VENTILATOR SENSOR FUSION AUGMENTED WITH CNN DETECTION FOR COVID-19

By

Taha Mahmoud Abdelmonaem Mahmoud

Mohamed Maged Khalil Mohamed

Mohamed Mahmoud Abdelmotaleb ElSayed

Mohamed Mostafa Mahrous Mohamed

Mostafa Karam Saeed Eissa

Nour Eldin Mohamed Sayed Tawfik

Under supervision of

Professor. Mohammed Reyad

Dr. Abdelrahman Abotaleb

A Graduation Project Report Submitted to
the Faculty of Engineering at Cairo University
in Partial Fulfillment of the Requirements for the
Degree of Bachelor of Science
in
Electronics and Communications Engineering

July 2021

Table of Contents

Chapter 7: CUDA implementation

7.1 Introduction to GPU

GPU provides higher instruction throughput and memory bandwidth than the CPU within a similar price and power envelope. Many applications leverage these higher capabilities to run faster on the GPU than on the CPU. Other computing devices, like FPGAs, are also very energy efficient, but offer much less programming flexibility than GPUs [15]. The difference between CPU and GPU is the design purpose of each of them. CPUs are designed to execute threads as fast as possible and can execute quite a few of them in parallel. However, GPUs are designed to execute 1000s of threads in parallel which results in greater throughput than CPUs [18].

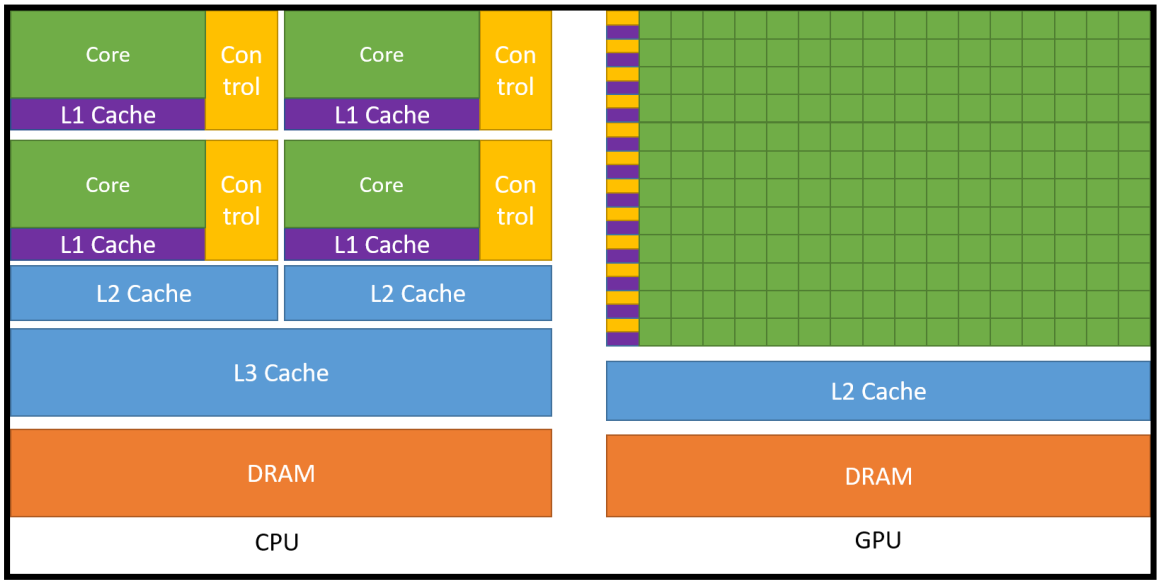

Figure 7-1: CPUs and GPUs have fundamentally different design philosophies.

Therefore, GPUs are specialized for high parallel computations and more data processing.

Systems are designed with a mix of GPUs and CPUs in order to maximize overall performance, so the program contains sequential and parallel parts to execute. A multi-threaded program has blocks of threads, each of them executes independently from each other where each thread has a sequence of operations to execute.

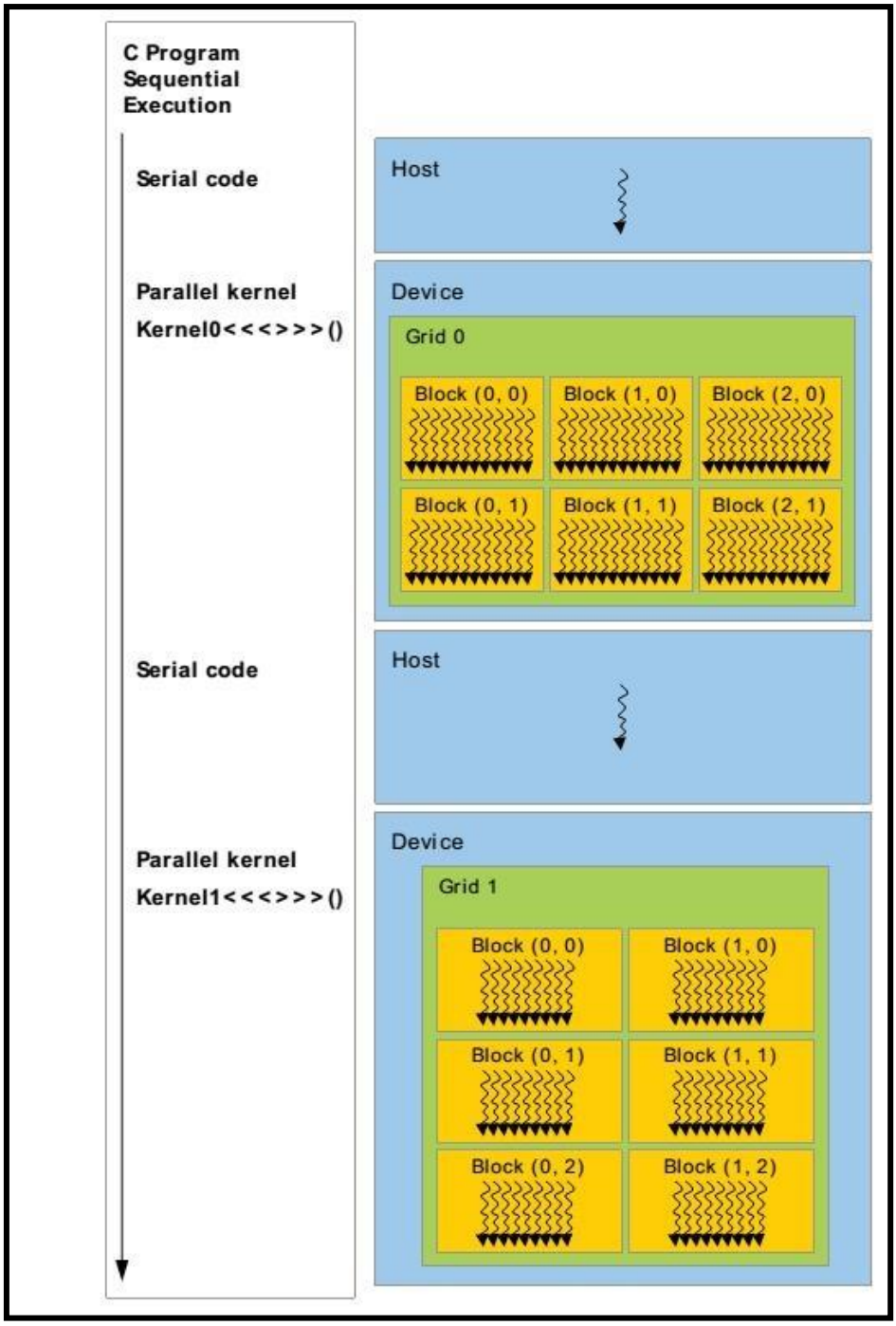

Figure 7-2: Execution of a CUDA program

A function in the programming model is called a kernel, when a kernel is called it's executed N times in parallel by N different CUDA threads. A kernel is defined using __global__ syntax and number of threads to execute this kernel is specified using <<< (Grid dimension), (block dimension) >>>. Each thread that executes the kernel has a **unique thread ID** and can be accessed within the kernel itself by using a built in variables; **ThreadIdx.x, ThreadIdx.y, ThreadIdx.z**.

