PARALLEL PROGRAMMING USING CUDA

BE FINAL YEAR PROJECT

BY
ANURAG KUMAR JAISWAL ROLL NO: 001311001048
SATISH KUMAR ROLL NO: 001311001002

SUPERVISOR:
MR. UTPAL KR. RAY
ASSISTANT PROFESSOR

INTRODUCTION

- **Graphic Processor Units :**A graphic processor unit is simply a processor attached to the graphics card used in video games, PlayStations and computers. The way they are different from the CPUs, the central processing units, are that they are massively threaded and parallel in their operations. This is because of the nature of their work – they are used for fast rendering; same operation is carried out for each of the pixels in the image. Thus, they have more transistors devoted to data processing rather than flow control and data caching.

PARALLEL PROGRAMMIING

- In computing, a parallel programming model is an abstraction of parallel computer architecture, with which it is convenient to express algorithms and their composition in programs. A parallel program is composed of simultaneously executing processes.
- There are two different approaches of achieving parallelism while solving a problem.
- They are CONTROL PARALLEL approach and DATA PARALLEL approach.

WHAT IS CUDA?

- CUDA is a parallel computing platform and application programming interface (API) model created by NVIDIA. It allows software developers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing – an approach known as GPGPU. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.

HOW CUDA WORKS?

- The GPU programming model is very different from a CPU programming model as it inherently supports parallel programming. A usual template followed to program a GPU is as below:
- Transfer the data to the GPU memory.
- Launch the GPU kernel from the host program.
- Wait till GPU finishes off the calculations.
- Transfer the results back to CPU memory.
- Thus, a CUDA program consists of two parts in its code:

  a) **host program** which runs on CPU

  b) **kernel** which does the calculations on GPU

CONTINUE…

- A brief description of the various programming features is as below:
- The execution of the kernels is done in threads and blocks. The structure of the grid of threads and blocks is passed in the host code while calling the kernel using a unique <<<…>>> syntax. A typical kernel call looks like:
- **funcAdd<<<Grid,Blocks,1>>>(A, B, C)**

- These threads and blocks are executed on the various multiprocessors of the GPU. The threads of each block are executed concurrently in one multi-processor; each of them may process some maximum number of blocks depending on the resources used by each thread. As the blocks get executed, new blocks are launched on the vacated processors.

CONTINUE…

- The threads are processed in groups of 32 called warps. The splitting into warps is done according to increasing thread IDs.
-
- Each thread and block has a unique ID, which is defined by a 3 component vector namely **threadIdx** and **blockIdx** respectively. The thread IDs are with respect to their position in the block. A typical grid of threads and blocks is shown in figure1.
-
- Threads within a block can synchronize using a barrier synchronization function __**syncthreads()**. This function stops the execution of threads until all the threads have reached that point.

CONTINUE…

- The threads are executed in parallel unless their paths diverge or they read/write from the same memory location. In the former case, the processor executes each path serially, disabling threads on other paths until the paths converge again. In the latter case too, the access is serialized, but in the case of non-atomic write, the outcome of the write is not guaranteed.

Continue…

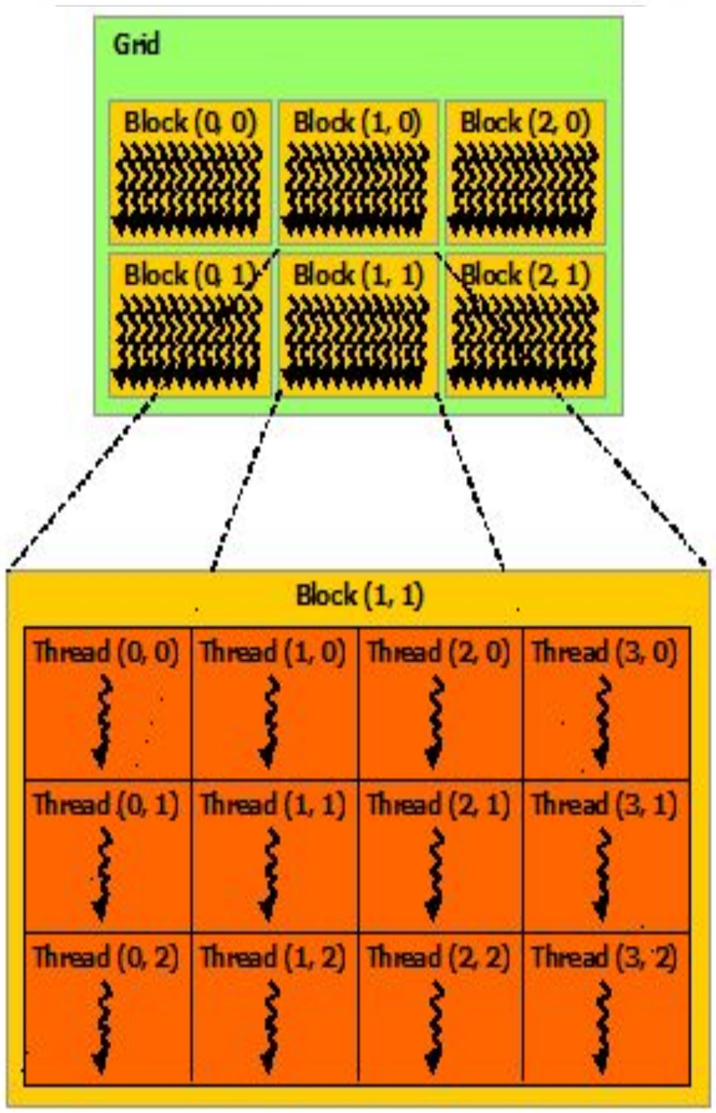


Figure 1: Grid and Thread Blocks

THREAD LOCATION: DIMENSION AND INDEX

- Every running thread has access to several structures that help it to discover its location within the grid and blocks
- **GridDim**: The dimensions of the grid. Use gridDim.x, gridDim.y and gridDim.z to access the dimensions. In the example of Figure 2.1, gridDim.x = 7, gridDim.y = 5 and gridDim.z = 0.
- **blockIdx :** The location of a block within the grid. All 512 threads in the middle block of the grid would find blockIdx.x = 3, blockIdx.y = 2, blockIdx.z = 0, if they queried this structure

blockDim : The dimensions of the blocks. All threads in the example would find blockDim.x = 32, blockDim.y = 16 and blockDim.z = 0.

threadIdx : This structure gives the location of a thread within its own block. Thus the top-left thread of each block would find threadIdx.x = threadIdx.y = threadIdx.z = 0, while the top-right threads would find threadIdx.x = 31, threadIdx.y = 0 and threadIdx.z = 0.

KERNELS AND MULTIPROCESSORS

- A kernel is the unit of work that the main program running on the host computer offloads to the GPU for computation on that device.
- In CUDA, launching a kernel requires specifying three things:
  - The dimensions of the grid
  - The dimensions of the blocks
  - The kernel function to run on the device.
- Kernel functions are specified by declaring them __global__ in the code.

