CUDA Programming

Lecture 2

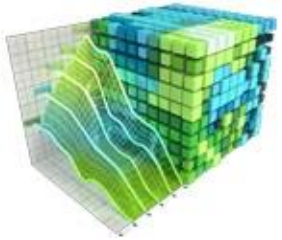
S.-Kazem Shekofteh

kazem.shekofteh@ziti.uni-heidelberg.de

*Post-Doc Research Fellow, ZITI, Institute of computer Engineering, **Heidelberg University, Germany***

*Faculty Member, Department of Computer Engineering, **Shandiz Institute of Higher Education, Iran***

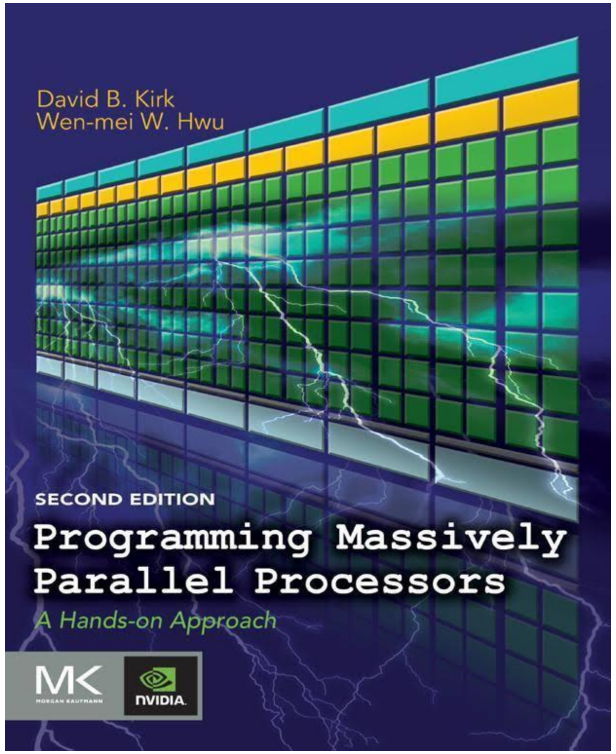
*PhD in Computer Engineering, **Ferdowsi University of Mashhad, Iran***



NVIDIA Visual Profiler

Reference

- David B. Kirk and Wen-mei W. Hwu, "Programming Massively Parallel Processors," 2^{nd} Ed. 2012



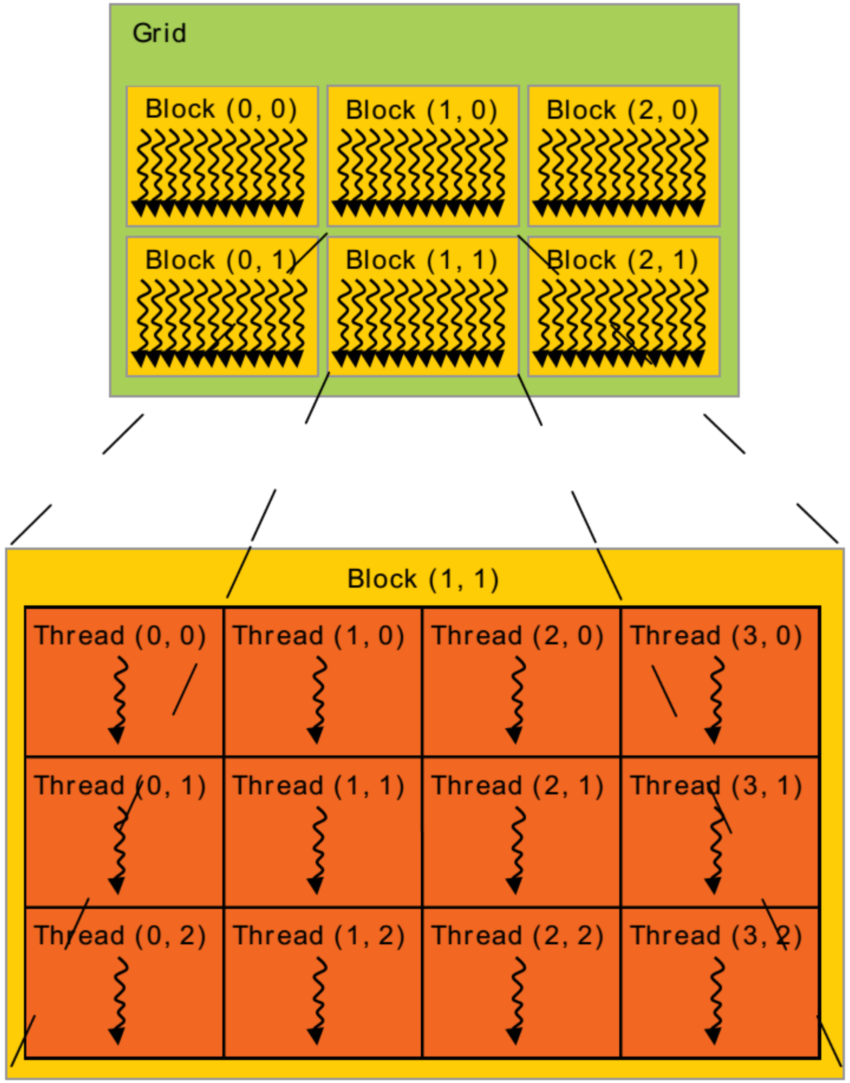
Outline

- Going deeper into CUDA thread organization
- Mapping threads to multidimensional data
- Matrix-Matrix multiplication sample
- Synchronization
- Introducing some advanced CUDA functions

Thread hierarchy

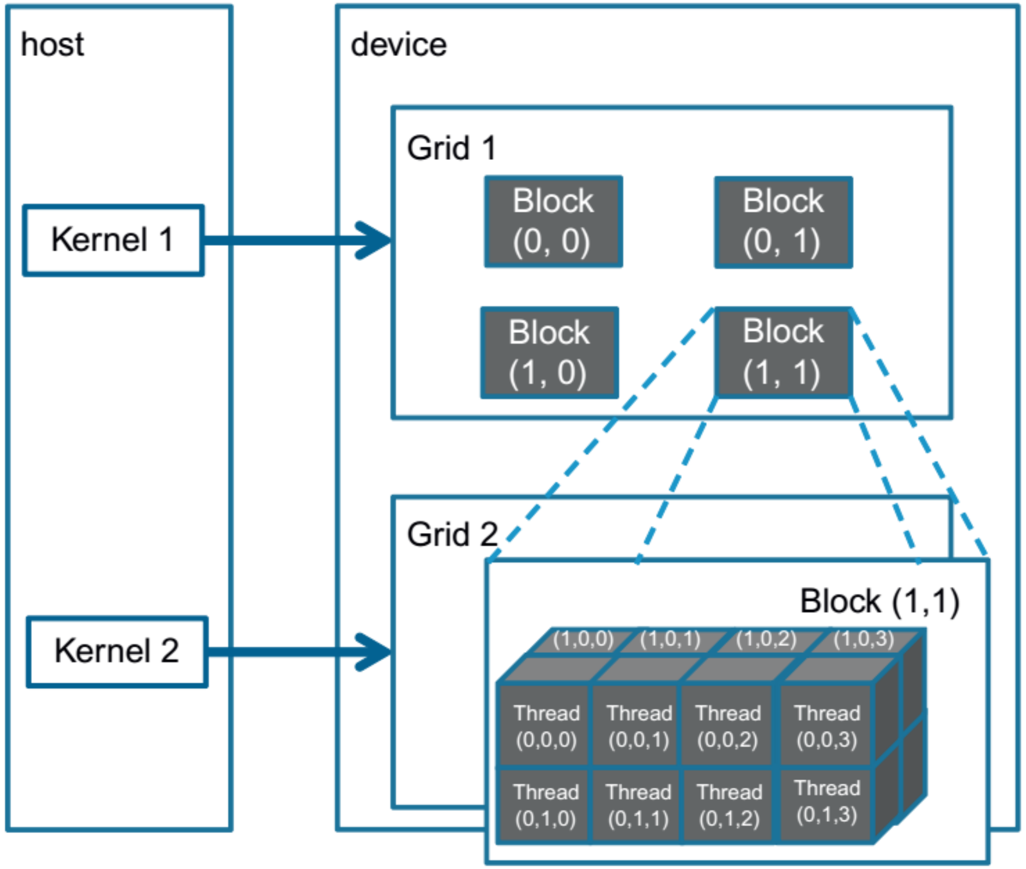
- The definition of
  - Grid
  - Block
  - Thread
- **`Dim3`** struct
  - x
  - y
  - z

```
dim3 dimBlock(128, 1, 1);
dim3 dimGrid(32, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(...);
```