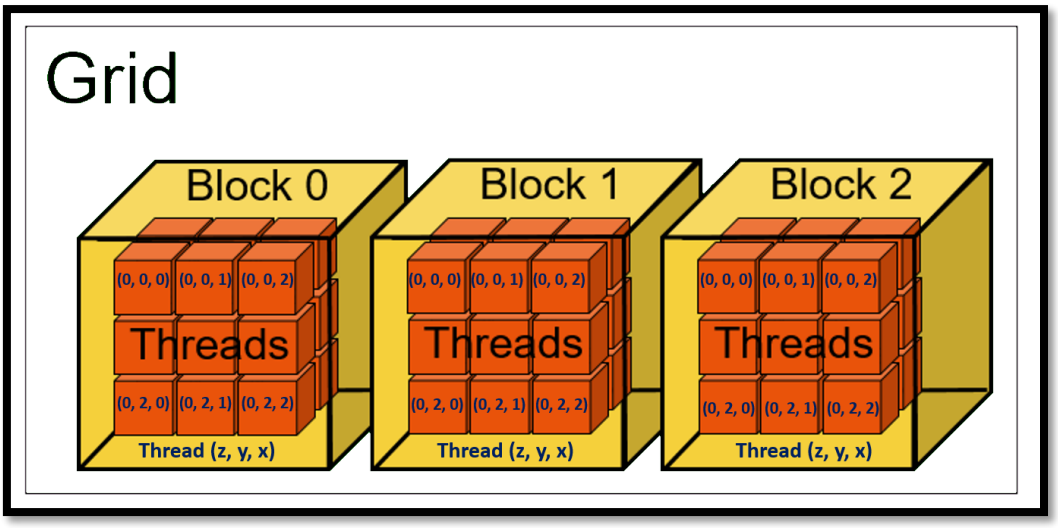

Figure 7-3: A multidimensional example of CUDA grid organization

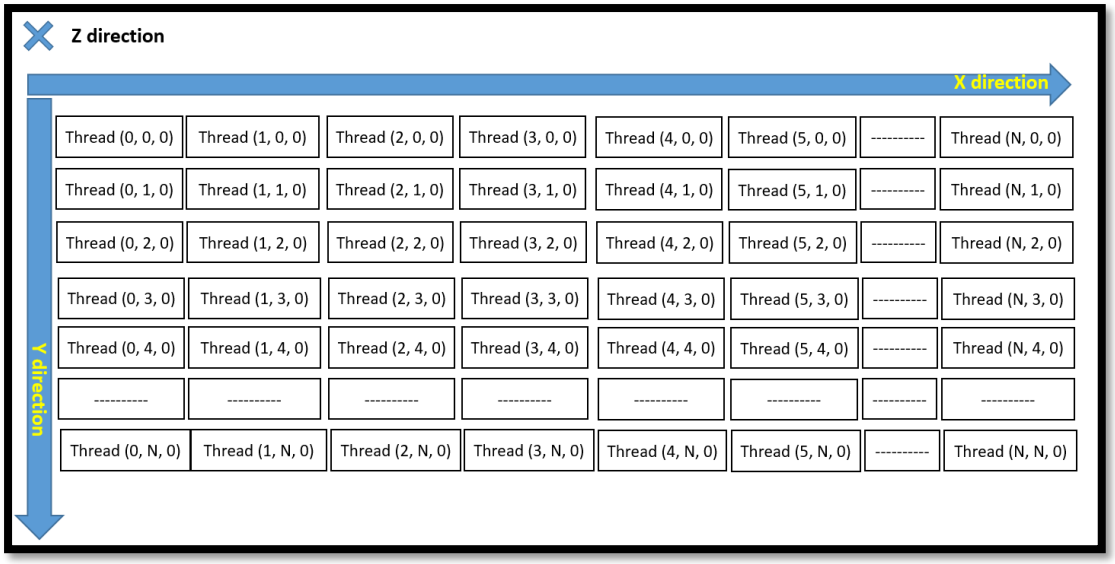

Figure 7-4: Threads distribution in X and Y directions

There's a limit to number of threads per block as they are in the same processor core and share a limited memory resource of that core. This number is 1024 threads total in 1 thread block. In order to execute a kernel, we need total number of threads:

$$\#threads_{total} = \#threads_{per\ block} * \#blcoks$$

Note: Thread ID relates to thread index using the following equation

$$Thread_{ID} = ThreadIdx.x + ThreadIdx.y * BlcokDim.x + ThreadIdx.z * BlockDim.x * BlockDim.y$$

Thread Blocks execute independently, they can execute in any order either parallel or sequentially. This allows blocks to be scheduled in any order across any number of cores. However, threads in the same block can cooperate by sharing data through shared memory and synchronize their execution to coordinate memory accesses using __syncthreads () which serves as a barrier all threads should wait for the slowest thread.

Cuda programming model assumes Cuda threads execute on a physically separate device that operates as a co-processor to the host assuming that both the host and the device maintain their own separate memory spaces DRAM. A program manages the global, constant and texture memory spaces visible to kernels through calls to Cuda runtime.

7.1.1 **Architecture of a modern GPU**

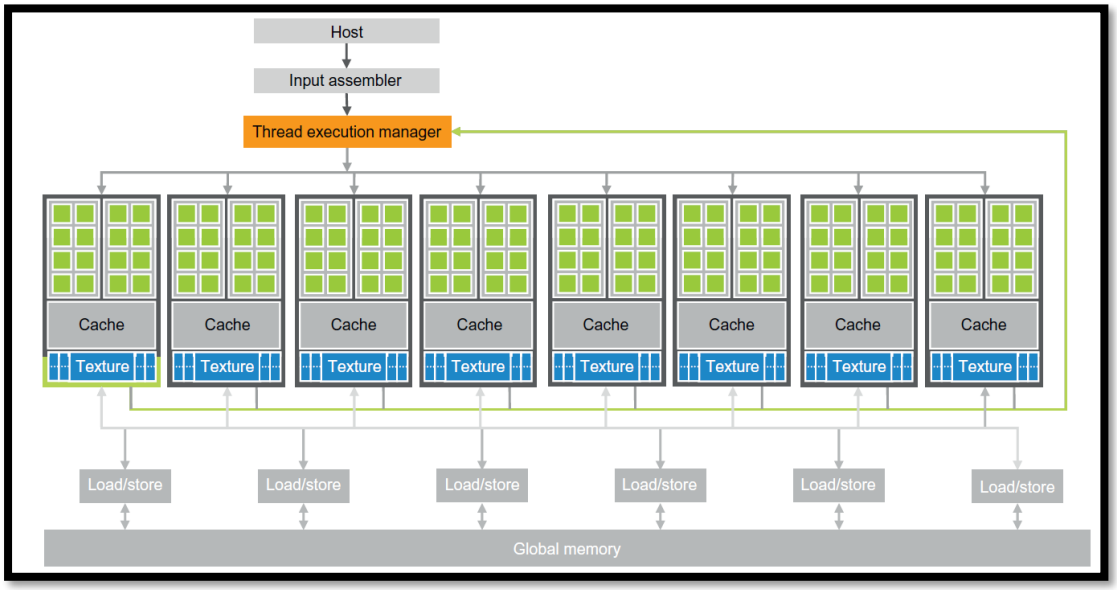

Figure 7-5: Architecture of a CUDA-capable GPU

Fig. 9.5 shows a high-level view of the architecture of a typical CUDA-capable GPU. It is organized into an array of highly threaded streaming multiprocessors (SMs). In Fig. 9.5, two SMs form a building block. However, the number of SMs in a building block can vary from one generation to another. Also, in Fig. 9.5, each SM has a number of streaming processors (SPs) that share control logic and instruction cache. Each GPU currently comes with gigabytes of Graphics Double Data Rate (GDDR), Synchronous DRAM (SDRAM), referred to as Global Memory in Fig. 9.5. These GDDR SDRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics. For graphics applications, they hold

video images and texture information for 3D rendering. For computing, they function as very high-bandwidth off-chip memory, though with somewhat longer latency than typical system memory. For massively parallel applications, the higher bandwidth makes up for the longer latency. More recent products, such as NVIDIA's Pascal architecture, may use High-Bandwidth Memory (HBM) or HBM2 architecture. For brevity, we will simply refer to all of these types of memory as DRAM. A good application typically runs 5000 to 12,000 threads simultaneously on a chip.

A streaming multiprocessor (SM) may run more than 1 block assigned by the GPU itself. However, 1 block runs only on the assigned SM. SMs run in parallel and independent on each other [18].

Cuda guarantees the following :

1. All threads in a block run on the SM at the same time
2. All blocks in a kernel finish before any blocks from the next kernel run

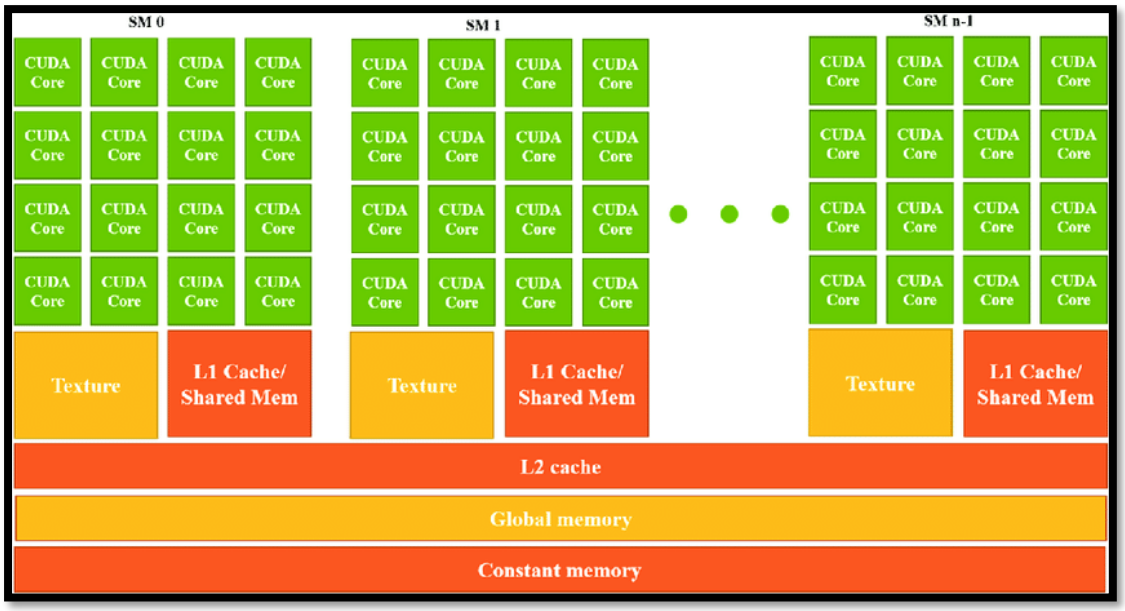

Figure 7-6: Schematic of NVIDIA GPU architecture, where SM refers to streaming multiprocessor

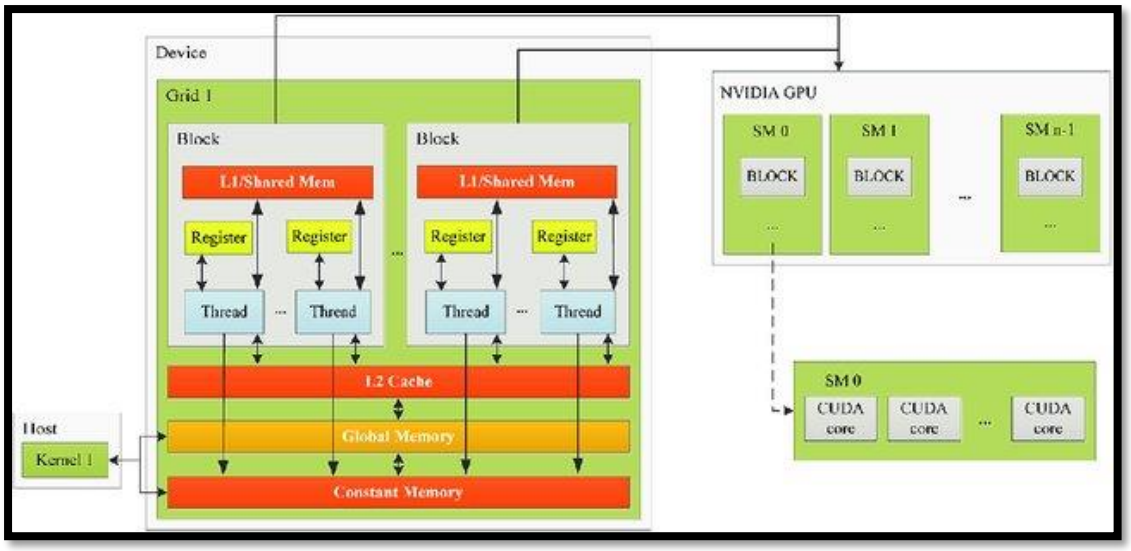

Figure 7-7: Schematic of the CUDA programming model

The main motivation for massively parallel programming is for applications to enjoy continued speed increase in future hardware generations. When an application is suitable for parallel execution, a good implementation on a GPU can achieve more than 100 times (100x) speedup over sequential execution on a single CPU core. If the application contains what we call "data parallelism," it is often possible to achieve a 10x speedup with just a few hours of work [18].