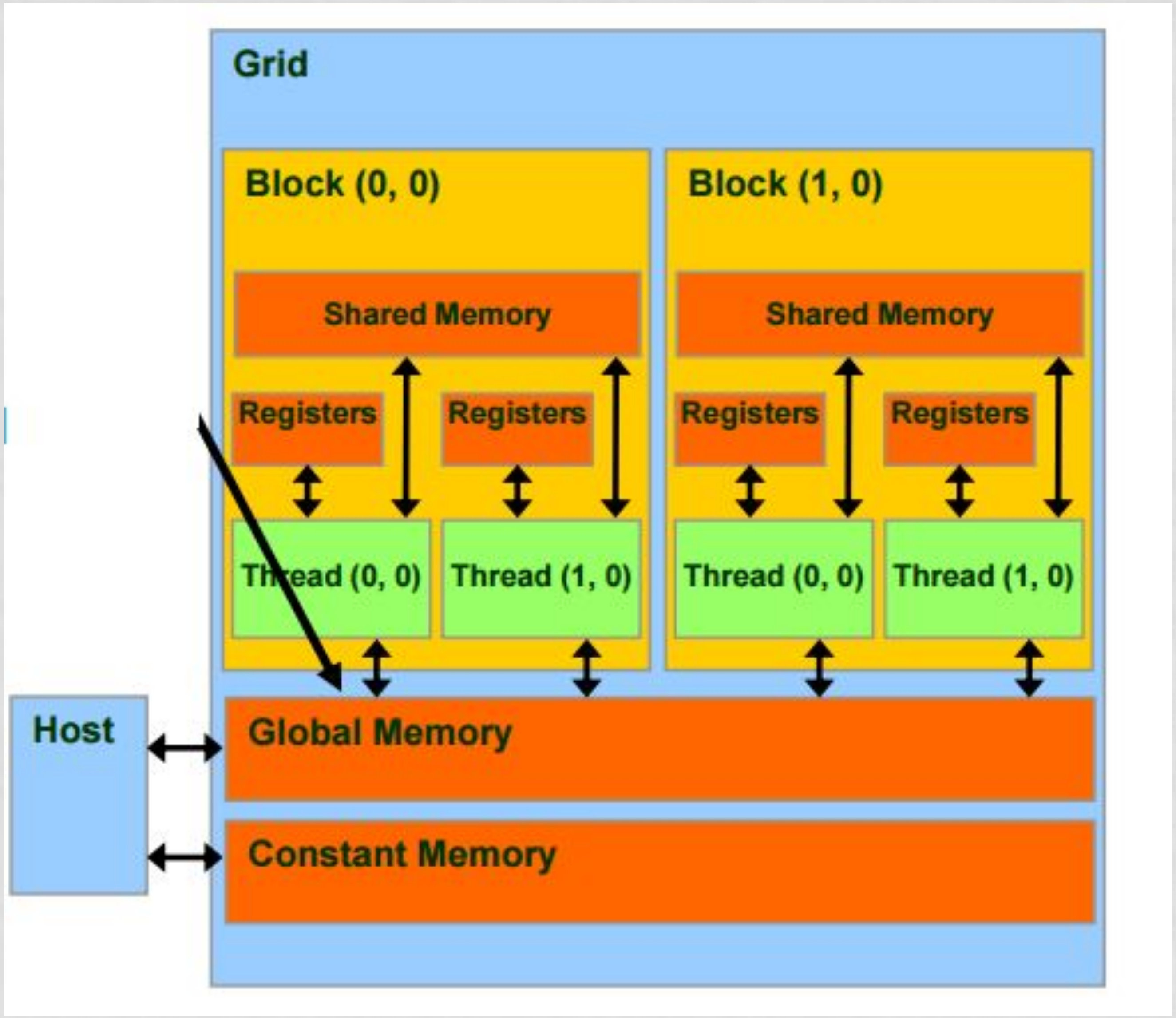
GPU MEMORY HIERARCHY

- GPU has different levels of memory - global memory, shared memory, constant memory, texture memory and local memory and registers

- **Registers and local memory**: Each multiprocessor has certain number of 32-bit registers. They are allocated per thread and only that thread can access the contents of the register. These are the fastest accessible memories, and are allocated at the compile time depending on the requirement of a thread. If the threads in a block need more registers than provided in a multiprocessor, the kernel fails to launch. Local Memory is also available per thread and the compiler automatically allocates certain variables on the local memory, like a big array. Local memory is not cached and hence its access is as costly as global memory.

MEMORY HIERARCHY

- **Global Memory:**It is an off-chip memory and is accessible to all threads. The data transferred from CPU is stored in the global memory initially. The memory is connected to all the processors through a wide bus, which can be up to 512 bits wide
- **Constant Memory**:Used for storing constant values that does not change over time. Constant memory resides in the global memory
- **Texture Memory:** CUDA also supports a part of the texturing hardware which is used for graphics processing. This allows the texture memory to be read using device functions called texture fetches. As the texture memory is cached, it is much faster than global memory access



GPU Memory Hierarchy

ADD TWO VECTORS USING CUDA

- The aim of this task is to add two vector and compare the time required by CPU based sequential program with GPU based parallel program.
- Approach:

```
__global__ void VecAdd(const int* A,const int* B,int* C,int N)
{
   int i = blockDim.x * blockIdx.x + threadIdx.x;
   if(i<N)
    C[i] = A[i] + B[i];
}
```