Thread hierarchy

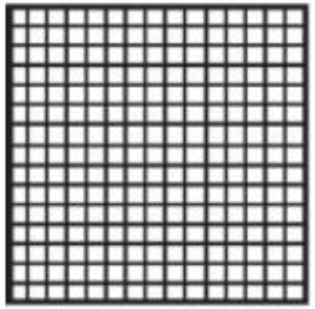
- Define grid and block for Kernel 1?
- Calculate total threads?



Mapping Threads to Multidimensional data

- The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data
  - Pictures are a 2D array of pixels

Example: A 72×62 picture while using 16×16 blocks



16×16 blocks

Mapping Threads to Multidimensional data

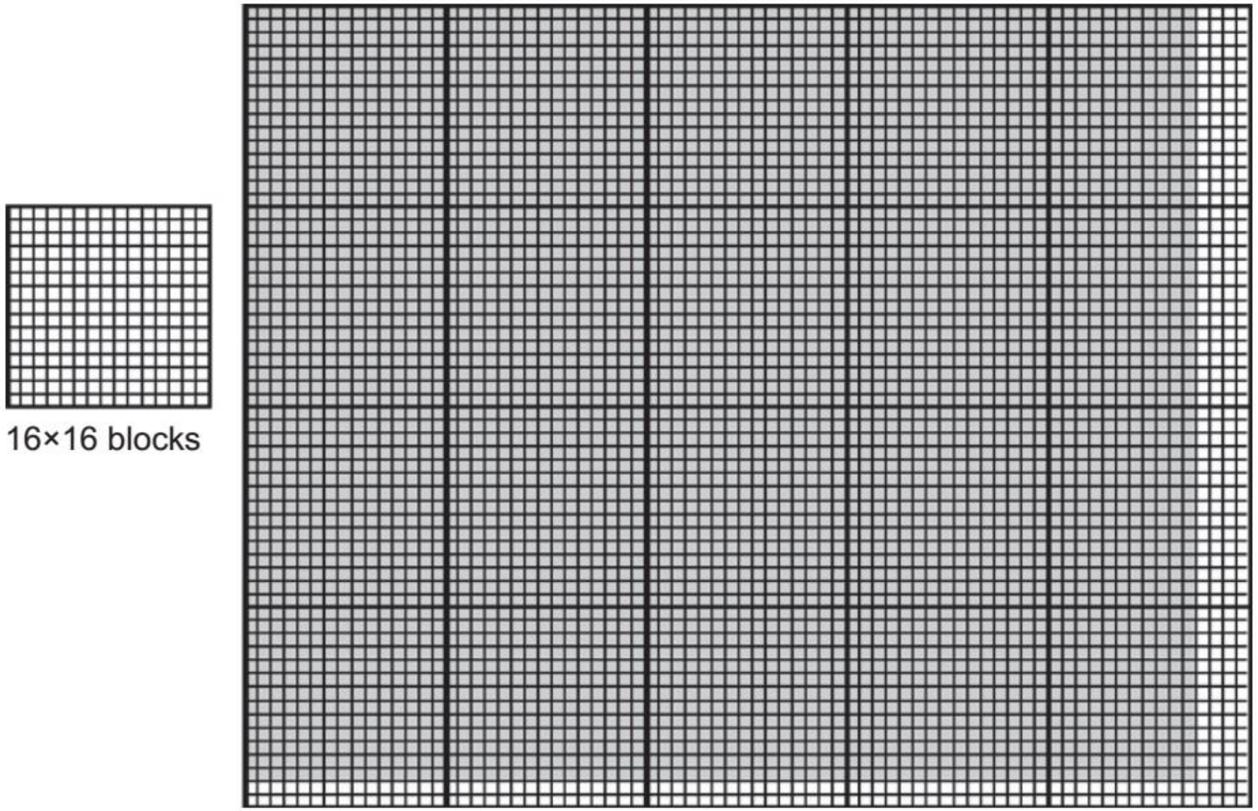
Choosing of a 16×16 block to process a 72×62 picture

→How many blocks in x and y direction?

→How many blocks totally?

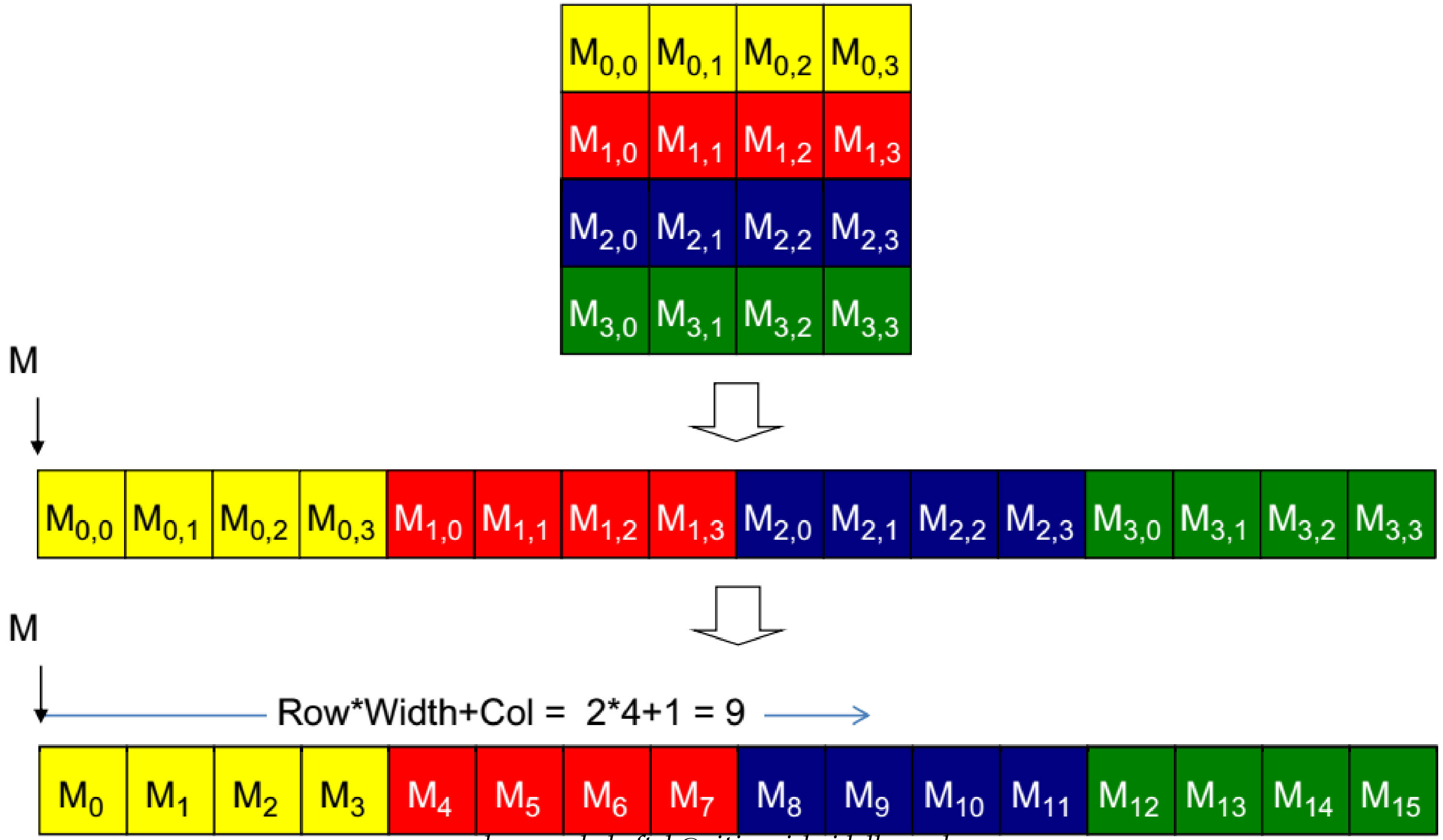
→How many threads will be generated?