Speeding up real applications depends on the portion of the application that can be parallelized. If the percentage of time spent in the part that can be parallelized is 30%, a 100X speedup of the parallel portion will reduce the execution time by no more than 29.7%. The speedup for the entire application will be only about 1.4X. In fact, even infinite amount of speedup in the parallel portion can only slash 30% off execution time, achieving no more than 1.43X speedup. The fact that the level of speedup one can achieve through parallel execution can be severely limited by the parallelizable portion of the application is referred to as Amdahl's Law. On the other hand, if 99% of the execution time is in the parallel portion, a 100X speedup of the parallel portion will reduce the application execution to 1.99% of the original time. This gives the entire application a 50X speedup. Therefore, it is very important that an application has the vast majority of its execution in the parallel portion for a massively parallel processor to effectively speed up its execution [18].

Researchers have achieved speedups of more than 100X for some applications. However, this is typically achieved only after extensive optimization and tuning after

the algorithms have been enhanced so that more than 99.9% of the application execution time is in parallel execution. In practice, straightforward parallelization of applications often saturates the memory (DRAM) bandwidth, resulting in only about a 10X speedup. The trick is to figure out how to get around memory bandwidth limitations, which involves doing one of many transformations to utilize specialized GPU on-chip memories to drastically reduce the number of accesses to the DRAM [19].

7.1.2 Warp and latency

In the majority of implementations, a block assigned to an SM is further divided into 32 thread units called warps. The size of warps is implementation-specific. Warps are not part of the CUDA specification; however, knowledge of warps can be helpful in understanding and optimizing the performance of CUDA applications on particular generations of CUDA devices. The size of warps is a property of a CUDA device, which is in the warpSize field of the device query variable (dev_prop in this case). The warp is the unit of thread scheduling in SMs. Each warp consists of 32 threads of consecutive threadIdx values: thread 0 through 31 form the 1st warp, 32 through 63 the 2nd warp.

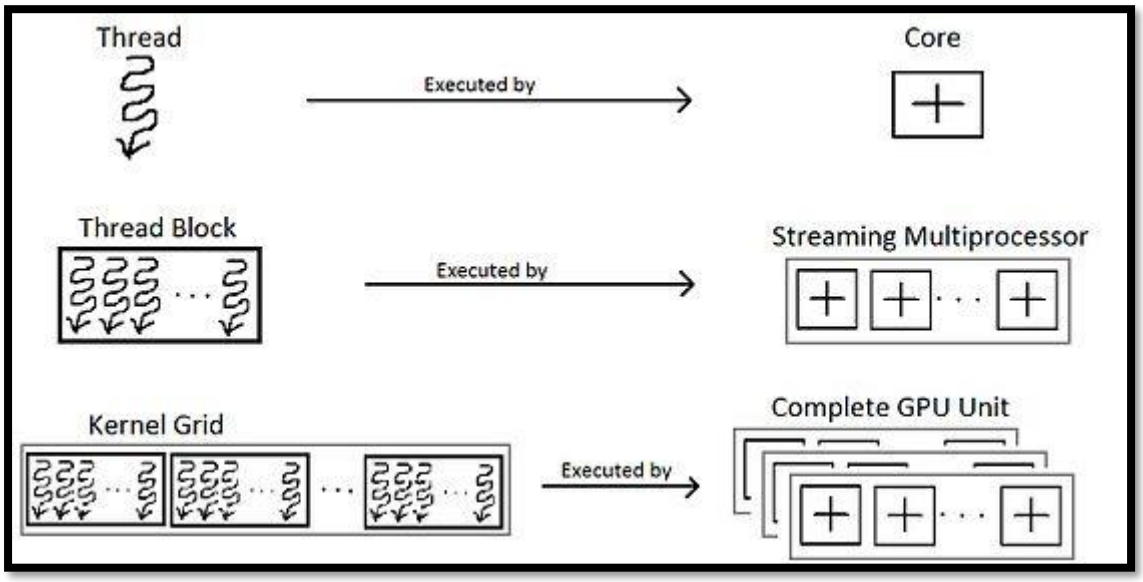

Figure 7-8: Thread blocks assigned to streaming multiprocessors

We can calculate the number of warps that reside in an SM for a given block size and a given number of blocks assigned to each SM. If each block has 256 threads, we can determine that each block has 256/32 or 8 warps. With three blocks in each SM, we

have 8 × 3 = 24 warps in each SM. An SM is designed to execute all threads in a warp following the Single Instruction, Multiple Data (SIMD) model — i.e., at any instant in time, one instruction is fetched and executed for all threads in the warp. This situation is illustrated with a single instruction fetch/dispatch shared among execution units (SPs) in the SM. These threads will apply the same instruction to different portions of the data. Consequently, all threads in a warp will always have the same execution timing.

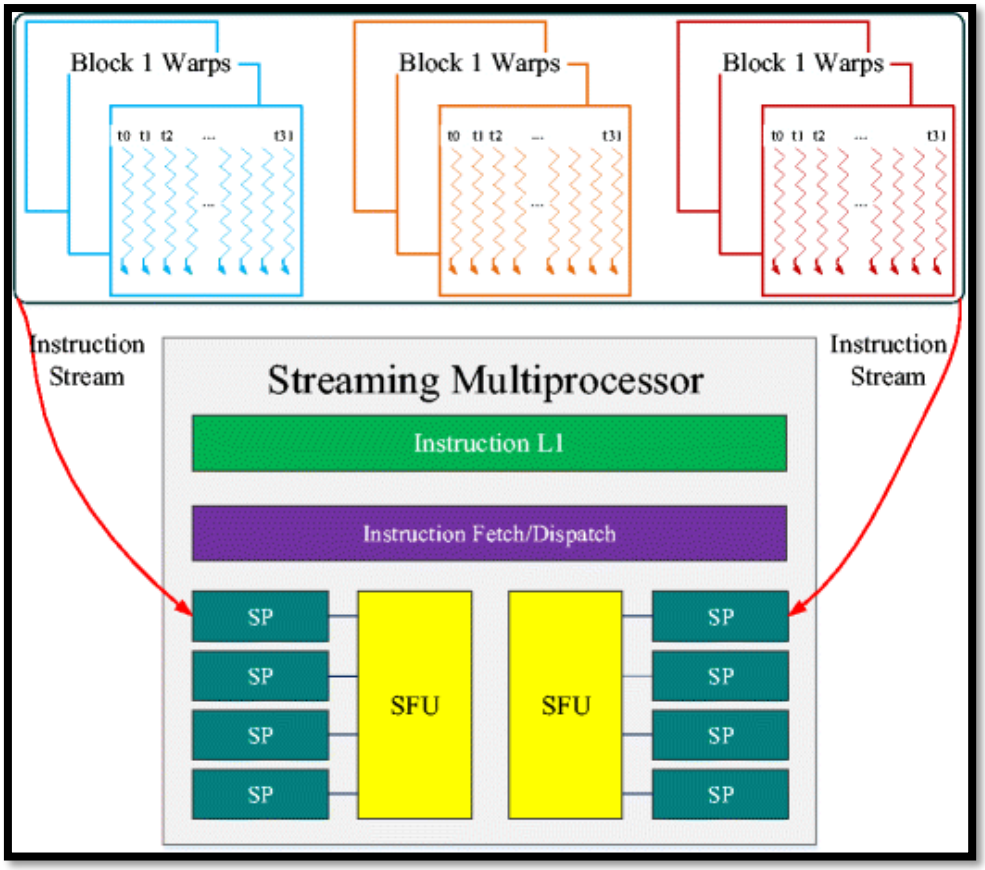

Figure 7-9: Thread block assignment to Streaming Multiprocessors (SMs)

In order to write efficient programs, maximize arithmetic intensity, where intensity $= \frac{Math\ Work}{Memory\ occupied}$. So, we need to maximize work per thread and/or minimize time spent on memory/thread. In order to minimize time spent on memory: Locate the most frequently accessed data to the fastest fast memory. The speed of different memories used in a system of CPU and GPU is sorted as follow, where Local memory is the fastest memory of all in accessing data and it can be seen in registers or caches.

$$Local\ memory\ >\ shared\ >>\ global\ >>\ CPU\ host\ memory$$

Efficient programs needs to consider the way memory addresses are accessed either a coalesced or strided access. GPU most efficient when threads read or write contiguous chunk of memory location. It's better than a strided access. Example of a strided: $g[i * 2]$, Example of a coalesced: $g[i] and\ g[offset\ +\ i]$.

Optimization performance goal is to maximize useful computations/sec, minimize time waiting at barriers, and minimize thread divergence.

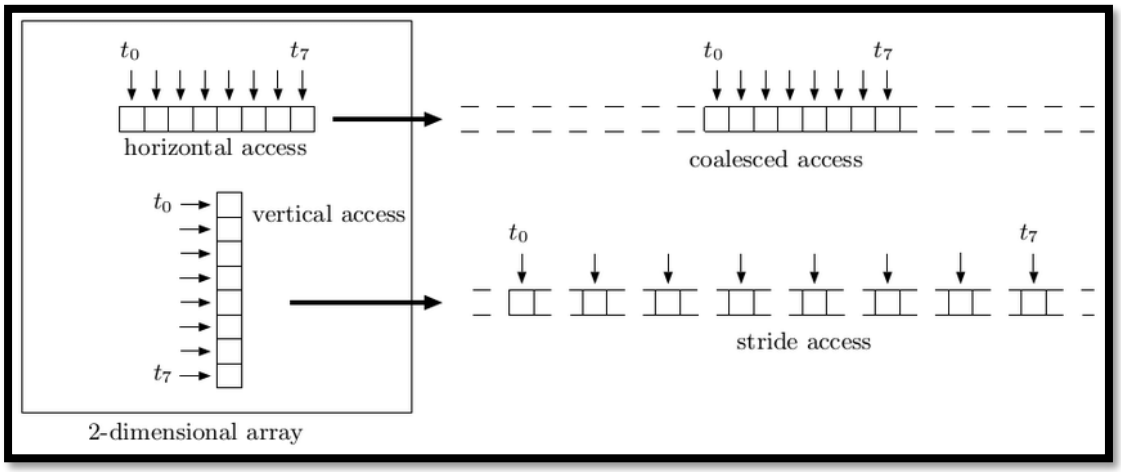

Figure 7-10: Coalesced and strided access patterns

Always avoid thread divergence: It can be seen at branches and loops.