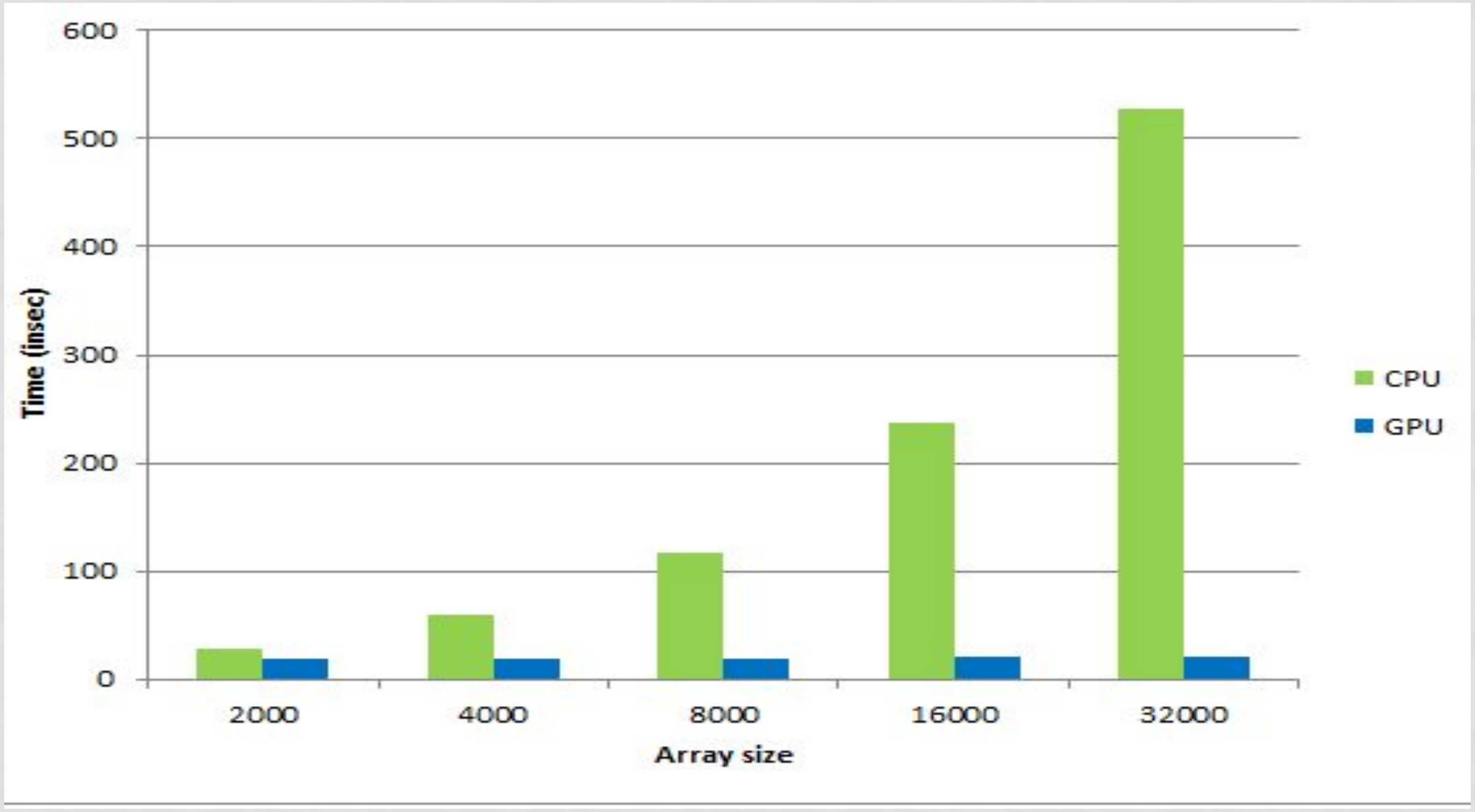
Results

The execution time was recorded for different sizes of the array and the following table was obtained.

Array Size	Memory Required	Time(in µsec) for CPU	Time(in µsec) for GPU
2000	0.034	28	20
4000	0.069	59	19
8000	0.137	117	20
16000	0.275	237	21
32000	0.549	527	21

Here , we observed that the execution time increase with Increase of array size in both cpu and gpu but the execution Time in gpu very less than the cpu.

GRAPH..



600
500
400
300
200
100
0
Time (insec)
2000
4000
8000
16000
32000
Array size
CPU
GPU

GPU BASED MATRIX MULTIPLICATION USING CUDA

- **The Problem Statement** :The aim of this problem is to multiply two vectors of equal dimension using the CUDA framework by using the computation power of the highly parallel structure of the GPU. The aim of this task is to compare the time required by CPU based sequential program with GPU based parallel program with and without shared memory.

A COMMON PROGRAMMING STRATEGY

- Global memory is much slower than shared memory
- So, a profitable way of performing computation on the device is to tile data to take advantage of fast shared memory:
  - Partition data into subsets that fit into shared memory
  - Handle each data subset with one thread block by:
    - Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the subset from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory to global memory

USING GLOBAL MEMORY

- There are three functions in this.

1. main() :
2. MatMul(Matrix A, Matrix B, Matrix C)
3. MatMulKernel(Matrix A, Matrix B, Matrix C)

- **main** function generates two random matrices with dimensions that are read from the commandline. Matrix A is filled randomly with numbers ,Then main() calls MatMul() which multiplies these and places the product in Matrix C

Matmul function takes two matrices A and B as input, and fills the entries of matrix C with the product. It first allocates memory on the device for matrix A, and copies A onto the device's global memory. Then it does the same for matrix B. It allocates space on the device for C, the matrix product, computes the number of blocks needed to cover the product matrix, and then launches a kernel with that many blocks. When the kernel is done, it reads matrix C off of the device, and frees the global memory.

MatMulKernel(Matrix A, Matrix B, Matrix C)
this runs on the device and computes the product matrix. It assumes that A and B are already in the device's global memory, and places the product in the device's global memory, so that the host can read it from there.

KERNEL FUNCTION

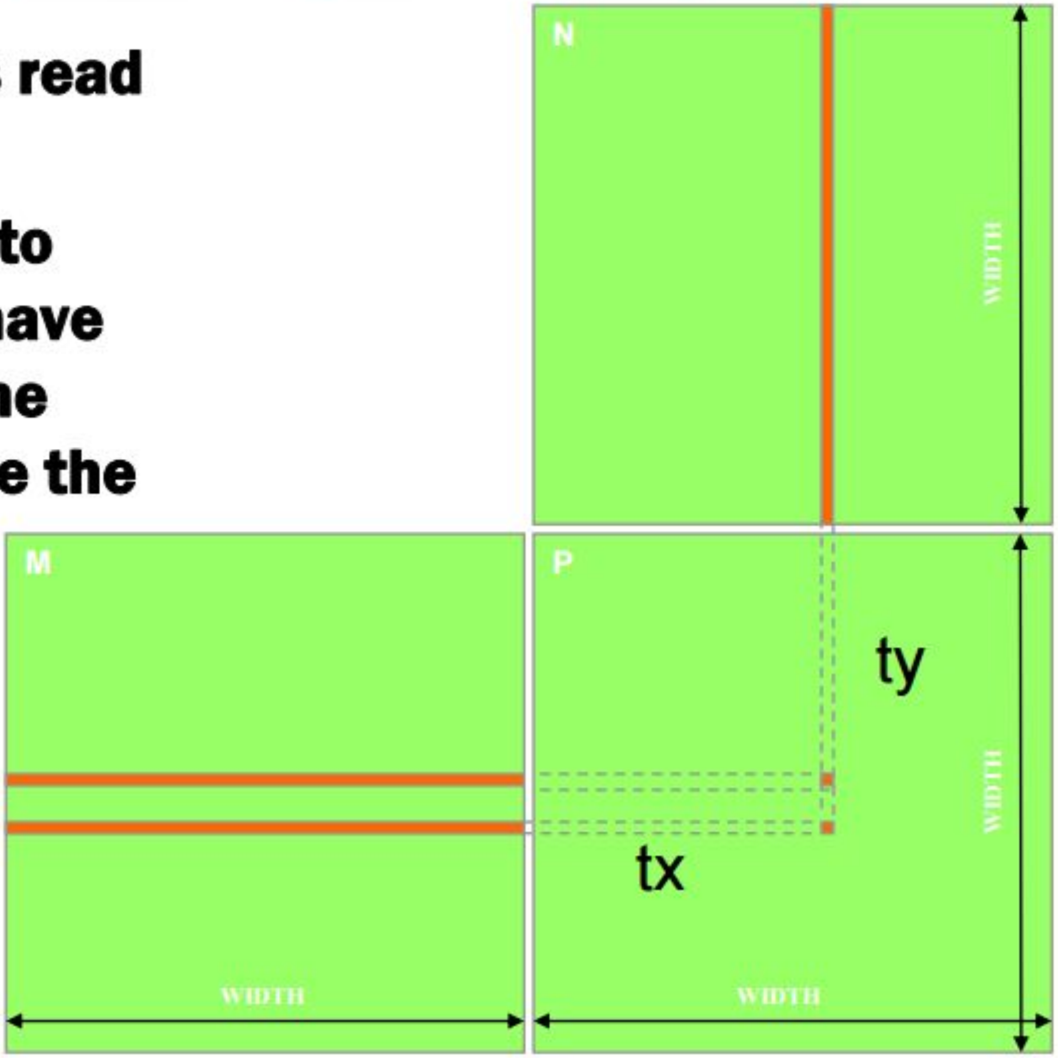
```
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
  // Each thread computes one element of C
  // by accumulating results into Cvalue
  float Cvalue = 0;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int col = blockIdx.x * blockDim.x + threadIdx.x;
  if(row > A.height || col > Width) return;
  for (int e = 0; e < A.width; ++e)
  Cvalue += A.elements[row * A.width + e] *
  B.elements[e * Width + col];
  C.elements[row * C.width + col] = Cvalue;
}
```

