NVIDIA implements a *page migration engine* which is a hardware support for page fault and migration. When allocating a managed memory area, it's not actually allocated unless it's going to be used by a GPU device or a CPU host [3]. If the CPU is going to initialize this specific managed memory area, this indicates that these pages are resident on the CPU memory address space. When an NVIDIA GPU device is invoked from a CPU runtime, the CUDA runtime doesn't automatically migrate these pages so there is no migration overhead when the kernel launches. When the kernel tries to access these pages, the page migration engine will detect a page fault, given that the pages already resident on the CPU memory space, and will automatically copy these pages to the GPU memory space which will result in that the GPU will stall the execution for the specific threads trying to access these pages (This is known as a migration overhead and hurts the performance badly, use only when necessary).

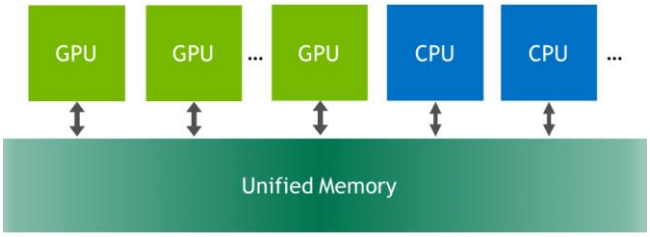

Fig. 7: Unified memory architecture [3]

The next section is diving into details how a certain GPU kernel shall be designed to use which memory and provides an example using 2 different versions for reduction sum kernel [5].

III. CUDA PROGRAMMING MODEL

In order to run operations on a GPU device, a CPU host shall invoke the GPU by calling a kernel that's implemented using a certain programming model i.e. CUDA or OpenCL.
This section discusses CUDA programming model and how to write and optimize GPU kernels via modifying memory access pattern and applying the GPU architecture knowledge from section I.

A GPU kernel is, from the CUDA programming model point of view, just a function that's going to be invoked from a CPU runtime and launched and executed N times in parallel by N different threads where all threads are following the SIMD execution model and SIMT management model described in section I. A GPU can launch a grid of threads where it can be 1-D, 2-D or 3-D grid and it's divided into blocks of threads where each block can represent finite number of threads i.e. 1024 threads per block. Number of threads per block and number of threads per warp can be known, i.e. in Fig. 8, by calling *cudaGetDeviceProperties API*.

```
Device Number: 0
    Device name: Tesla T4
    Memory Clock Rate (KHz): 5001000
    Memory Bus Width (bits): 256
    Number of totalGlobalMem 15843721216
    Number of sharedMemPerBlock 49152

    Number of warpSize 32
    Number of maxThreadsPerBlock 1024
    Number of maxBlocksPerMultiProcessor 16
    Number of multiProcessorCount 40
    Number of maxThreadsPerMultiProcessor 1024

    Number of maxThreadsDim 1024
    Number of maxGridSize 2147483647
    Number of totalConstMem 65536
    Number of sharedMemPerMultiprocessor 65536
    Peak Memory Bandwidth (GB/s): 320.064000
    asyncEngineCount: 3
```

Fig. 8: GPU details queried using CUDA cudaGetDeviceProperties. [9]

The following sub-section dives further into how to use the memory hierarchy and the GPU architecture knowledge into designing efficient kernels and analyze their performance.

Algorithms

The algorithm under discussion in this paper is a reduction sum kernel. In order to sum all elements in an array, we need to define a GPU reduction kernel where the operation used is summation. The first approach would be to launch a grid of threads where each thread will sum 2 consecutive elements in the array. This memory access pattern is called a *coalesced access pattern* where threads are reading consecutive chunks of memory back to back and hence increasing data locality; check Fig. 10 for a visual representation of what this kernel will try to achieve. This implementation, from the GPU architecture point of view, suffers from thread divergence where, for instance, thread 1, 3, 5, 7, 9 aren't active as the kernel runs only if the thread index is an even number. Hence, for a warp of size 32 threads, half the threads in this warp are idle and wasting power waiting for other threads to finish; check algorithm 1 for a grasp on how the kernel is designed to handle large arrays.

Here is a brief description of the algorithm steps [9]:
While the original array is in the global memory, the algorithm sequence is to:

1. Partition dataset into smaller chunks in shared memory by using thread blocks to load data into shared memory and then perform the rest of the algorithm.
2. Each thread block processes a chunk of these partitions, use reduction tree to summarize the results from each chunk into the final answer.
3. For extremely large arrays, a serial for loop is used to relaunch the reduction kernel and perform the same operation on the new sub-array.
4. Total sum of the entire array will be in index [0].