→How many extra threads will be generated?



Mapping Threads to Multidimensional data

- Access to an element within the 2-D array (Recall from ANSI C)



Mapping Threads to Multidimensional data

- Access to an element within the 2-D array

```
__global__ void PictureKernell(float* d_Pin, float* d_Pout, int n, int m) {

  // Calculate the row # of the d_Pin and d_Pout element to process
  int Row = blockIdx.y*blockDim.y + threadIdx.y;

  // Calculate the column # of the d_Pin and d_Pout element to process
  int Col = blockIdx.x*blockDim.x + threadIdx.x;

  // each thread computes one element of d_Pout if in range
  if ((Row < m) && (Col < n)) {
    d_Pout[Row*n+Col] = 2*d_Pin[Row*n+Col];
  }

}
```

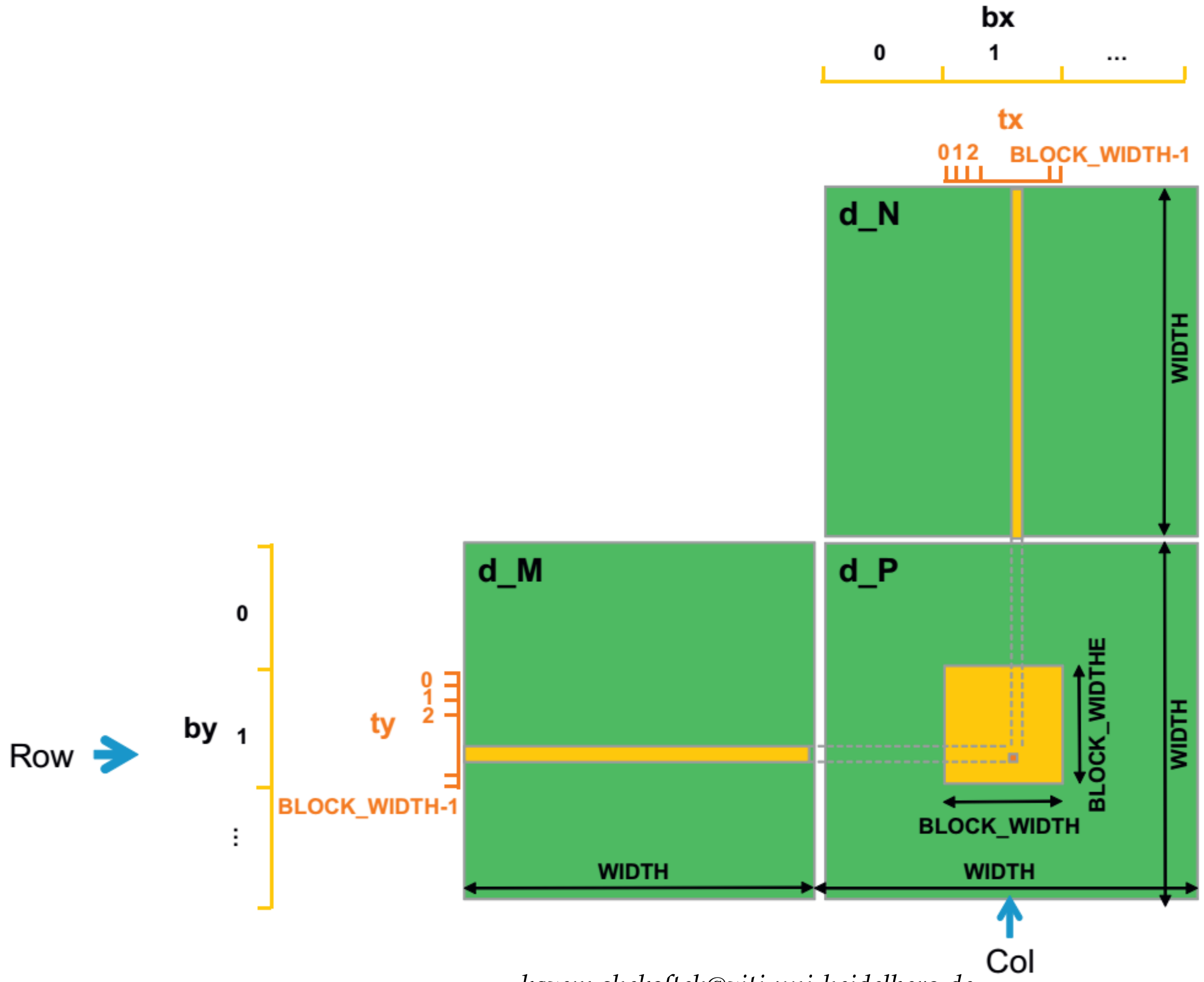
Matrix Multiplication Kernel

- We have studied **`vecAddkernel()`** and a sample **`pictureKernel()`** where each thread performs only one floating-point arithmetic operation.
- Matrix-matrix multiplication between an I×J matrix **`d_M`** and a J×K matrix **`d_N`** produces an I×K matrix **`d_P`**.
- For simplicity, we will limit our discussion to square matrices, where I=J=K (hereinafter shown as **`WIDTH`**).
- Each element of the product matrix **`d_P`** is an inner product of a row of **`d_M`** and a column of **`d_N`**.
- We map threads to **`d_P`** elements with the same approach as what we used for **`pictureKernel()`**.
  - each thread is responsible for calculating one **`d_P`** element.