MATRIX MULTIPLICATION WITH SHARED MEMORY

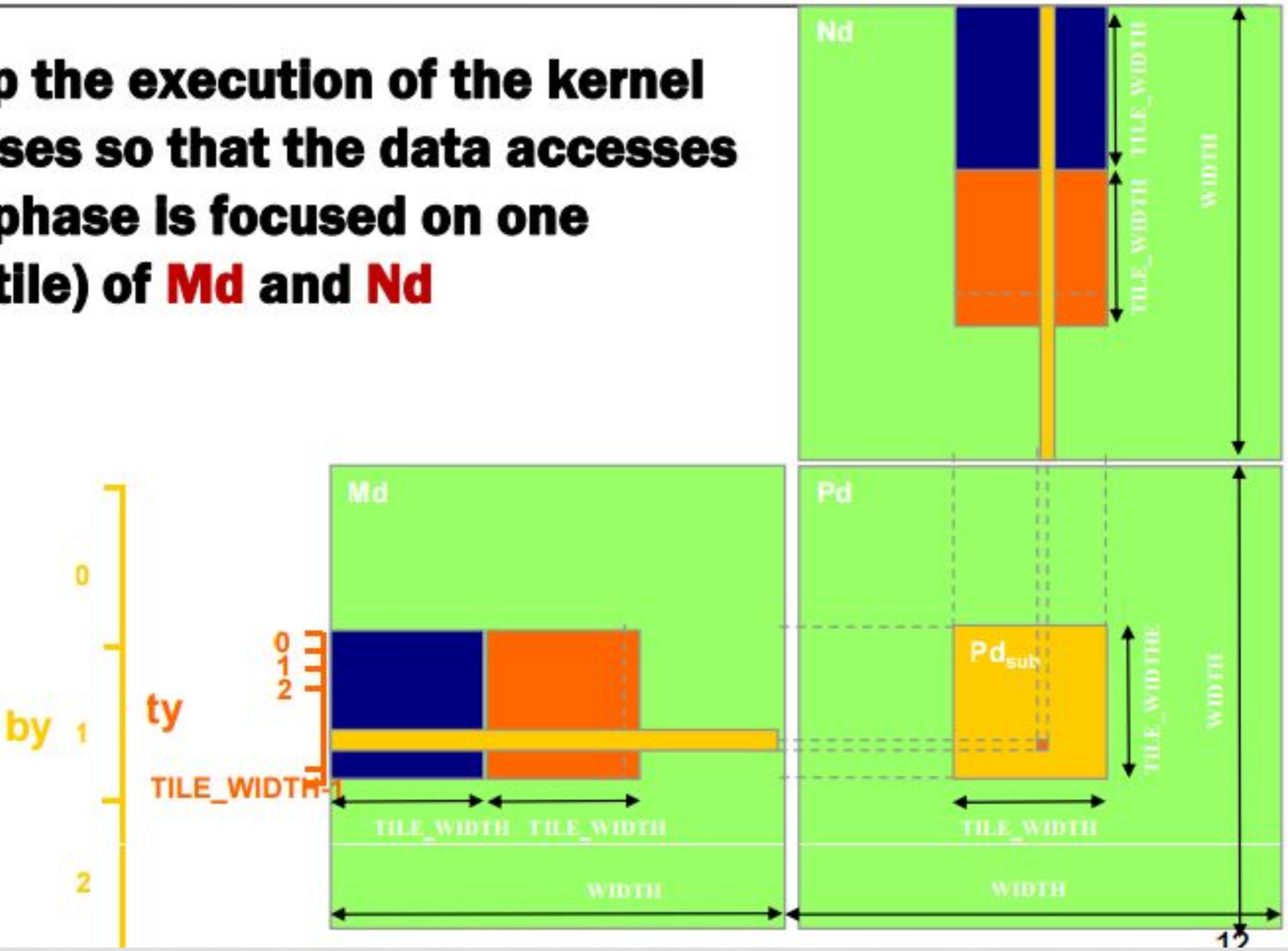
- We notice that each thread loads (2 × MD.width) elements from global memory.two for each iteration through the loop, one from matrix MD and one from matrix ND. Since accesses to global memory are relatively slow, this can bog down the kernel, leaving the threads idle for hundreds of clock cycles, for each access. One way to reduce the number of accesses to global memory is to have the threads load portions of matrices MD and ND into shared memory, where we can access them much more quickly. Ideally, we would load both matrices entirely into shared memory, but unfortunately, shared memory is a rather scarce resource, and won't hold two large matrices. Devices of compute capability 1.x have 16kB of shared memory per multiprocessor, and for 2.x have 48kB. So we will content ourselves with loading portions of MD and ND into shared memory as needed, and making as much use of them as possible while they are there.

Idea: Use Shared Memory to reuse global memory data

- Each input element is read by *WIDTH* threads.
- Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth
  - Tiled algorithms