1. If (condition) { some code }; else {some other code}
2. (Pre_loop) loop (Post loop): threads wait for others to finish

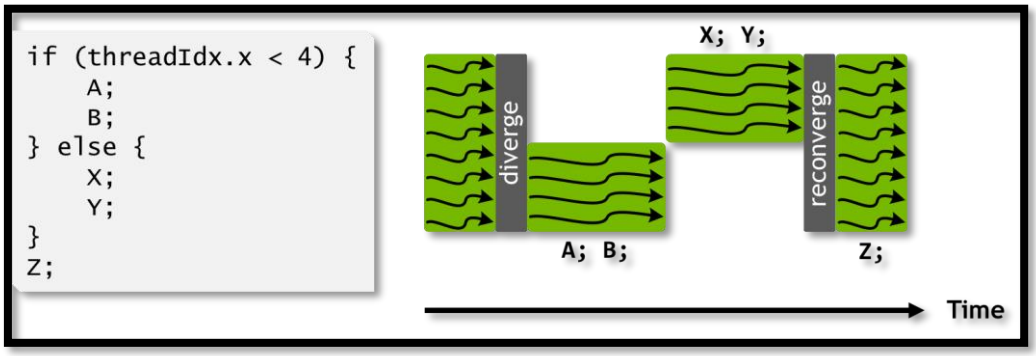

Figure 7-11: Thread divergence

7.1.3 Parallel communication patterns

There are certain common parallel algorithms known to be used in different problems. The common patterns are mapping, gathering, scattering and stencil.

1. **Mapping (1 to 1):** 1 thread reads 1 element only

2. **Gathering (Many to 1):** 1 thread reads 1 element and gather its value with the neighboring elements and then writes its value again to the first element i.e., taking average of 3 neighboring elements
3. **Scattering (1 to many):** 1 thread reads 1 element and scatter its value to its neighbors
4. **Stencil (several to 1):** Threads read elements of input array in a fixed pattern i.e., 2D Moore, 2D von Neumann and 3D von Neumann.
5. **Reduce (all to 1):** i.e., summing all elements in an array
6. **Scan / Sort (all to all)**

In order to measure how well the algorithm is doing, we define 2 metrics, step complexity and work complexity. Step complexity is how long it takes to complete a computational operation, while work complexity is total amount of work it took to perform that computation. In algorithms, we always think about step complexity.

7.1.4 Scan algorithm

Scan algorithm takes a list of numbers as an input and does a certain operation i.e. addition, however we need to define an identity element for each operation.

Table 7-1: Different identity values for scan algorithm

Operation	Identity	Why?
+	ϕ (zero element)	$\phi + a = a$
Minimum (on unsigned chars)	ϕxFF	Min($\phi xFF, a$) $= a$
Multiply	1	$a\ x\ 1 = a$
Logical Or	False	A || false = A
Logical And	True	A && true = A

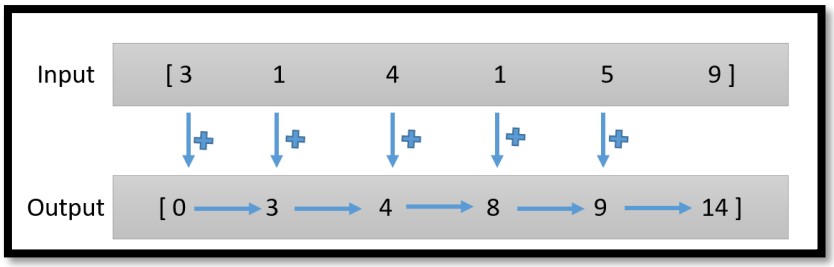

Figure 7-12: Sum scan example

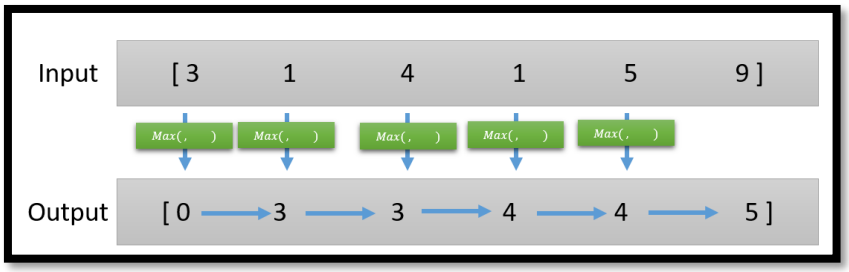

Figure 7-13: Max scan example

In serial implementation of scan algorithm with addition operation, we have 2 types of scan. It's either inclusive or exclusive scan algorithm. In the serial implementation of both algorithms, they have W (n) = S (n) = O (n)

Serial implementation of exclusive scan, first define the identity element and iterate over the length of the input array. For each iteration, update the output of the current index I, Accumulate the output of operation of adding the element of index i and the identity element.

Figure 7-14: Exclusive scan example

Serial implementation of inclusive scan, first define the identity element and Iterate over the length of the input array. For each iteration, accumulate the output of operation of adding the element of index i and the identity element and update the output of the current index i.

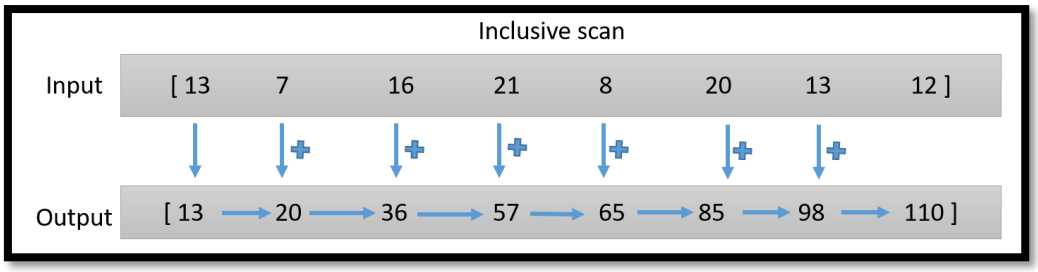

Figure 7-15: Inclusive scan example

Parallel scan algorithms:

Table 7-2: Parallel scan algorithms

	Step efficient	**Work efficient**
Hillis & Steele	More efficient	
Belelloch		More efficient

1. Hillis and steel inclusive scan:

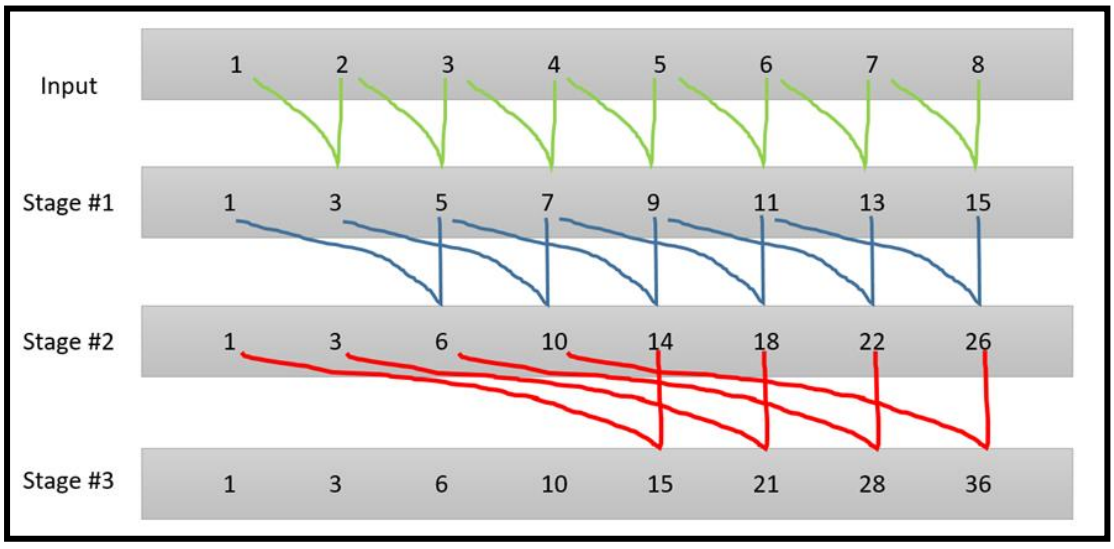

Figure 7-16: Hillis and steel inclusive scan algorithm in action

- Starting with step #0, When reaching step #i, add yourself to your 2^i left neighbor and If no neighbor that far to the left just copy yourself to the same location
- **W (n) = O ($n \log_2 n$) and S (n) = O ($\log_2 n$)**

2. Blelloch scan: Exclusive scan

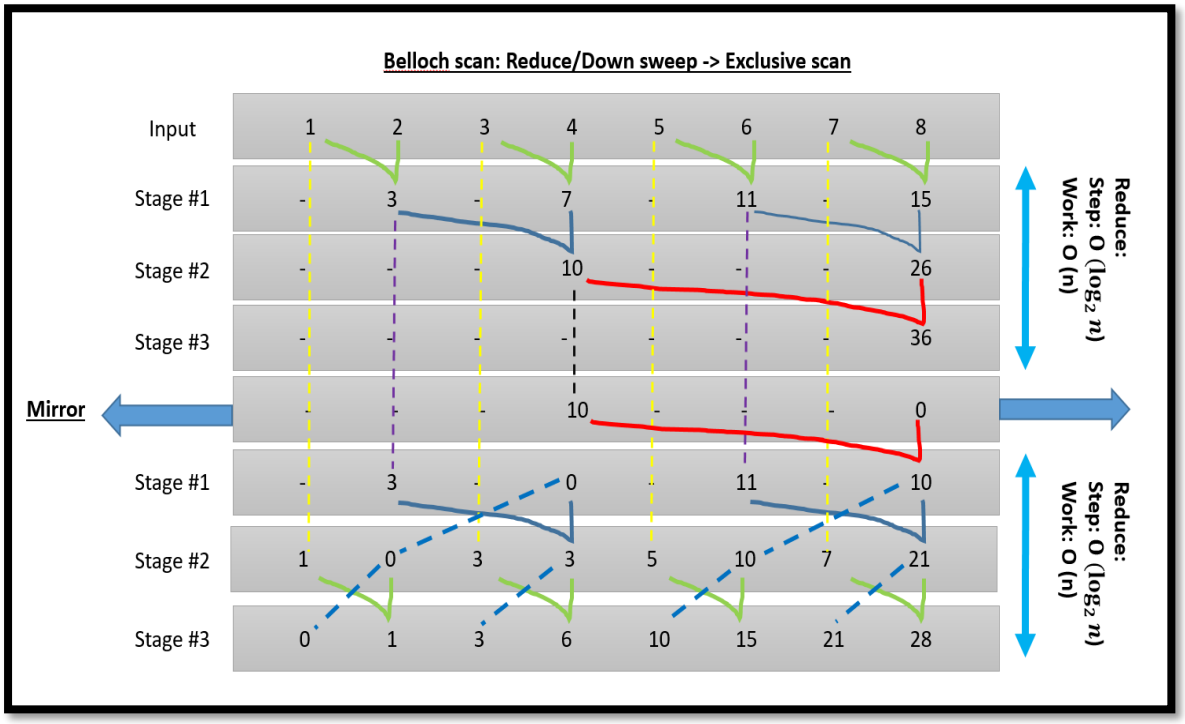

Figure 7-17: Belloch scan: Reduce/Down sweep (exclusive scan)

Summed elements will be paired to the items to its left. **Total work complexity = O (2n), total step complexity = O (2 $\log_2 n$)**

7.2 Algorithms implemented in the code

7.2.1 Sum Reduction algorithm

A reduction operation is defined as partitioning and summarizing a set of input values into one value. The reduction operation can be max, min, sum, etc. In general, it has a condition for the dataset to be associative and commutative in order to be applicable for a given dataset [20].