Matrix Multiplication Kernel

- The **`d_P`** element calculated by a thread is
  - in row blockIdx.y × blockDim.y + threadIdx.y
  - in column blockIdx.x × blockDim.x + threadIdx.x

Matrix Multiplication Kernel



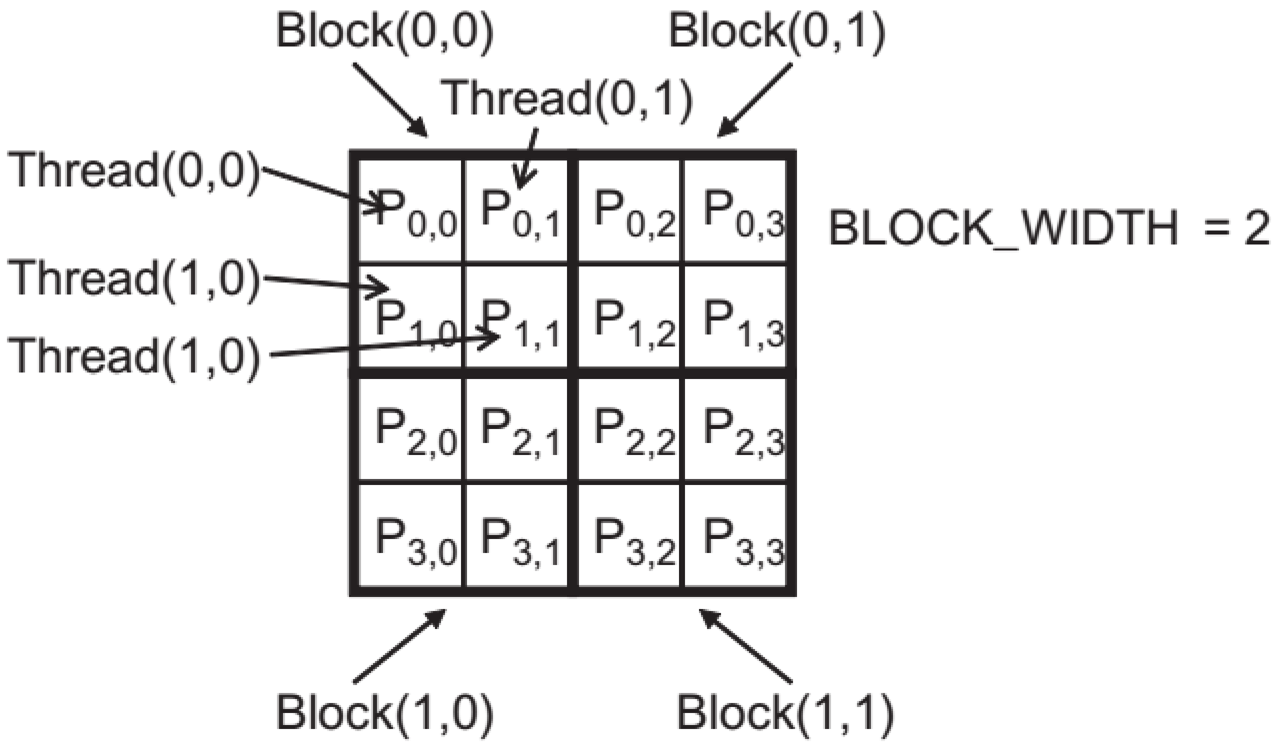
Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width) {

  // Calculate the row index of the d_Pelement and d_M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;

  // Calculate the column index of d_P and d_N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;

  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (intk = 0; k < Width; ++k) {
      Pvalue += d_M[Row*Width+k]*d_N[k*Width+Col];
    }
    d_P[Row*Width+Col] = Pvalue;
  }

}
```

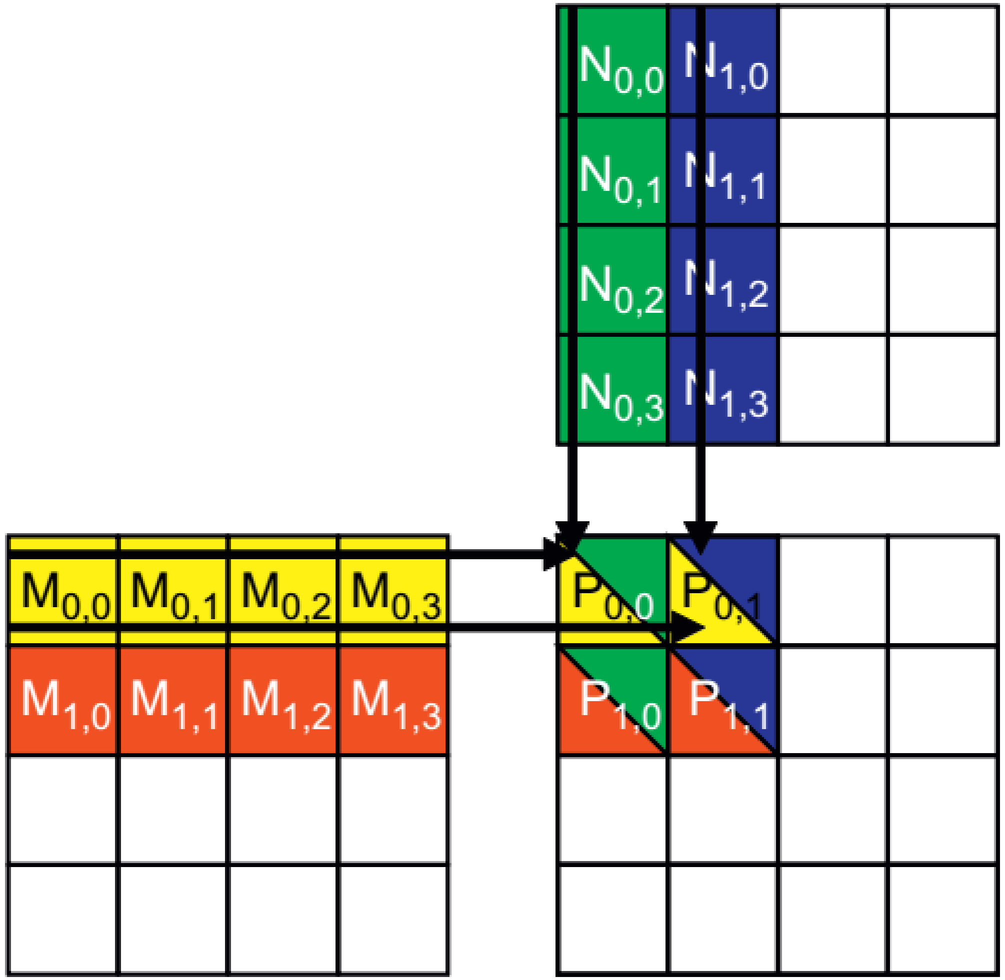
Matrix Multiplication Kernel

```
#define BLOCK_WIDTH 16

// Setup the execution configuration
    int NumBlocks = Width/BLOCK_WIDTH;
    if (Width % BLOCK_WIDTH) NumBlocks++;
    dim3 dimGrid(NumBlocks, NumbBlocks);
    dim3 dimBlock(BLOCK_WIDTH, BLOCK_WIDTH);

// Launch the device computation threads!
matrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