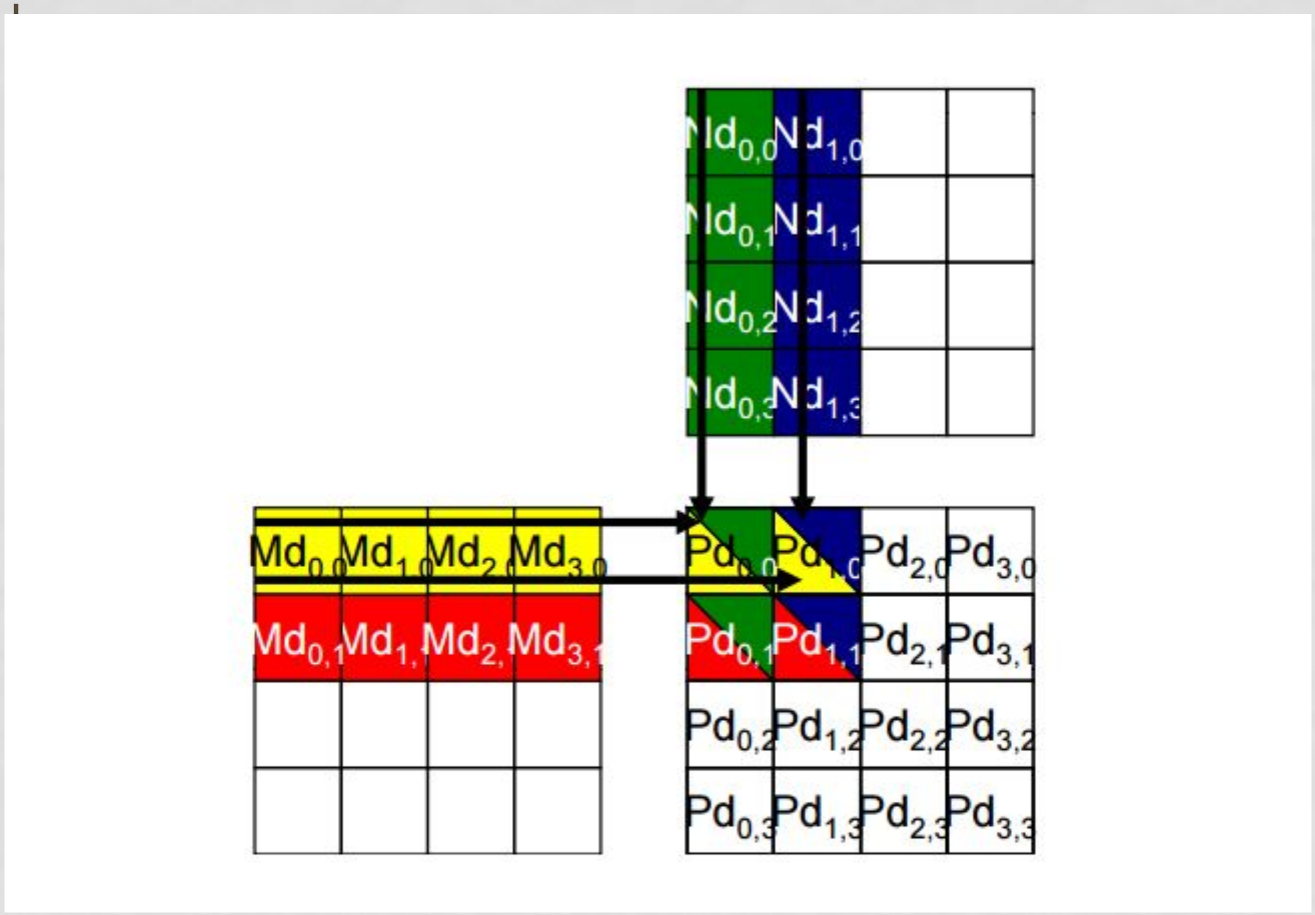


- Break up the execution of the kernel into phases so that the data accesses in each phase is focused on one subset (tile) of Md and Nd



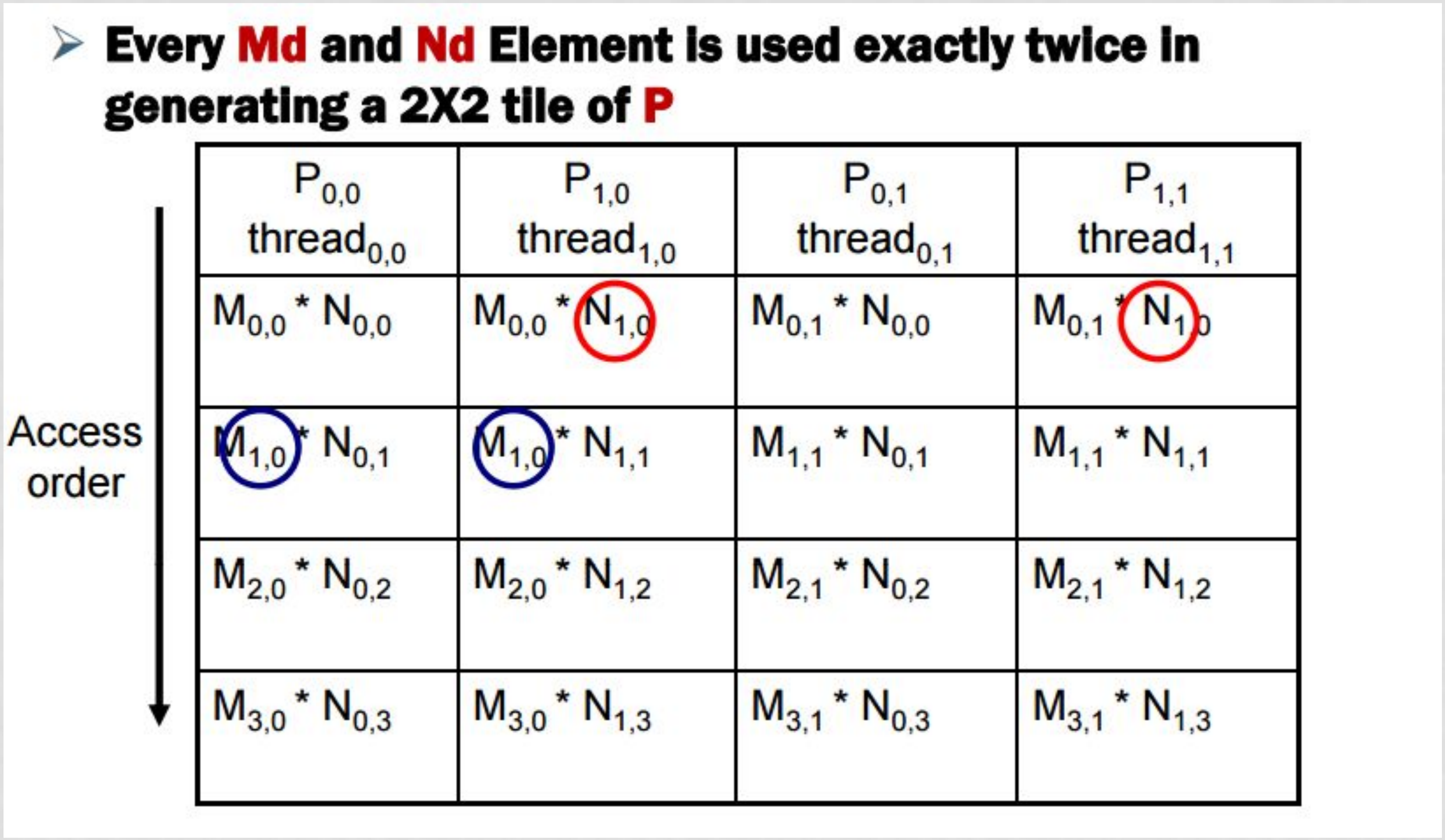
12

EXAMPLE



Nd0,0 Nd1,0
Nd0,1 Nd1,1
Nd0,2 Nd1,2
Nd0,3 Nd1,3
Md0,0 Md1,0 Md2,0 Md3,0
Md0,1 Md1,1 Md2,1 Md3,1
Pd0,0 Pd1,0 Pd2,0 Pd3,0
Pd0,1 Pd1,1 Pd2,1 Pd3,1
Pd0,2 Pd1,2 Pd2,2 Pd3,2
Pd0,3 Pd1,3 Pd2,3 Pd3,3