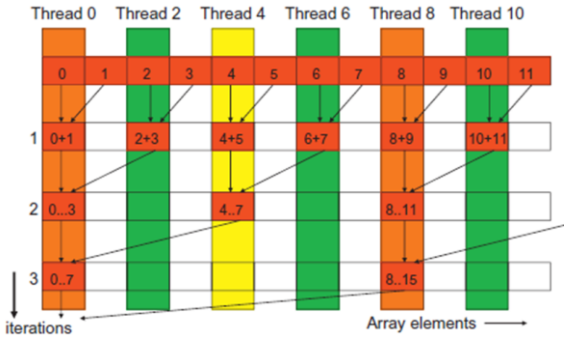

Fig. 9: Reduction sum - visual representation for algorithm 1 [5]

Algorithm 1 (Parallel Reduction, WT description)

Input: Matrix M_{exw}, it has total n elements; n = e*w

Output: S = $\bigoplus_{i=1}^{n} a_i$

Requirement: Make sure elements is in x direction only (1D dataset)

__shared__ float partialSum [1..n]

1. **forall** $i \in 1 : n$ **do**
2. partialSum [i] ← a_i
3. **enddo**
4. **for** stride ∈ 1 : BlockDim.x **do**
5. **forall** i ∈ $1 : n/2^{stride}$
6. partialSum [i] ← partialSum [2i - 1] ⊕ partialSum [2i]
7. **enddo**
8. **enddo**
9. S ← partialSum [1]
10. **return S**

Notes on algorithm 1 [5] [9]**:**

1. The above algorithm always assumes that number of threads in a block is equal to number of elements in dataset. i.e. 1024 elements -> use 1 Block with 1024 threads.

2. Only even threads of the block of threads hold the pair-wise partial sums after a single iteration.

3. Before starting another iteration, the stride we use to move to the next neighbor element is doubled and number of threads used is fewer than the 1st iteration. This means,

a. In the 1st iteration the even threads are taking the next neighbor ($\text{ThreadIdx.x} + 1$).
b. In the 2nd iteration, even threads are taking the 2nd next neighbor ($\text{ThreadIdx.x} + 2$)
c. In the 3rd iteration, even threads are taking the 3rd next neighbor. ($\text{ThreadIdx.x} + 4$)

4. After all iterations are done, $\text{ThreadIdx.x} = 0$ has the right to write the final sum value in partialSum.

This algorithm suffers from thread divergence, as in the 1st iteration number of threads working = $n/2$, in the 2nd iteration it's $n/2^2$ and in the 3rd iteration it's $n/2^4$. Number of threads executing the kernel decreases in half as number of iterations increases.

The second approach can be seen in Fig. 10, the algorithm is optimized as follow [5] [9]:

1. 1 thread loads 2 elements in shared memory instead of 1 element, in this case we can process double the number of threads of 1 block. i.e. 1024 threads in 1 block -> load 2048 elements in shared memory

2. Instead of adding neighbor elements in the 1st iteration, add elements that are half a section away from each other.

3. For extremely large arrays, there's a need to have multiple kernel calls.

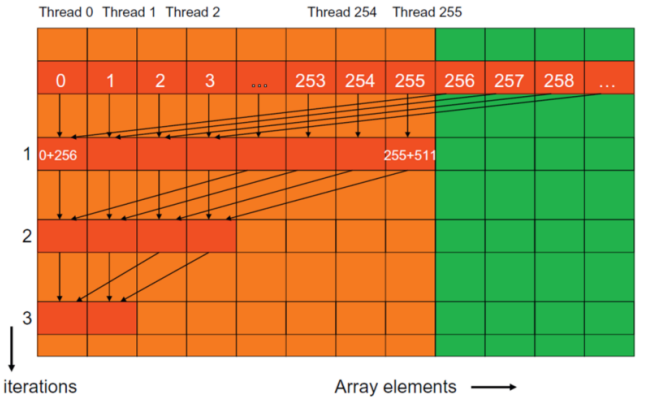

Fig. 10: Reduction sum - Visual representation for algorithm 2 [5]

The second algorithm doesn't suffer from thread divergence and utilize the usage of shared memory efficiently.

IV. Applications and Benchmarks

GPUs are widely used in different industries and solving many different applications i.e. machine learning and deep neural network training/inference, molecular visualization and analysis and in medical domain such as Magnetic resonance imaging (MRI). In [9], there is an implementation for a CUDA code model used for accelerating CNN inference on a GPU and the final application was detection for COVID-19 from CXR images that were taken for different patients' lungs.