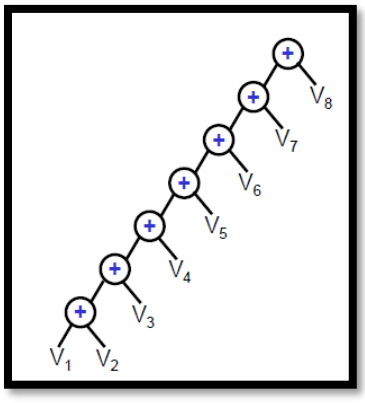

Figure 7-18: Serial sum algorithm

7.2.1.1 Serial implementation

The sequential implementation sequence is to iterate through the input and perform the sum reduction operation between the result value and current input value. This results in N operations for N values

- Work complexity is defined as total number of operations the algorithm performs, which in this case = $O(n)$.
- Step complexity is defined as total number of steps the algorithm executes, which in this case = $O(n)$.

Algorithm 1 (Sequential 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)

1. **for** i ∈ 1 : n **do**
2. S := S + a_i
3. **enddo**
4. **return S**

7.2.1.2 Parallel implementation

Parallel reduction algorithm performs same number of operations as in sequential algorithm, however in fewer number of steps which in this case $\log_2(n)$; where n is the problem size.

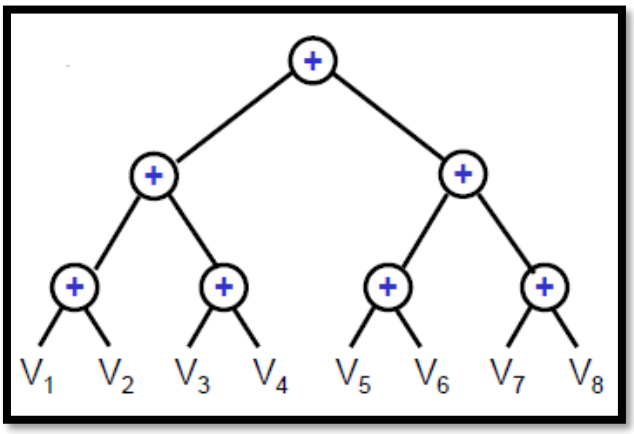

Figure 7-19: Parallel sum algorithm

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.

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)

1. __shared__ float partialSum [1..n]

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

Notes on the algorithm:

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.
    a. This means,
        i. In the 1st iteration the even threads are taking the next neighbor (ThreadIdx.x + 1).

ii. In the 2nd iteration, even threads are taking the 2nd next neighbor (ThreadIdx.x + 2)

iii. In the 3rd iteration, even threads are taking the 3rd next neighbor. (ThreadIdx.x + 4)

4. After all iterations are done, only ThreadIdx.x #0 has the right to write the final sum value in partialSum [0].

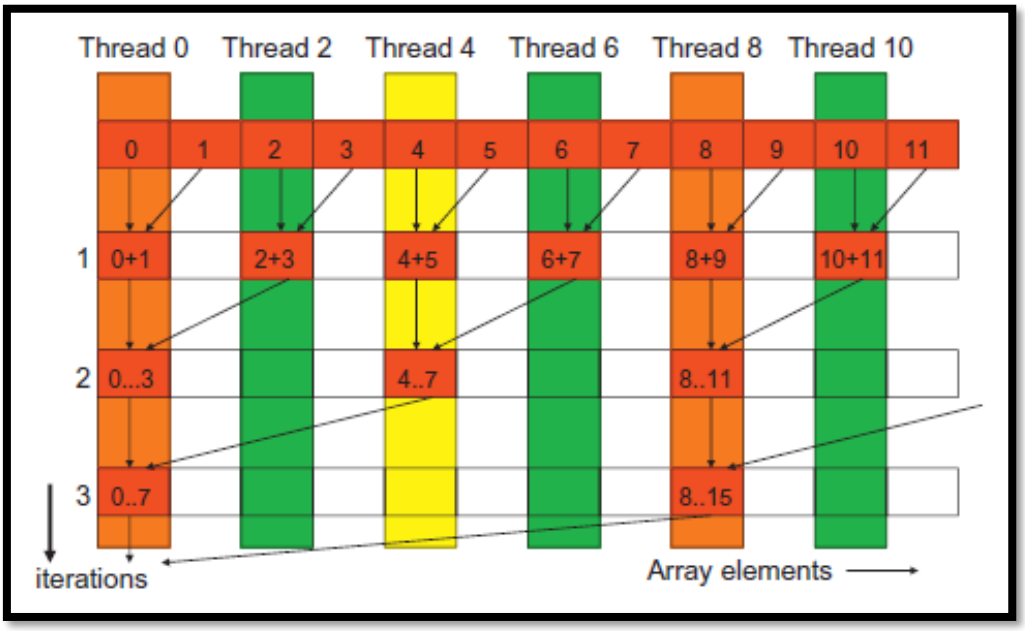

Figure 7-20: Parallel example for reduction sum

- An algorithm is said to be work efficient if it has same number of operations as the sequential algorithm, so this parallel implementation is work efficient with $W(n) = O(n)$.
- Step complexity for this algorithm is $S(n) = O(\log_n(n))$.

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 $\frac{1}{2}$ as number of iterations increases [22].

7.2.1.3 Final optimized reduction sum algorithm

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.

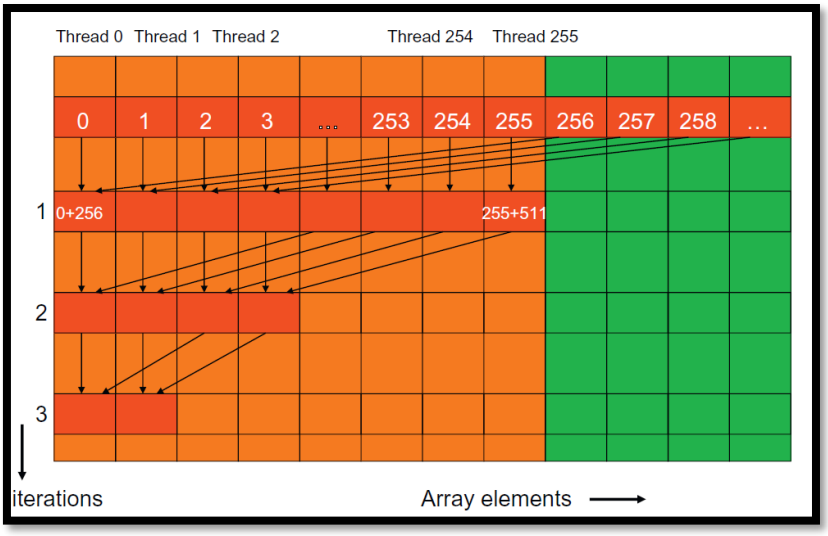

Figure 7-21: Example on Final optimized reduction sum

Cuda C Implementation of Reduction kernel:

```
__global__ void BN_Kernel_Mean_Reduction(float *input, int H1, int W1, int D1,  float *Mean, int W2)
{
   /*
      This code works on 2 * Block_Size elements.
      i.e. for 512 Block_Size -> we are reducing 1024 elements.
      Each thread loads 2 elements, one at tx and the
      other shifted by blockIdx.x.
   */

   __shared__ float partialSum[2 * BLOCK_SIZE];
   float tmp = 0;

   unsigned int tx = threadIdx.x;
   int bx = blockDim.x;

   int by_index = blockIdx.y;
   int bx_index = blockIdx.x;

   // The start variable is to get offset for input matrix in loading
   unsigned int start = blockIdx.x * (2 * blockDim.x);
   int start_yDir = blockIdx.y * W1;

   if (start + tx < W1 && start_yDir < H1 * W1)
      // Load 2 elements in the shared memory
      partialSum[tx] = input[start + tx + start_yDir];
   else
      partialSum[tx] = tmp;

   if (tx + bx + start < W1 && start_yDir < H1 * W1)
      partialSum[bx + tx] = input[start + bx + tx + start_yDir];
   else
      partialSum[bx + tx] = tmp;

   unsigned int stride = 0;

   __syncthreads();

   for (stride = blockDim.x; stride > 0; stride = stride / 2)
   {
      __syncthreads();
      if (tx < stride)
         partialSum[tx] += partialSum[tx + stride];
   }
   __syncthreads();

   if (tx == 0)
      Mean[bx_index + by_index * W2] = partialSum[tx];

}
```

Final optimized Reduction sum algorithm using WT description:

Algorithm 3 (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

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

7.2.2 **GEMM algorithm**

General matrix multiplication algorithm is a used to represent covolutional layer operation using highly optimized matrix multiplication kernel algorithm. The central idea is unfolding and replicating the inputs to the covolutional kernel such that all elements needed to compute one output element will be stored as one sequential block. This technique will reduce the forward operation of the convolutional layer to one large matrix – matrix multiplication. This algorithm has many different approaches that differes in the additional space needed for the input unrolling operation, however all performs same number of operations.

In this algorithm the input has shape $C_{input}\ x\ H_{input}\ x\ W_{input}$, where C_{input} is number of input feature maps. The convolution filters are $M\ x\ C_{input}\ x\ K\ x\ K$, where M is the number of output feature maps. Filter banks are total number of filters used in a covolution operation which is equal to $M\ x\ C_{input}$, where a single filter bank has $K\ x\ K$ number of weights.

Consider the following convolutional layer:

- Input has 3 input feature maps each of size 3 x 3
- Output has 2 output feature maps each of size 2 x 2
- This layer is using total of $M\ x\ C$ total number of filter banks
- These 6 filter banks each has $2\ x\ 2$ weights.