EXAMPLE CONTINUE

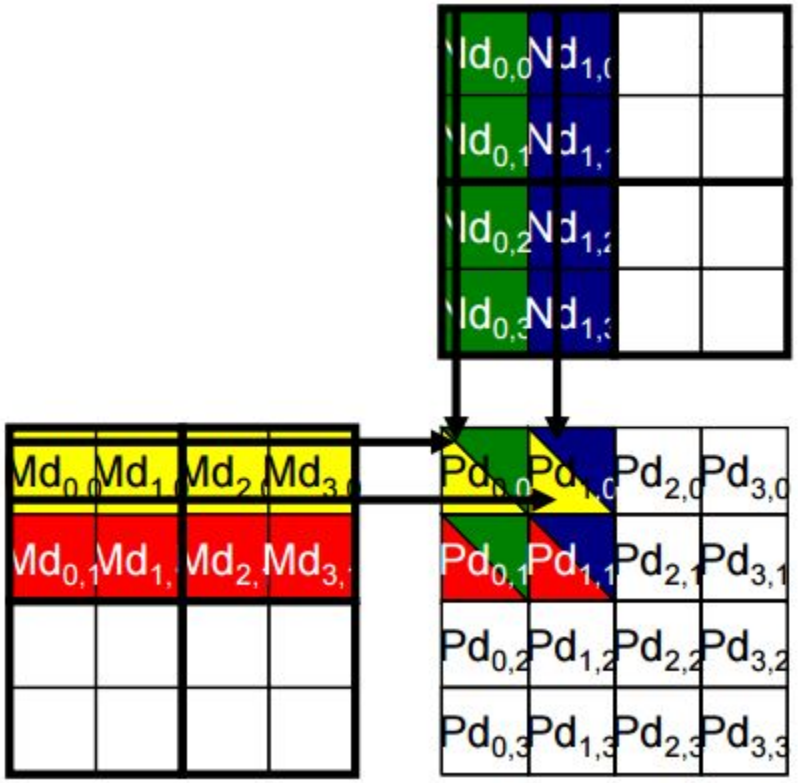


➢ **Every Md and Nd Element is used exactly twice in generating a 2X2 tile of P**

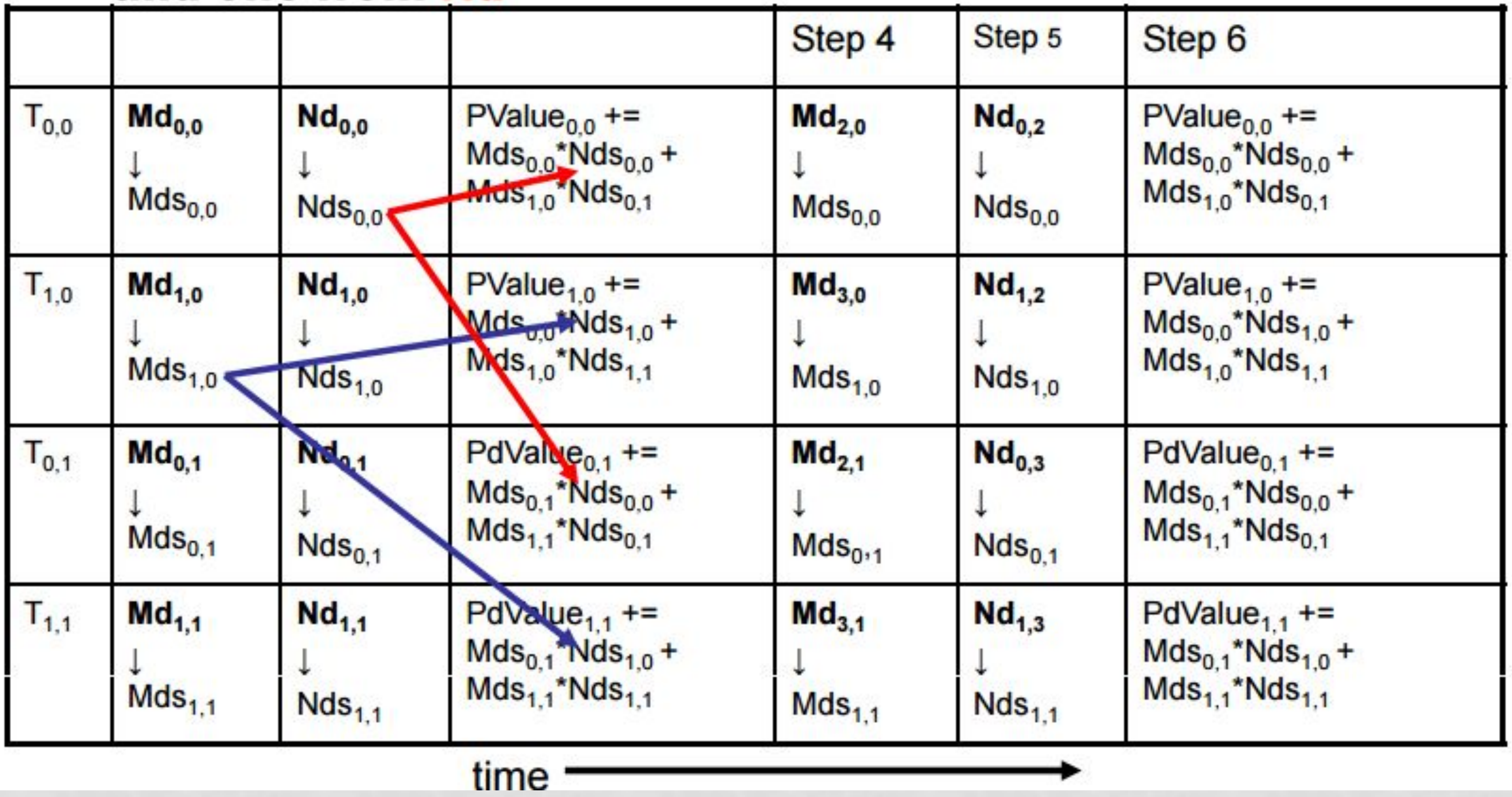
Access order

$P_{0,0}$ thread$_{0,0}$	$P_{1,0}$ thread$_{1,0}$	$P_{0,1}$ thread$_{0,1}$	$P_{1,1}$ thread$_{1,1}$
$M_{0,0} * N_{0,0}$	$M_{0,0} * N_{1,0}$	$M_{0,1} * N_{0,0}$	$M_{0,1} * N_{1,0}$
$M_{1,0} * N_{0,1}$	$M_{1,0} * N_{1,1}$	$M_{1,1} * N_{0,1}$	$M_{1,1} * N_{1,1}$
$M_{2,0} * N_{0,2}$	$M_{2,0} * N_{1,2}$	$M_{2,1} * N_{0,2}$	$M_{2,1} * N_{1,2}$
$M_{3,0} * N_{0,3}$	$M_{3,0} * N_{1,3}$	$M_{3,1} * N_{0,3}$	$M_{3,1} * N_{1,3}$