Matrix Multiplication Kernel



Matrix Multiplication Kernel



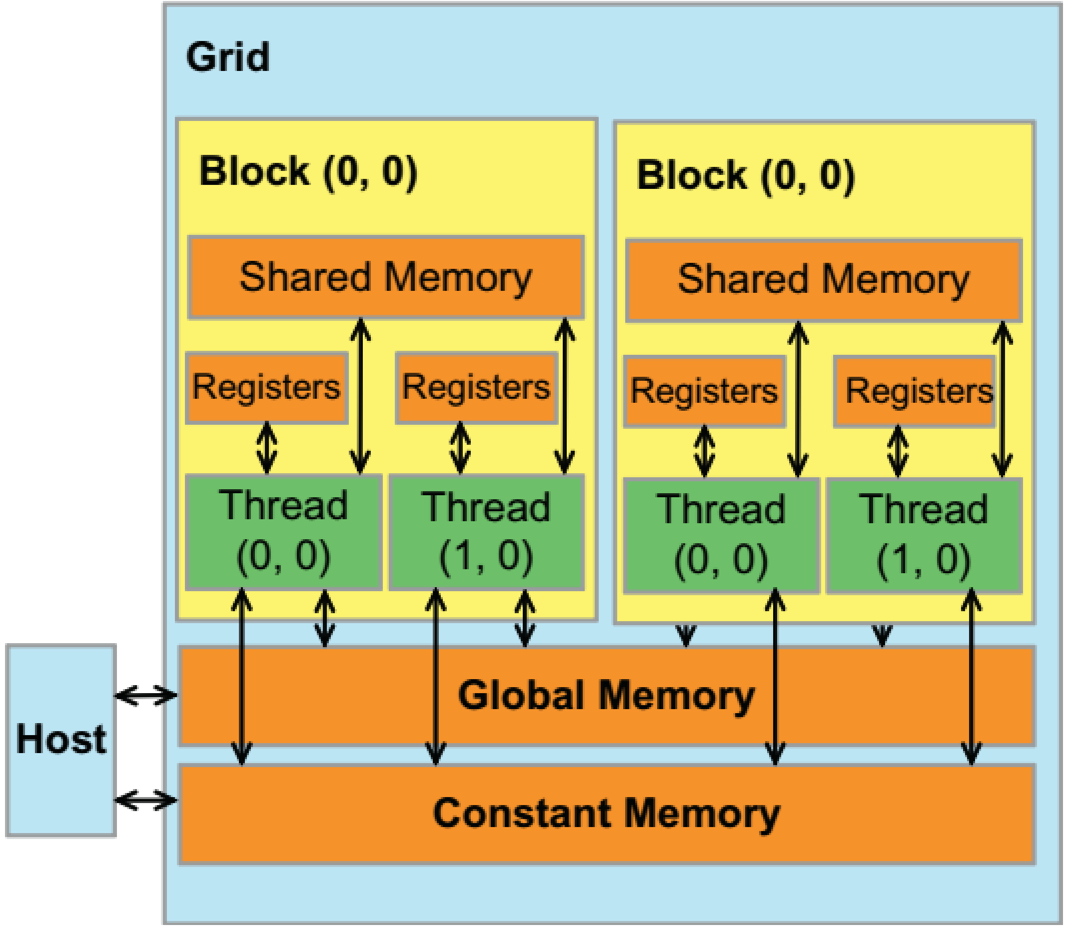
CUDA Device Memory

Device code can:

- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can

- Transfer data to/from per grid global and constant memories



CUDA Device Memory

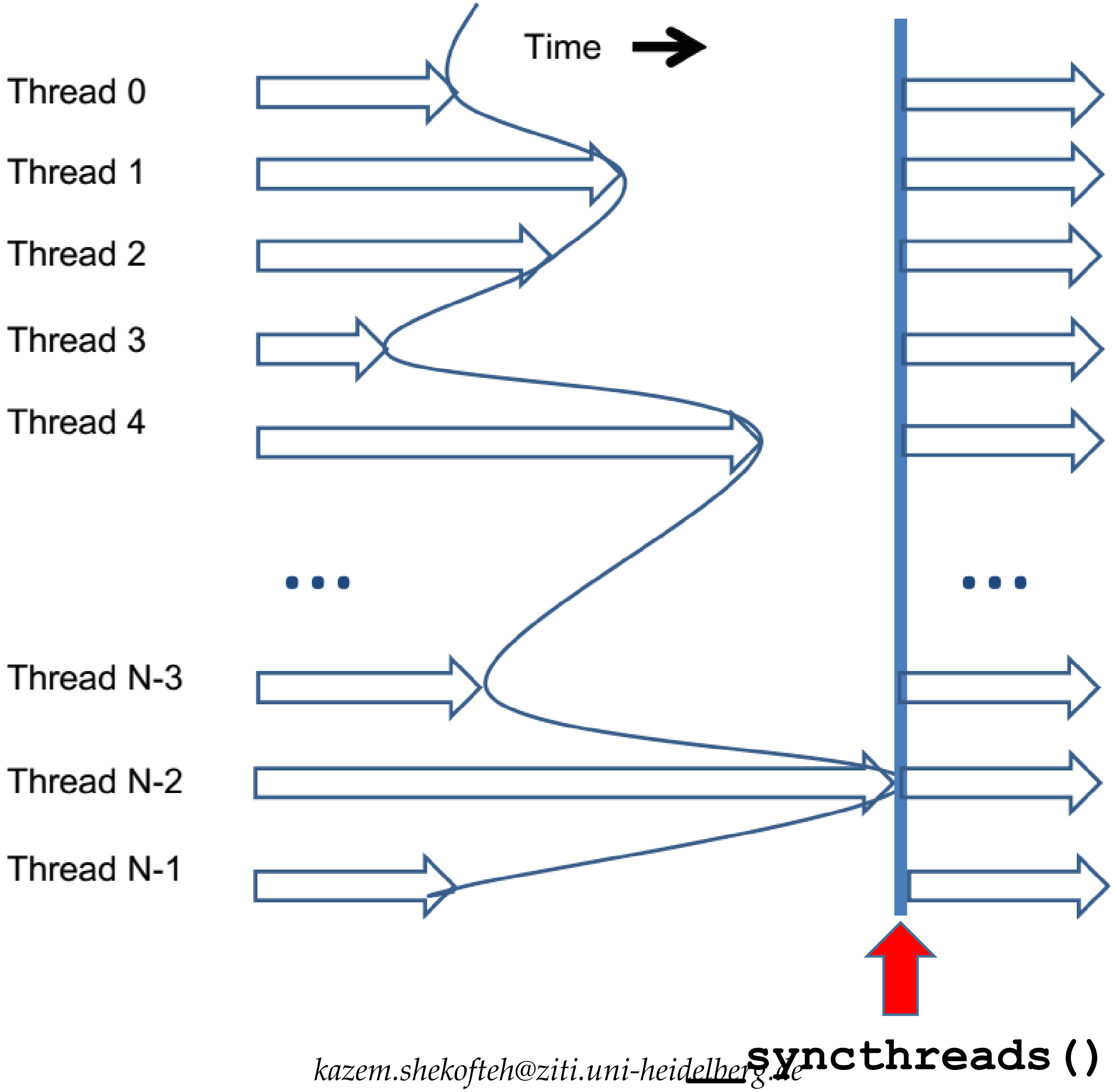
Table 5.1 CUDA Variable Type Qualifiers

Variable Declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

Synchronization

- How to coordinate the execution of multiple threads?
- CUDA allows threads in the **same block** to coordinate their activities using a barrier synchronization function `__syncthreads()`.
- When a kernel function calls `__syncthreads()`, all threads in a block will be held at the calling location until every thread in the block reaches the location.
- This ensures that all threads in a block have completed a phase of their execution of the kernel before any of them can move on to the next phase.
- Example: A group of friends shopping together …