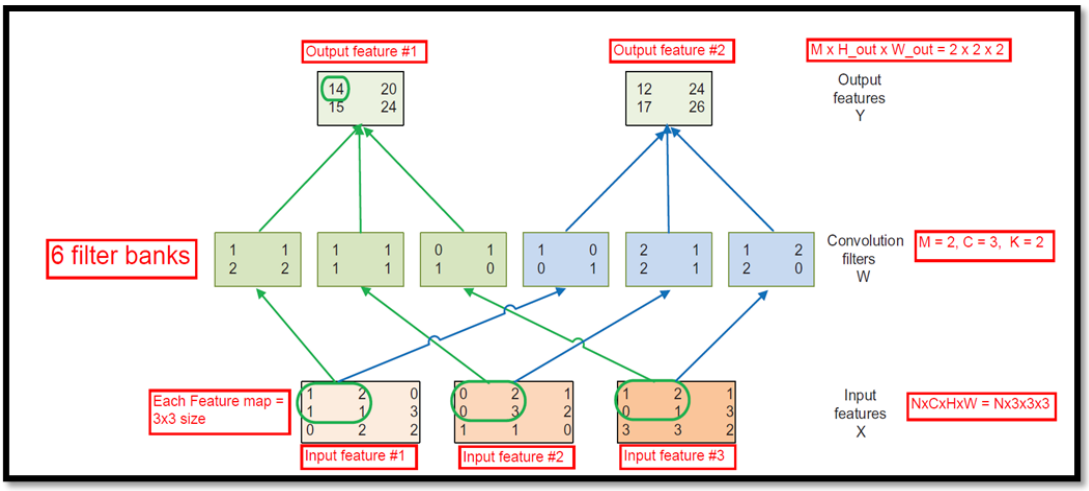

Figure 7-22: Reduction of a convolutional layer to GEMM

In order to apply the GEMM algorithm, there is a need to construct 2 matrices, input features ($X_{unrolled}$) and filter-bank matrix involves all the current convolution filters.

7.2.2.1 The unrolling algorithm

This algorithm generates an unrolled matrix. The conceptual design for this algorithm is that the input features can be concatenated into on large matrix, where each row of this matrix contain all input values necessary to compute one element of an output feature. From the definition of a convolution filter movment, stride and convolution operation of dot product, the unrolling algorithm results in replicating each input element multiple times [19].

The replication process depends on the location of the input element, to illustrate this we use the previous example to get number of replication:

- From the movement of 1 filter across 1 input feature map:
    - The center element is used 4 times
    - The central element on each edge is used 2 times.
    - The 4 elements at the corners of each input feature are used only once and will not need to be replicated
    - Total number of elements =
        - 1 center element * 4 times +
        - 4 central edge elements * 2 times +
        - 4 corner elements * 1 time = 16 elements.
        - For 3 input features, we need $3 3\ x\ 16 = 48$ elements.
    - This indicates that we need $O\left(K^2 C_{input} H_{input} W_{input}\right)$ additional space.

The size of unrolled input in this algorithm is as follow:

- $Number\ of\ rows = C_{input}\ x\ K\ x\ K$
- $Number\ of\ cols =\ H_{out}\ x\ W_{out}$
- $Number\ of$ Ouput feature maps doesn't affect the duplication in the unrolling algorithm.

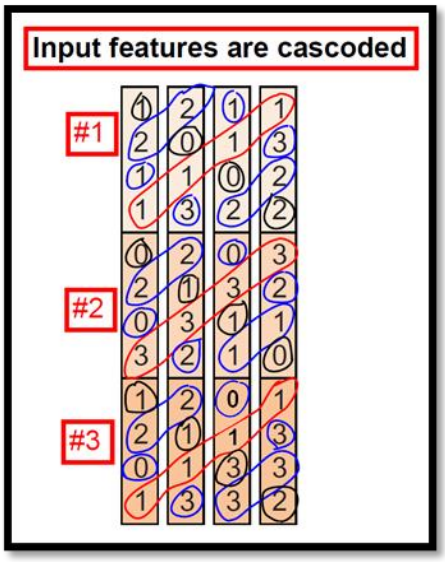

Figure 7-23: Unrolled input features

Algorithm 4 (Unrolling image, WT description)

Input: Matrix M_{exw}

Output: $M_{unrolled}$

1. **for** $local_{row} \in 1 : Filter_{Height}$ **do**
2. **for** $local_{col} \in 1 : Filter_{Width}$ **do**
3. **forall** i ∈ 1 : Tile **do**
4. $depth_{offset} := M_{unrolled\ width} * Filter_{Height} * Filter_{Height}$
5. $index_1 := local_{col} * M_{unrolled\ width} + local_{row} * M_{unrolled\ width} * Filter_{height} + i + depth_{offset}$
6. $index_2 := local_{row} * Input_{width} + local_{col} + Input_{height} * Input_{width}$
7. $M_{unrolled}[index] \leftarrow M[index2]$
8. **enddo**
9. **enddo**
10. **enddo**

Each CUDA thread will be responsible for gathering $k * k$ input elements from 1 input feature map. The convolution algrithm is now reduced to a matrix multiplication algorithm between filter-bank matrix and unroller input.

CUDA C implementation for input unrolling kernel:

```
__global__ void INPUT_UNROLLING(int stride, int Filter_Height,
                                float *Input, int H1, int W1, int D1,
                                float *X_unrolled, int H2, int W2, int D2,
                                int Output_Height, int Output_Width)
{
   int bx = blockIdx.x, by = blockIdx.y, bz = blockIdx.z;
   int tx = threadIdx.x, ty = threadIdx.y;

   // Select row and column values
   int row =  by * TileDW + ty;
   int col = bx * TileDW + tx;
   int depth = bz;
   int col_no_strided = col, row_no_strided = row;
   int depth_offset = depth * W2 * Filter_Height * Filter_Height;

   /*
     Note for bx, by and bz= 0, stride = 2:
        @ tx = 0, ty = 0 -> First multiply the col * stride, row * stride; = 0, 0
                    you are shifting in x direction using local col
                    you are shifting in y direction using local row;
        @ tx = 1, ty = 0 -> First multiply the col * stride, row * stride; = 2, 0
                    you are shifting in x direction using local col
                    you are shifting in y direction using local row;
        @ tx = 0, ty = 1 -> First multiply the col * stride, row * stride; = 0, 2
                    you are shifting in x direction using local col
                    you are shifting in y direction using local row;
   */

   col *= stride; row *= stride;

   // Limit number of threads
   if (row_no_strided < Output_Height && col_no_strided < Output_Width && depth < D1)
   {
      // Each thread unrolls k x k elements
      for (int local_row = 0; local_row < Filter_Height; local_row++)
      {
         for (int local_col = 0; local_col < Filter_Height; local_col++)
         {
            // 1. local row and column shifts affect the locations in Unrolled matrix
            // 2. For each col and row non strided values you are adding an offset to columns and rows in Unrolled matrix
            // 3. Offset the depth using "depth_offset" variable
            X_unrolled[local_col * W2 + local_row * Filter_Height * W2 + col_no_strided +
                              row_no_strided * Output_Width + depth_offset]  =
            Input[(row + local_row) * W1 + (col + local_col) + depth * H1 * W1];
         }
      }
   }
}
```

The filter-bank matrix is designed in a linearized arrangement of all filter banks, such that each row contains all weight values needed to produce one output feature map. The height of the filter-bank matrix is the number of output feature maps (M), this allows the output feature maps to share a single expanded input matrix. The width of the filter-bank matrix is the number of weight values needed to generate each output feature map element [19].

The filter-bank matrix has dimesnios ($M\ x\ (C\ *\ k\ *\ k)$. From the last example, the filter-bank matrix is as present in figure 9-24.

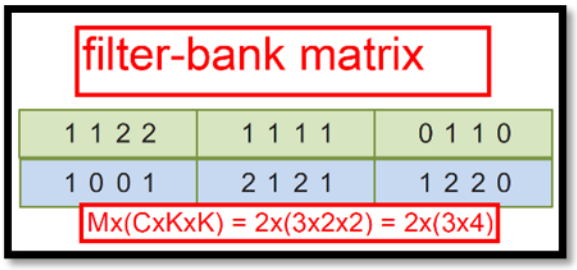

Figure 7-24: Filter-bank matrix

Now we can proceed to simply multiply the filter-bank Matrix ($M\ x\ (C_{input}\ *\ k\ *\ k)$ by unrolled input matrix ($C_{input}\ *\ K\ *\ K)\ x\ (H_{out}\ *\ W_{out}$) and the result is the output feature maps matrix ($M\ x\ (H_{out} * W_{out})$).

7.2.3 Matrix multiplication algorithms

Multiplying 2 matrices: $M_{ixj}\ x\ N_{jxk} = P_{ixk}$. Each element of the output matrix P is considered an inner product of a row of M and column of N.

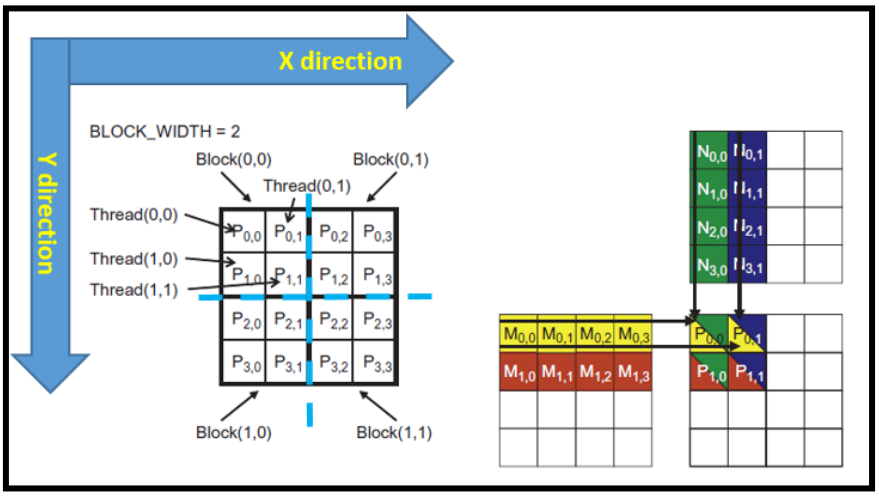

Figure 7-25: Matrix multiplication actions of one thread block

$$P_{Row,Col} = \sum M_{Row,k}\ x\ N_{k,Col}\ ; k = 0, 1, \ldots width - 1$$

Mapping threads indices to access the correct output elements in matrix P is done using the following formulas:

$$Row = threadIdx.y + blockIdx.y * blockDim.y$$

$$Col = threadIdx.x + blockIdx.x * blockDim.\text{x}$$

The result of Row and Col variables is the location of output element.