BREAK MD AND ND INTO BLOCK



➢ **Each phase of a Thread Block uses one tile from Md and one from Nd**

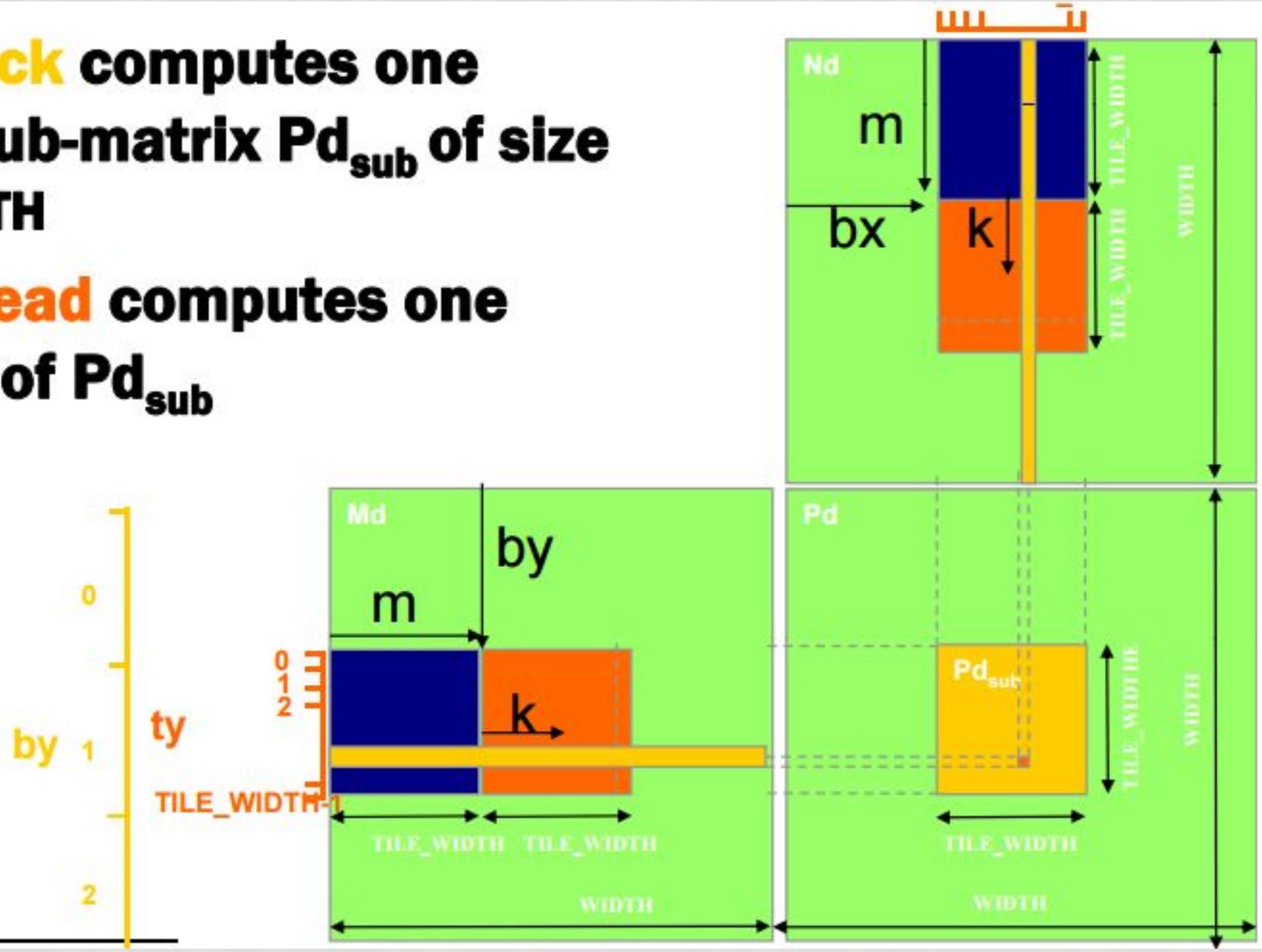


				Step 4	Step 5	Step 6
$T_{0,0}$	$\mathbf{Md_{0,0}}$ $\downarrow$ $Mds_{0,0}$	$\mathbf{Nd_{0,0}}$ $\downarrow$ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$	$\mathbf{Md_{2,0}}$ $\downarrow$ $Mds_{0,0}$	$\mathbf{Nd_{0,2}}$ $\downarrow$ $Nds_{0,0}$	$PValue_{0,0}$ += $Mds_{0,0}$*$Nds_{0,0}$ + $Mds_{1,0}$*$Nds_{0,1}$
$T_{1,0}$	$\mathbf{Md_{1,0}}$ $\downarrow$ $Mds_{1,0}$	$\mathbf{Nd_{1,0}}$ $\downarrow$ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$	$\mathbf{Md_{3,0}}$ $\downarrow$ $Mds_{1,0}$	$\mathbf{Nd_{1,2}}$ $\downarrow$ $Nds_{1,0}$	$PValue_{1,0}$ += $Mds_{0,0}$*$Nds_{1,0}$ + $Mds_{1,0}$*$Nds_{1,1}$
$T_{0,1}$	$\mathbf{Md_{0,1}}$ $\downarrow$ $Mds_{0,1}$	$\mathbf{Nd_{0,1}}$ $\downarrow$ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$	$\mathbf{Md_{2,1}}$ $\downarrow$ $Mds_{0,1}$	$\mathbf{Nd_{0,3}}$ $\downarrow$ $Nds_{0,1}$	$PdValue_{0,1}$ += $Mds_{0,1}$*$Nds_{0,0}$ + $Mds_{1,1}$*$Nds_{0,1}$
$T_{1,1}$	$\mathbf{Md_{1,1}}$ $\downarrow$ $Mds_{1,1}$	$\mathbf{Nd_{1,1}}$ $\downarrow$ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$	$\mathbf{Md_{3,1}}$ $\downarrow$ $Mds_{1,1}$	$\mathbf{Nd_{1,3}}$ $\downarrow$ $Nds_{1,1}$	$PdValue_{1,1}$ += $Mds_{0,1}$*$Nds_{1,0}$ + $Mds_{1,1}$*$Nds_{1,1}$

time $\longrightarrow$

BLOCK/TILE MULTIPLY

- **Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH**
- **Each thread computes one element of Pd_{sub}**



EXPLANATION

- We may decompose matrices Md and Nd into non-overlapping submatrices of size BLOCK_SIZE× BLOCK_SIZE. If we look at our red row and red column, they will pass through the same number of these submatrices, since they are of equal length. If we load the left-most of those submatrices of matrix Md into shared memory, and the top-most of those submatrices of matrix Nd into shared memory, then we can compute the first BLOCK_SIZE products and add them together just by reading the shared memory. But here is the benefit: as long as we have those submatrices in shared memory, every thread in our thread block (computing the BLOCK_SIZE×BLOCK_SIZE submatrix of Pd) can compute that portion of their sum as well from the same data in shared memory.
- When each thread has computed this sum, we can load the next BLOCK_SIZE× BLOCK_SIZE submatrices from Md and Nd, and continue adding the term-by-term products to our value in Pd. And after all of the submatrices have been processed, we will have computed our entries in Pd. The kernel code for this portion of our program is shown below