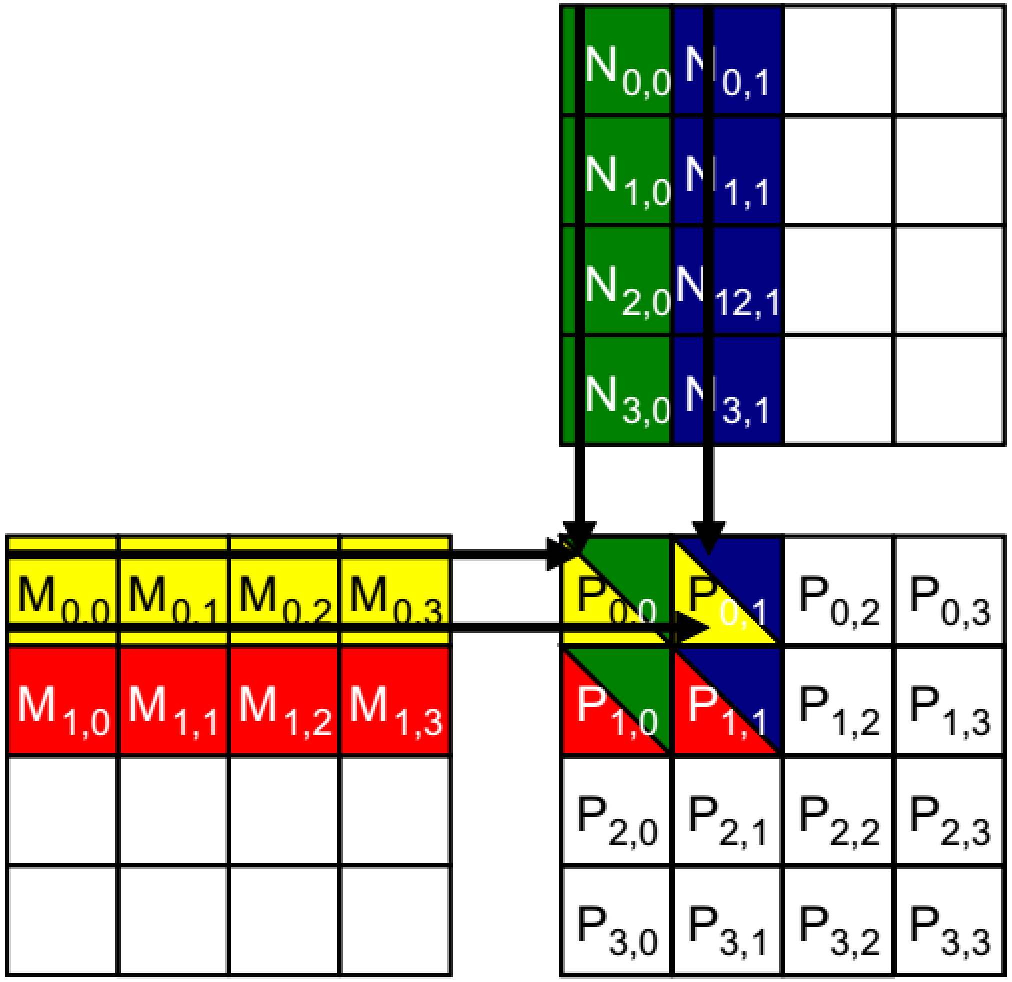
Synchronization



Reducing Global Memory Traffic

- Global memory is large but slow.
- Shared memory is small but fast.
- A common strategy is to partition the data into subsets called *tiles* so that each *tile* fits into the shared memory.

Reducing Global Memory Traffic

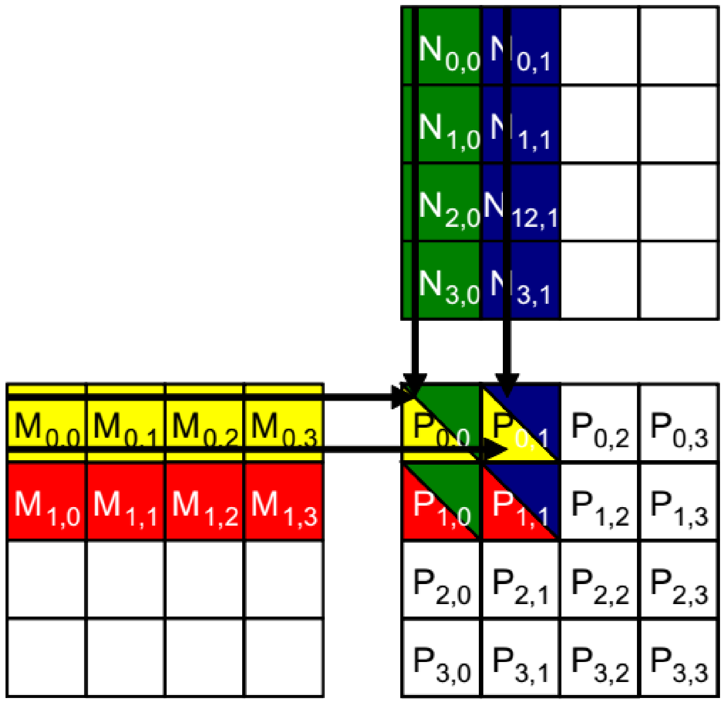


Reducing Global Memory Traffic

- Several redundant access to data

Access order

$thread_{0,0}$	$M_{0,0}$ * $N_{0,0}$	$M_{0,1}$ * $N_{1,0}$	$M_{0,2}$ * $N_{2,0}$	$M_{0,3}$ * $N_{3,0}$
$thread_{0,1}$	$M_{0,0}$ * $N_{0,1}$	$M_{0,1}$ * $N_{1,1}$	$M_{0,2}$ * $N_{2,1}$	$M_{0,3}$ * $N_{3,1}$
$thread_{1,0}$	$M_{1,0}$ * $N_{0,0}$	$M_{1,1}$ * $N_{1,0}$	$M_{1,2}$ * $N_{2,0}$	$M_{1,3}$ * $N_{3,0}$
$thread_{1,1}$	$M_{1,0}$ * $N_{0,1}$	$M_{1,1}$ * $N_{1,1}$	$M_{1,2}$ * $N_{2,1}$	$M_{1,3}$ * $N_{3,1}$