In parallel matrix multiplication, each thread runs the kernel and calculates Row and Col variables independent on other threads as they are stored in local registers.

A thread calculates a single output element in matrix P by using a loop that reads a whole row of Matrix M and a whole column of Matrix N and writes the values in the corresponding location in P matrix [20].

Algorithm 5 (Parallel Matrix multiplication, WT description)

Input: Matrix M_{exw}, Matrix N_{wxz}

Output: Matrix P_{exz} = $\prod\ (M, N)$

1. **forall** Row ∈ $1 : M\ rows$, Col ∈ $1 : N\ columns$ **do**
2. **for** k ∈ 1 : M columns **do**
3. Pvalue ← M[Row x (M columns) + k] x N [k x (N columns) + Col]
4. **enddo**
5. P[Row x P columns + col] ← Pvalue
6. **enddo**

Accessing global memory addresses is slow, however shared memory is faster but smaller than global memory. The matrix multiplication algorithm is extended to be a tiled algorithm. Tiling means to partition data into subsets called tiles and each of these tiles fits into the shared memory. Tile means that a large area is covered by tiles which

in this case blocks of threads. To make sense of how tiling can be applied to matrix multiplication, we need to analyze matrix multiplication algorithm:

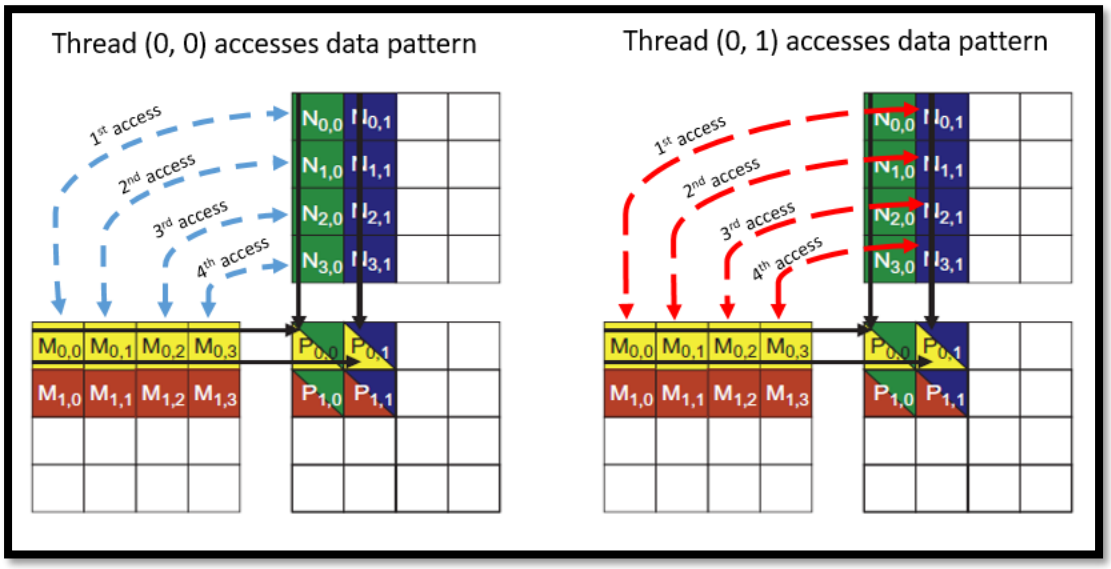

Figure 7-26: A small example of matrix multiplication - Global memory accesses performed by threads

Thread (0, 0) access pattern indicates that it reads 8 elements in 4 accesses:

- (M_{00}, N_{00}), (M_{01}, N_{10}), (M_{02}, N_{20}), (M_{03}, N_{30});

Thread (1, 0) access pattern indicates that it reads 8 elements in 4 accesses as well:

- (M_{00}, N_{01}), (M_{01}, N_{11}), (M_{02}, N_{21}), (M_{03}, N_{31});

The 2 threads read the same 4 elements from M matrix from global memory, this gives the tiling algorithm an advantage to load the elements first in the shared memory so it's loaded from global memory 1 time only. This causes a radically reduction in total number of global memory accesses. The amount of total reduction depends on block size used in tiling algorithm. If using a block size of 16 x 16 then the global memory traffic can be reduced to $\frac{1}{16}$. Tiling localizes the memory locations accessed among threads and the timing of their accesses, it divides the long access sequences of each thread into phases and uses barrier synchronization to keep the timing of accesses to each section at close intervals. So again, the basic idea for a tiled matrix multiplication algorithm is for threads to collaborate to load subsets of M and N elements into shared memory before using these elements in multiplication. Algorithm analysis: In each phase, all threads in a block collaborate to load a tile of M and a tile of N into shared memory. This indicates that every thread loads 1 element from M and 1 from N matrices. The dot product calculations performed by each thread are now divided into phases [23].

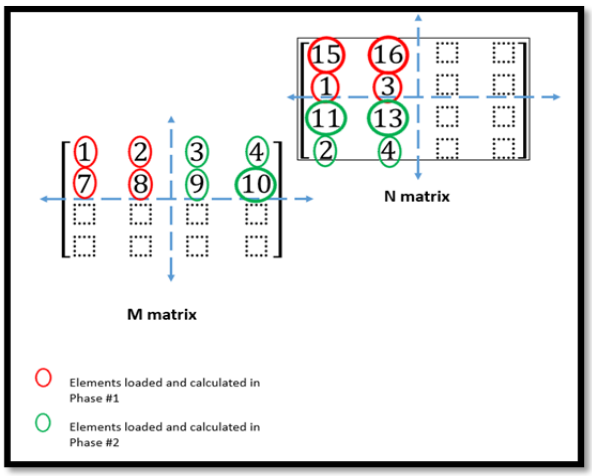

Figure 7-27: Tiling phases for loading and calculating subset of matrix multiplication

By using 4 blocks each has 2x2 threads, each thread calculates 1 output element

We have 2 phases:

In phase #1:

- 1st step: All threads collaborate to load certain elements from both matrices.
- 2nd step: threads calculate the partial multiplication output
  - 1*15 + 2*1 = Partial_0
  - 1*16 + 2*3 = Partial_1
  - 7*15 + 8*1 = Partial_2
  - 7*16 + 8*3 = Partial_3

In phase #2:

- 1st step: All threads load certain elements different from the elements loaded from phase #1
- 2nd step: Multiplication accumulates and get the other partial multiplication output accumulated with the one calculated from the 1st phase. The result is that we have a final output element by the end of the 2nd phase
  - Partial_0 + 3*11 + 4*2 = $Final\ output_0$
  - Partial_1 + 3*13 + 4*4 = $Final\ output_1$
  - Partial_2 + 9*11 + 10*2 = $Final\ output_3$
  - Partial_3 + 9*13 + 10*4 = $Final\ output_4$

Final Matrix multiplication kernel using CUDA C:

```
__global__ void MatrixMulKernel(float *M, int H1, int W1, int D1, float *N, int H2, int W2, int D2,
                                float *P, int H3, int W3, int D3, int num_blocks, int activation,
                                int IS_BIASED, Matrix bias_mat)
{
  __shared__ float Mds[Tile_GEMM][Tile_GEMM];
  __shared__ float Nds[Tile_GEMM][Tile_GEMM];

  int bx = blockIdx.x;  int by = blockIdx.y;  int tx = threadIdx.x;  int ty = threadIdx.y;
  // Identify the row and column of the d_P element to work on
  int Row = by * Tile_GEMM + ty;  int Col = bx * Tile_GEMM + tx;  float Pvalue = 0;

  // Loop over the d_M and d_N tiles required to compute d_P element
  for (int ph = 0; ph < num_blocks; ++ph) {
    // Collaborative loading of d_M and d_N tiles into shared memory
    if ((Row < H1) && (ph * Tile_GEMM + tx) < W1)  {
      Mds[ty][tx] = M[Row * W1 + ph * Tile_GEMM + tx];
    }
    if ((ph * Tile_GEMM + ty) < H2 && Col < W2)  {
      Nds[ty][tx] = N[(ph * Tile_GEMM + ty) * W2 + Col];
    }

    __syncthreads();

    for (int k = 0; k < Tile_GEMM && (ph * Tile_GEMM) + k < W1; ++k)  {
      Pvalue += Mds[ty][k] * Nds[k][tx];
    }
    __syncthreads();
  }

  if ((Row < H1) && (Col < W2)) {
    P[Row * W3 + Col] = Pvalue;

    switch (IS_BIASED) {
      case BIASED:
        Pvalue = Pvalue + bias_mat.elements[Row];
        break;

      default:
        break;
    }

    switch (activation) {
      case SWISH_ACTIVATION:
        // Swish activation function
        P[Row * W3 + Col] = Pvalue / (1 + exp(-1 * Pvalue));
        break;
      case SIGMOID_ACTIVATION:
        // Sigmoid activation function
        P[Row * W3 + Col] = 1 / (1 + exp(-1 * Pvalue));
        break;

      default:
        break;
    }
  }
}
```

Matrix multiplication algorithm using WT description:

Algorithm 6 (Tiled Parallel Matrix multiplication, WT description)

Input: Matrix M_{exw}, Matrix N_{wxz}

Output: Matrix $P_{exz} = \prod (M, N)$

1. __shared__ float Mds [TILE] [TILE]
2. __shared__ float Nds [TILE] [TILE]
3. int Row := BlockIdx.y * Tile + ThreadIdx.y
4. int Col := BlockIdx.x * Tile + ThreadIdx.x
5. **forall** Row $\in 0 :$ $M\ rows$, Col $\in 0 :$ $N\ columns$ **do**
6. **for** phase $\in 0 :$ $Number$ of blocks **do**
7. **forall** Row < M #of rows **and** phase * Tile + ThreadIdx.x < M #of columns **do**:
8. Mds [ThreadIdx.y] [ThreadIdx.x] := M [Row * (M #of columns) + phase * Tile + ThreadIdx.x
9. **enddo**
10. **forall** phase * Tile + ThreadIdx.y < N #of columns **and** Col < N #of columns) **do**
11. Nds [ThreadIdx.y] [ThreadIdx.x] := N [(phase * Tile + ThreadIdx.y) * (N #of columns) + Col];
12. **enddo**
13. __syncthreads()
14. **for** k ∈ 0 : Tile **and** phase * Tile + k < M #of columns **do**
15. Pvalue ← Mds [ThreadIdx.y][k] x Nds [k][ThreadIdx.x]
16. **enddo**
17. __syncthreads();
18. **enddo**
19. P[Row x P columns + col] ← Pvalue
20. **enddo**