CALLING THE KERNEL

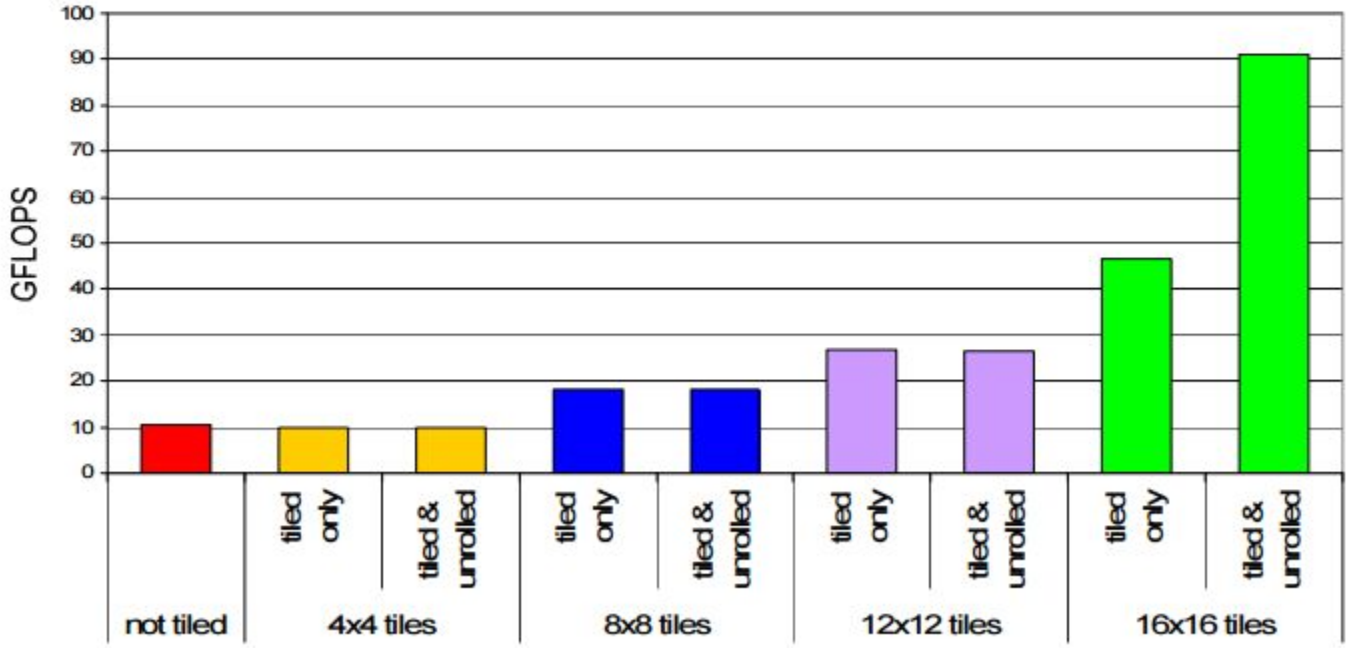
- The kernel in the MatMulKernel() kernel makes use of the __syncthreads() call. Whenever a thread calls __syncthreads(), all threads in that thread's block must compute up to this point before any thread is allowed to pass that point. With the first call to __syncthreads() we thus insure that every entry of the submatrices of A and B have been loaded into shared memory before any thread begins its computations based on those values. The second call to __syncthreads() ensures that every element of the submatrix of C has been processed before we begin loading the next submatrix of A or B into shared memory. Note that while the __syncthreads() primitive enables this sort of inter-thread synchronization, its use does minimize parallelism and may degrade performance if not used wisely and sparingly

Let’s take a look at the host code related to invocation of the kernel:

```
void MatMul(const Matrix A, const Matrix B, Matrix C)
{     ...
      // Invoke kernel
      dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
      dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
MatMulKernel<<>>(d_A, d_B, d_C);
      err = cudaThreadSynchronize();
      printf("Run kernel: %s\n", cudaGetErrorString(err)); ... \}
```

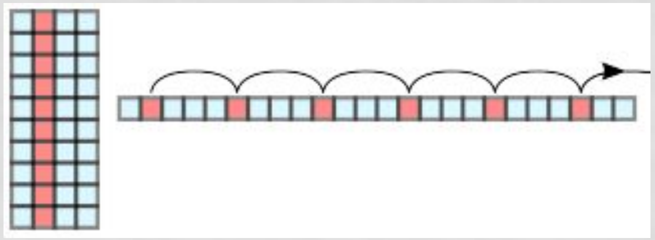
The dim3 structure defined in the CUDA libraries is very useful. It defines a triple of unsigned ints used to hold the dimensions of a grid or block

BLOCK SIZE EFFECTS



AN ASIDE ON STRIDE

- It may be worthwhile to say something about the stride variable used in this code. As mentioned above, matrices are stored in row major order, meaning that if we look in global memory for our matrix we will find it stored in a one-dimensional array consisting of the first row, followed by the second row, followed by the third row, etc... The stride refers to the width of the matrix, because that is how big one's steps must be if one wishes to "walk" down a column of the matrix. For example, if we have a 10 × 4 matrix in memory, then the elements of the first column of the matrix lie 4 apart in memory, as shown in the figure below.



- We give matrices a "stride" in this code because for submatrices, the size of a step one must take in memory in order to walk down a column is not the width of that submatrix, but rather the width of the parent matrix, which is its stride.

RESULTS

Matrix Size	Memory Required (in MB)	CPU Time (in sec)	Multiplier	GPU Time (in sec)	Multiplier	GPU Time using shared memory	Multiplier
500*500	1.43	0.473709	-----	0.122799	-----	0.112445	-----
1000*1000	5.722	3.912552	8.26	0.249378	2.03	0.159023	1.41
2000*2000	22.888	34.003586	8.69	1.180480	4.73	0.478651	3.01
4000*4000	91.5527	286.24081	8.42	8.540837	7.235	2.768426	5.78
8000*8000	366.211	2658.7105	9.29	67.29404	7.88	21.516653	7.77

Here it is observed that there is significant improvement in time if gpu shared memory is used.

CONCLUSION..

- In vector addition we found that there is significant improvement in time when addition of two vector of equal size is done on gpu using cuda platform than the addition n cpu.

- In matrix multiplication we apply two approach.
- 1. using global memory

the code just given works and is fast, but could be so much faster if we take advantage of shared memory.

2. using shared memory

this is faster and significant improvement in time than the global memory approach

SUMMARY

- **CUDA** is a parallel computing platform and API model created by Nvidia. It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels.
- Using cuda we implement two thing.

1. Vector addition
2. Matrix multiplication using global and shared memory

We measure the time required for the execution of respective code on gpu and cpu as well and compare the result.

SUMMARY-TYPICAL STRUCTURE OF A CUDA PROGRAM

- Global variables declaration
  - __ host __
  - __device__... __global__, __constant__, __texture__
- Function prototypes
  - __global__ void kernelOne()
- Main ()
  - allocate memory space on the device – cudaMalloc(&d_GlblVarPtr, bytes)
  - transfer data from host to device – cudaMemCpy(d_GlblVarPtr, h_Gl…)
  - execution config p uration setup
  - kernel call – kernelOne<<>>(args…);
  - transfer results from device to host – cudaMemCpy(h_GlblVarPtr,…)
  - optional: compare against golden (host computed) solution
- Kernel – void kernelOne(type args,…) repeat as needed (yp g ,)
  - variables declaration - __local__, __shared__
  - automatic variables transparently assigned to registers or local memory
  - syncthreads()…
- Other functions

REFERENCES

- https://en.wikipedia.org/wiki/Graphics_processing_unit
- https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units
- https://www.nvidia.com/object/what-is-gpu-computing.html
- http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-microsoft-windows/#axzz3cRo5p3v6
- https://developer.nvidia.com/maxwell-compute-architecture
- http://www.notebookcheck.net/NVIDIA-GeForce-840M.105681.0.html
- https://en.wikipedia.org/wiki/Maxwell_(microarchitecture)
- https://en.wikipedia.org/wiki/Gaussian_elimination