Algorithm 2 (Parallel Reduction, WT description)

Input: Matrix M_{exw}, it has total n elements; n = e*w

Output: S = $\bigoplus_{i=1}^{n} a_i$

Requirement: Make sure elements is in x direction only (1D dataset)

Pre-defined: BlockDim.x = BlockSize

```
Const BlockSize := 16
__shared__ float partialSum [1..2 * BlockSize]
1.  while n ! = 1
2.      forall i ∈ 1 : n do
3.          partialSum [i] ← a_i
4.          partialSum [i + BlockSize] ← a_{i +BlockSize}
5.      enddo
6.      for stride ∈ BlockSize : 1 do
7.        forall i ∈ 1 : stride
8.            partialSum [i] ← partialSum [i + stride] ⊕ partialSum [i]
9.        enddo
10.     enddo
11.     S ← partialSum [1]
12.     n := ceil(n / (2 * BlockSize)
13. return S
```

In medical industry, GPUs are involved in applications like visual molecular dynamics, where biomolecular system can be analyzed and displayed. In medical computer vision applications, GPUs are used to accelerate the training of different computer vision models, i.e. SegNet and U-Net, which are used in semantic and instance segmentation applications i.e. segmenting teeth or upper and lower jaw from a DICOM image. There are numerous applications for medical computer vision applications which all require GPUs to get the result fast and with a very good accuracy i.e. Brain image scans are analyzed for brain tumors detection and ocular assessment which might help in the early detection of brain disorders i.e. Autism, Alzheimer and schizophrenia [10].

GPUs are an important component in supercomputer design i.e. frontier supercomputer [1] [11]. Benchmark tools are used in order to measure the performance of a super computer, and evaluate its performance and this results in assessing the overall system efficiency using measures like floating-point operations per second (FLOPS).

The following benchmark tests can be used for performance evaluation [12]:

- LINPACK (HPL, HPL-AI) – solves a dense system of linear equations.

- HPCG – simulate different memory access patterns, as memory access is one of the important bottlenecks that should be addressed carefully when designing a GPU kernel [5], along with heavy existing industrial computational patterns.

- DGEMM – measure performance for matrix multiplication as the matrix multiplication kernel is one of the main bottlenecks in GPU kernel software design [13] [14].

- FFT – measures the floating point rate of a double precision 1-D Discrete Fourier Transform.

V. Conclusion

Designing GPU kernels require understanding of GPU architecture, starting from the underlying execution model and ending with the memory model. GPUs are also a critical component in supercomputers design which represent a very large scale heterogeneous system i.e. Frontier supercomputer that can achieve exascale performance; one exaflop calculations per second. GPU applications rely heavily on very well optimized software that can utilize the underlying hardware for performance i.e. using supercomputers for very large scale applications.

References

[1] V. Rajaraman, "Frontier—World's first ExaFLOPS supercomputer," *Resonance*, vol. 28, no. 4, pp. 567-576, 2023.

[2] T. F. W. R. T. a. M. M. Aamodt, General-purpose graphics processor architectures, Morgan & Claypool Publishers., 2018.

[3] M. Harris, "Unified Memory for CUDA Beginners," NVIDIA, 19 June 2017. [Online]. Available: https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. [Accessed 1 May 2024].

[4] NVIDIA, "CUDA C++ Programming Guide," NVIDIA, 20 March 2024. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[5] D. a. W.-M. W. Kirk, Programming massively parallel processors: a hands-on approach, Morgan kaufmann, 2016.

[6] E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, "NVIDIA Tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39-55, 2008.

[7] W. W. Fung, I. Sham, G. Yuan and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient GPU control flow," in *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, 2007.

[8] G. Diamos, R. Johnson, V. Grover, O. Giroux, J. Choquette, M. Fetterman, A. Tirumala, P. Nelson and R. Krashinsky, "Execution of divergent threads using a convergence barrier". United states of America Patent 10,067,768, 2018.

[9] M. Elsayed, "Accelerating CNN inference on GPU using CUDA programming model written in C language," 2021. [Online]. Available: https://github.com/Accumulated/Accelerating-CNN-on-GPU-using-CUDA-C. [Accessed 2024].

[10] F. F. X. C. J. L. L. N. K. L. Z. C. Z. Q. A. S. Y. X. L. a. L. W. X. Li, "Computer vision for brain disorders based primarily on ocular responses," *Frontiers in Neurology*, vol. 12, p. 584270, 2021.

[11] C. Z. J. L. D. B. V. M. V. T. B. M. B. R. B. S. C. M. E. T. E. S. Atchley, "Frontier: Exploring Exascale," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2023.

[12] "HPC Challenge Benchmark," [Online]. Available: https://hpcchallenge.org/hpcc/. [Accessed 2024].

[13] P. Warden, "Why GEMM Is at the Heart of Deep Learning," 20 April 2015. [Online]. Available: https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/.

[14] A. Anderson, A. Vasudevan, C. Keane and D. Gregg, "Low-memory GEMM-based convolution algorithms for deep neural networks," arXiv, 2017.