7.3 Model timing details

7.3.1 MBConv function in python

```
def forward(self, x):
    z = x

    t0= time.clock()

    if self.expansion_conv is not None:
        x = self.expansion_conv(x)

    x = self.depthwise_conv(x)

    x = self.squeeze_excitation(x)

    x = self.project_conv(x)

    # Add identity skip
    if x.shape == z.shape and self.with_skip:
      if self.training and self.drop_connect_rate is not None:
        self._drop_connect(x)
      x += z

    t1 = time.clock() - t0
```

Figure 7-28: Timing in MBConv

7.3.2 Stem and Head layers

```
def forward(self, x):
    t0= time.clock()

    f = self.stem(x)

    t1 = time.clock() - t0
    stem_timing = t1
    print("Time elapsed for stem: ", t1*1000, " msec")


    f = self.blocks(f)


    t0= time.clock()

    y = self.head(f)

    t1 = time.clock() - t0
```

Figure 7-29: Stem and Head layers

To use CPU or GPU functions: For GPU just put. Cuda attribute at the end of mode

```
model = EfficientNet(num_classes=1000,
                     width_coefficient=1.0, depth_coefficient=1.0,
                     dropout_rate=0.2)
model.cuda()
```

7.3.3 Cuda code details for timing

```
static void HandleError( cudaError_t err,
                         const char *file,
                         int line ) {
    if (err != cudaSuccess) {
        printf( "%s in %s at line %d\n", cudaGetErrorString( err ),
                file, line );
        exit( EXIT_FAILURE );
    }
}
#define HANDLE_ERROR( err ) (HandleError( err, __FILE__, __LINE__ ))


float time_defined = 0, tmp_time = 0, total_time_for_layer = 0;;
cudaEvent_t start_timing, stop_timing;
```

First we define some variables and events used later in the code functions.

- **Time_defined variable:** It is used to get the time when the execution tis done.
- **Tmp_time:** It's needed for accumulating results if the same kernel is called multiple times. This case happens when calling a reduction kernel to calculate summation of matrix elements in very large matrices.
- **Total_time_for_layer:** It's used to accumulate the timing from all kernels, i.e. in an MBConv layer function we need to call around 5 kernels to do the operations needed. All of these timings are accumulated in this variable.

```
// MBConv1_0 layer implementation
start();

DEFINE_FILTERS_FOR_MBCONV(&F1, NULL, 0, 0, 0 * FD1,
                          &F2, MBConv1_0_depthwise_conv_conv2d_weights, DWFilter_size, DWFilter_size, 32 * FD2,
                          &F3, MBConv1_0_squeeze_excitation1_conv2d_weights, 1, 1, 32 * FD3,
                          &F4, MBConv1_0_squeeze_excitation2_conv2d_weights, 1, 1, 8 * FD4,
                          &F5, MBConv1_0_project_conv_conv2d_weights, 1, 1, 32 * FD5);
Matrix ConvOut1_0;
MBConv_Layer(&ConvOutStem, &ConvOut1_0,
             &F1, &F2, &F3, &F4, &F5,
             FD1, FD2, FD3, FD4, FD5,
             ConvOutStem.depth, FD5, DWFilter_size,
             stride, padding, skip_connection,
             MBConv1_0_squeeze_excitation1_conv2d_bias, MBConv1_0_squeeze_excitation2_conv2d_bias,
             NULL, NULL,
             NULL, NULL,
             MBConv1_0_depthwise_conv_BN_mean,     MBConv1_0_depthwise_conv_BN_variance,
             MBConv1_0_depthwise_conv_BN_weights,  MBConv1_0_depthwise_conv_BN_bias,
             MBConv1_0_project_conv_BN_mean,       MBConv1_0_project_conv_BN_variance,
             MBConv1_0_project_conv_BN_weights,    MBConv1_0_project_conv_BN_bias);
stop("MBConv 1_0", 0);
```

Cuda events: They are used to start and to stop the cudaTimer

7.3.4 Functions used to measure time and their definitions

```
void start();
void stop(char *notification, int pause_time);
void after_pause(char *notification);
void reset_time();
```

```
void start()
{
  HANDLE_ERROR(cudaEventCreate(&start_timing));
  HANDLE_ERROR(cudaEventCreate(&stop_timing));
  HANDLE_ERROR(cudaEventRecord(start_timing, 0));
}

void stop(char *notification, int pause_time)
{
  HANDLE_ERROR(cudaEventRecord(stop_timing, 0));
  HANDLE_ERROR(cudaEventSynchronize(stop_timing));
  HANDLE_ERROR(cudaEventElapsedTime(&time_defined, start_timing, stop_timing));

  if(pause_time)
 {
    tmp_time += time_defined;
 }

  else
  {
    tmp_time = 0;
    printf("Time elapsed for %s:  %.8f ms\n", time_defined, notification);
    total_time_for_layer += time_defined;
  }
}
```

```
void after_pause(char *notification)
{
  printf("Time elapsed for %s: %.8f ms\n", notification, tmp_time);
  total_time_for_layer += tmp_time;

  tmp_time = 0;
}
```

In order to measure the executing time for any function or kernel execution, just surround it with start and stop functions. If there are multiple iterations, then we need the $afer\_pause$ function.

7.3.5 Cuda model performance compared to CPU and GPU model implementation

```
On average:
Time elapsed for padding kernel:  0.020188 msec

Time elapsed for Input unrolling kernel:  0.02089600

Time elapsed for Conv2d Matrix multiplication kernel: 0.1259040

Time elapsed for BatchNorm equation kernel: 0.098552163265

Time elapsed for ElementWise Multiplication kernel: 0.012348

Time elapsed for DepthWise Matrix multiplication: 0.03929

Time elapsed for Reduction mean for calculating sum: 0.0461044705

Time elapsed for Identity skip (summation) kernel: 0.00735288
```

Figure 7-30: Average time elapsed for each kernel

The architecture of EfficientNet-B0 is the following:

```
1 - Stem    - Conv3x3|BN|Swish

2 - Blocks  - MBConv1, k3x3
            - MBConv6, k3x3 repeated 2 times
            - MBConv6, k5x5 repeated 2 times
            - MBConv6, k3x3 repeated 3 times
            - MBConv6, k5x5 repeated 3 times
            - MBConv6, k5x5 repeated 4 times
            - MBConv6, k3x3
                          totally 16 blocks

3 - Head    - Conv1x1|BN|Swish
            - Pooling
            - Dropout
            - FC
```

Table 7-3: Comparing python model CPU and GPU with the implemented CUDA code

Comparing python model CPU and GPU with the implemented CUDA code

Layer Name	Cuda code (Timing for a function kernels' only)	Cuda code (GPU and CPU function execution)	Python Model CPU	Python Model GPU
Stem	0.19865599	0.55142403	4.148000000000707	1.2309999999988719
MBConv1_0	0.34947202	0.68025601	5.976999999999677	2.5250000000003325
MBConv6_1	0.65676796	0.81308800	15.402999999999167	1.4750000000010033
MBConv6_2	0.60608000	0.80227202	9.26899999999975	1.4420000000008315
MBConv6_3	0.40761596	0.53843200	6.833000000000311	1.9789999999986208
MBConv6_4	0.44540805	0.51401597	5.202999999999847	1.422999999999064
MBConv6_5	0.28009599	0.33795199	3.4039999999997406	1.40000000000029
MBConv6_6	0.38182402	0.42374399	3.068000000000737	1.4219999999998123
MBConv6_7	0.37651196	0.41212800	2.764000000000877	2.0039999999994507
MBConv6_8	0.42937601	0.46089599	3.1439999999989254	1.4609999999990464
MBConv6_9	0.61430395	0.63324797	3.894999999999982	2.0610000000012008
MBConv6_10	0.60118401	0.76784003	3.9789999999992887	2.029999999999532
MBConv6_11	0.50601602	0.57753599	2.997999999999834	1.4249999999993435
MBConv6_12	0.70182401	0.73945600	3.733999999999682	1.9740000000005864
MBConv6_13	0.69724804	0.73695999	2.6519999999994326	1.4219999999998123
MBConv6_14	0.70345598	0.73680001	2.741999999999578	2.0239999999986935
MBConv6_15	0.67593598	0.71692801	3.959000000000046	1.4189999999985048
Head	0.52758396	0.61539199	2.825999999998885	1.7940000000002954
New timing:	9.15935991 msec	10.38739166 msec	85.998 msec	30.511 msec

Conclusion

By the end of this project, we have learnt a lot of things. We learnt how to build the hardware and software of UAV from scratch. This gave us the opportunity to know more about Kalman filter, sensor fusion, and aerodynamics.

We got the knowledge of how to construct a ventilator using some sensors and pneumatic system and how to communicate between all elements of the system to get the targeted application of ventilation. To do this we learnt more about ventilation modes and a lot of medical concepts to be able to translate this into a code.

For the deep learning part. We know a lot of things about GPUs, CPUs, and parallelism. Also, we know how to implement a CNN detection network using CUDA C achieving higher performance than one done using python.

During the year, we learnt how to work as a group, manage our time, and how to organize ourselves to achieve the tasks within a limited time.

References

[15] Tan, Mingxing, and Quoc V. Le. “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks.” ArXiv:1905.11946 [Cs, Stat], Sept. 2020. arXiv.org,

[16] Sandler, Mark, et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks.” ArXiv:1801.04381 [Cs], Mar. 2019. arXiv.org,

[17] Hu, Jie, et al. “Squeeze-and-Excitation Networks.” ArXiv:1709.01507 [Cs], May 2019. arXiv.org,

[18] CUDA C++ Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[19] Cook, S., 2012. CUDA programming: a developer's guide to parallel computing with GPUs. Newnes.

[20] Udacity introduction to parallel programming https://youtube.com/playlist?list=PLAwxTw4SYaPnFKojVQrmyOGFCqHTxfdv2

[21] Anderson, A., Vasudevan, A., Keane, C. and Gregg, D., 2017. Low-memory gemm-based convolution algorithms for deep neural networks. arXiv preprint arXiv:1709.03395.

[22] Chellapilla, K., Puri, S., & Simard, P. (2006). High performance convolutional neural networks for document processing. < https://hal.archives-ouvertes.fr/inria-00112631/document >.

[23] Warden, Pete. "Why GEMM Is at the Heart of Deep Learning". Pete Warden's Blog, April 20, 2015, https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/.

[24] Kirk, D.B. and Wen-Mei, W.H., 2016. Programming massively parallel processors: a hands-on approach. Morgan kaufmann.