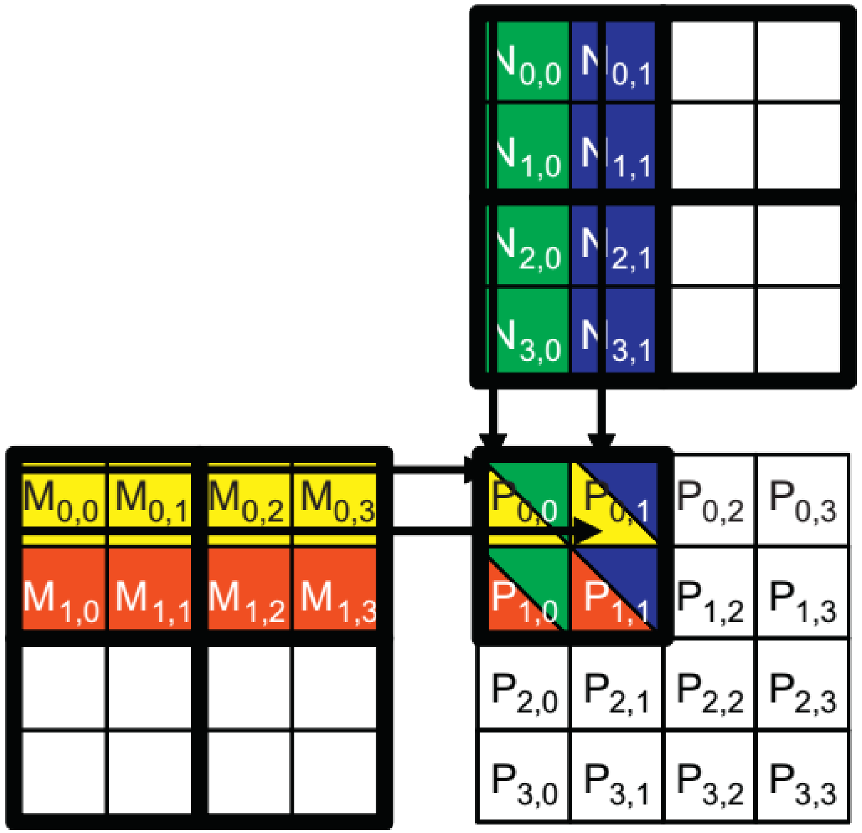


A Tiled Matrix-Matrix Multiplication Kernel

- An algorithm where threads collaborate to reduce the traffic to the global memory.
- Threads collaboratively load **`M`** and **`N`** elements into the **shared memory** before they individually use these elements in their dot product calculation.
- The size of shared memory is quite small!

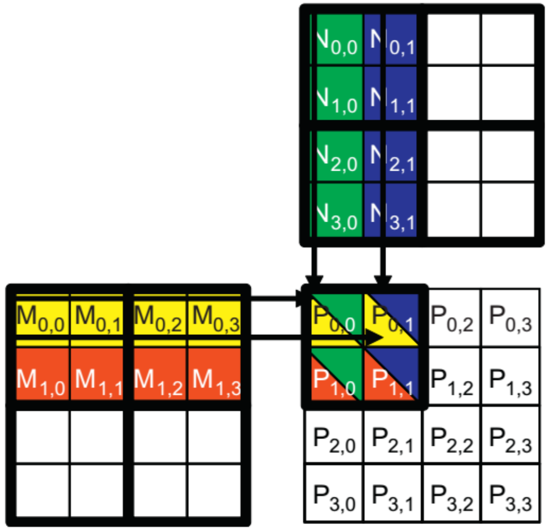
A Tiled Matrix-Matrix Multiplication Kernel

- Assume that we divide the **`M`** and **`N`** matrices into 2×2 tiles



A Tiled Matrix-Matrix Multiplication Kernel

- The dot product calculations performed by each thread are now divided into phases.
- In each phase, all threads in a block collaborate to load a tile of **M** elements and a tile of **N** elements into the shared memory.
- This is done by having every thread in a block to load one **M** element and one **N** element into the shared memory



A Tiled Matrix-Matrix Multiplication Kernel

	Phase 1		Phase 2	Phase 1		Phase 2
$\text{thread}_{0,0}$	$\mathbf{M}_{0,0} \downarrow \text{Mds}_{0,0}$	$\mathbf{N}_{0,0} \downarrow \text{Nds}_{0,0}$	$\text{PValue}_{0,0}$ += $\text{Mds}_{0,0}{}^*\text{Nds}_{0,0}$ + $\text{Mds}_{0,1}{}^*\text{Nds}_{1,0}$	$\mathbf{M}_{0,2} \downarrow \text{Mds}_{0,0}$	$\mathbf{N}_{2,0} \downarrow \text{Nds}_{0,0}$	$\text{PValue}_{0,0}$ += $\text{Mds}_{0,0}{}^*\text{Nds}_{0,0}$ + $\text{Mds}_{0,1}{}^*\text{Nds}_{1,0}$
$\text{thread}_{0,1}$	$\mathbf{M}_{0,1} \downarrow \text{Mds}_{0,1}$	$\mathbf{N}_{0,1} \downarrow \text{Nds}_{1,0}$	$\text{PValue}_{0,1}$ += $\text{Mds}_{0,0}{}^*\text{Nds}_{0,1}$ + $\text{Mds}_{0,1}{}^*\text{Nds}_{1,1}$	$\mathbf{M}_{0,3} \downarrow \text{Mds}_{0,1}$	$\mathbf{N}_{2,1} \downarrow \text{Nds}_{0,1}$	$\text{PValue}_{0,1}$ += $\text{Mds}_{0,0}{}^*\text{Nds}_{0,1}$ + $\text{Mds}_{0,1}{}^*\text{Nds}_{1,1}$
$\text{thread}_{1,0}$	$\mathbf{M}_{1,0} \downarrow \text{Mds}_{1,0}$	$\mathbf{N}_{1,0} \downarrow \text{Nds}_{1,0}$	$\text{PValue}_{1,0}$ += $\text{Mds}_{1,0}{}^*\text{Nds}_{0,0}$ + $\text{Mds}_{1,1}{}^*\text{Nds}_{1,0}$	$\mathbf{M}_{1,2} \downarrow \text{Mds}_{1,0}$	$\mathbf{N}_{3,0} \downarrow \text{Nds}_{1,0}$	$\text{PValue}_{1,0}$ += $\text{Mds}_{1,0}{}^*\text{Nds}_{0,0}$ + $\text{Mds}_{1,1}{}^*\text{Nds}_{1,0}$
$\text{thread}_{1,1}$	$\mathbf{M}_{1,1} \downarrow \text{Mds}_{1,1}$	$\mathbf{N}_{1,1} \downarrow \text{Nds}_{1,1}$	$\text{PValue}_{1,1}$ += $\text{Mds}_{1,0}{}^*\text{Nds}_{0,1}$ + $\text{Mds}_{1,1}{}^*\text{Nds}_{1,1}$	$\mathbf{M}_{1,3} \downarrow \text{Mds}_{1,1}$	$\mathbf{N}_{3,1} \downarrow \text{Nds}_{1,1}$	$\text{PValue}_{1,1}$+= $\text{Mds}_{1,0}{}^*\text{Nds}_{0,1}$ + $\text{Mds}_{1,1}{}^*\text{Nds}_{1,1}$

time ⟶

A Tiled Matrix-Matrix Multiplication Kernel

- Note that each value in the shared memory is used twice.
- For example, the $M_{1,1}$ value, loaded by $thread_{1,1}$ into $Mds_{1,1}$, is used twice, once by $thread_{0,1}$ and once by $thread_{1,1}$.
- This is done by having every thread in a block to load one **`M`** element and one **`N`** element into the shared memory.
- By loading each global memory value into shared memory so that it can be used multiple times, we reduce the number of accesses to the global memory.
- In this case, we reduce the number of accesses to the global memory by **half**.

A Tiled Matrix-Matrix Multiplication Kernel

- Note that the calculation of each dot product is now performed in two phases, shown as phase 1 and phase 2.
- In general, if an input matrix is of dimension **`N`** and the tile size is **`TILE_WIDTH`**, the dot product would be performed in **`N/TILE_WIDTH`** phases.
- The creation of these phases is key to the reduction of accesses to the global memory.

A Tiled Matrix-Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
     int Width) {

     __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
     __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

     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_WIDTH + ty;
     int Col = bx * TILE_WIDTH + tx;

     float Pvalue = 0;
     // Loop over the d_M and d_N tiles required to compute d_P element
     for (int m = 0; m < Width/TILE_WIDTH; ++m) {

       // Coolaborative loading of d_M and d_N tiles into shared memory
       Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
       Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];


       for (int k = 0; k < TILE_WIDTH; ++k) {
         Pvalue += Mds[ty][k] * Nds[k][tx];
       }

     }
     d_P[Row*Width + Col] = Pvalue;
   }
```

kazem.shekofteh@ziti.uni-heidelberg.de

A Tiled Matrix-Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
      int Width) {

      __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
      __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

      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_WIDTH + ty;
      int Col = bx * TILE_WIDTH + tx;

      float Pvalue = 0;
      // Loop over the d_M and d_N tiles required to compute d_P element
      for (int m = 0; m < Width/TILE_WIDTH; ++m) {

        // Coolaborative loading of d_M and d_N tiles into shared memory
        Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
        Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];


        for (int k = 0; k < TILE_WIDTH; ++k) {
          Pvalue += Mds[ty][k] * Nds[k][tx];
        }

      }
      d_P[Row*Width + Col] = Pvalue;
    }
```

A Tiled Matrix-Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
     int Width) {

     __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
     __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

     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_WIDTH + ty;
     int Col = bx * TILE_WIDTH + tx;

     float Pvalue = 0;
     // Loop over the d_M and d_N tiles required to compute d_P element
     for (int m = 0; m < Width/TILE_WIDTH; ++m) {

       // Coolaborative loading of d_M and d_N tiles into shared memory
       Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
       Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];


       for (int k = 0; k < TILE_WIDTH; ++k) {
         Pvalue += Mds[ty][k] * Nds[k][tx];
       }

     }
     d_P[Row*Width + Col] = Pvalue;
   }
```

Phase1

A Tiled Matrix-Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
    int Width) {

    __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
    __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

    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_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;

    float Pvalue = 0;
    // Loop over the d_M and d_N tiles required to compute d_P element
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {

      // Coolaborative loading of d_M and d_N tiles into shared memory
      Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
      Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];


      for (int k = 0; k < TILE_WIDTH; ++k) {
        Pvalue += Mds[ty][k] * Nds[k][tx];
      }

    }
    d_P[Row*Width + Col] = Pvalue;
  }
```

Phase2

A Tiled Matrix-Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
     int Width) {

     __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
     __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

     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_WIDTH + ty;
     int Col = bx * TILE_WIDTH + tx;

     float Pvalue = 0;
     // Loop over the d_M and d_N tiles required to compute d_P element
     for (int m = 0; m < Width/TILE_WIDTH; ++m) {

       // Coolaborative loading of d_M and d_N tiles into shared memory
       Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
       Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];


       for (int k = 0; k < TILE_WIDTH; ++k) {
         Pvalue += Mds[ty][k] * Nds[k][tx];
       }

     }
     d_P[Row*Width + Col] = Pvalue;
   }
```

There are some problems with this code!

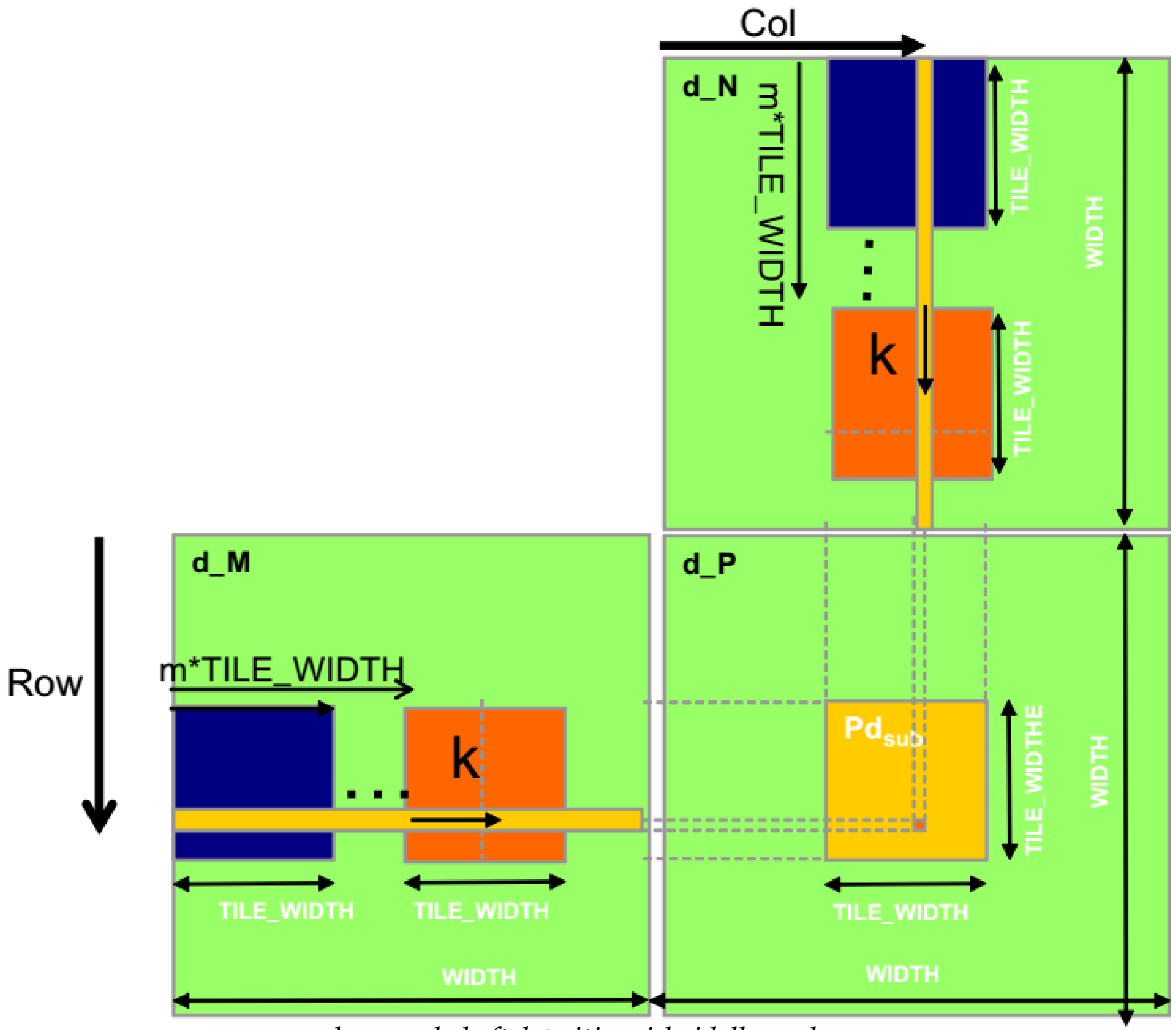
A Tiled Matrix-Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P,
     int Width) {

     __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
     __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

     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_WIDTH + ty;
     int Col = bx * TILE_WIDTH + tx;

     float Pvalue = 0;
     // Loop over the d_M and d_N tiles required to compute d_P element
     for (int m = 0; m < Width/TILE_WIDTH; ++m) {

       // Coolaborative loading of d_M and d_N tiles into shared memory
       Mds[ty][tx] = d_M[Row*Width + m*TILE_WIDTH + tx];
       Nds[ty][tx] = d_N[(m*TILE_WIDTH + ty)*Width + Col];
       __syncthreads();

       for (int k = 0; k < TILE_WIDTH; ++k) {
         Pvalue += Mds[ty][k] * Nds[k][tx];
       }
       __syncthreads();
     }
     d_P[Row*Width + Col] = Pvalue;
   }
```

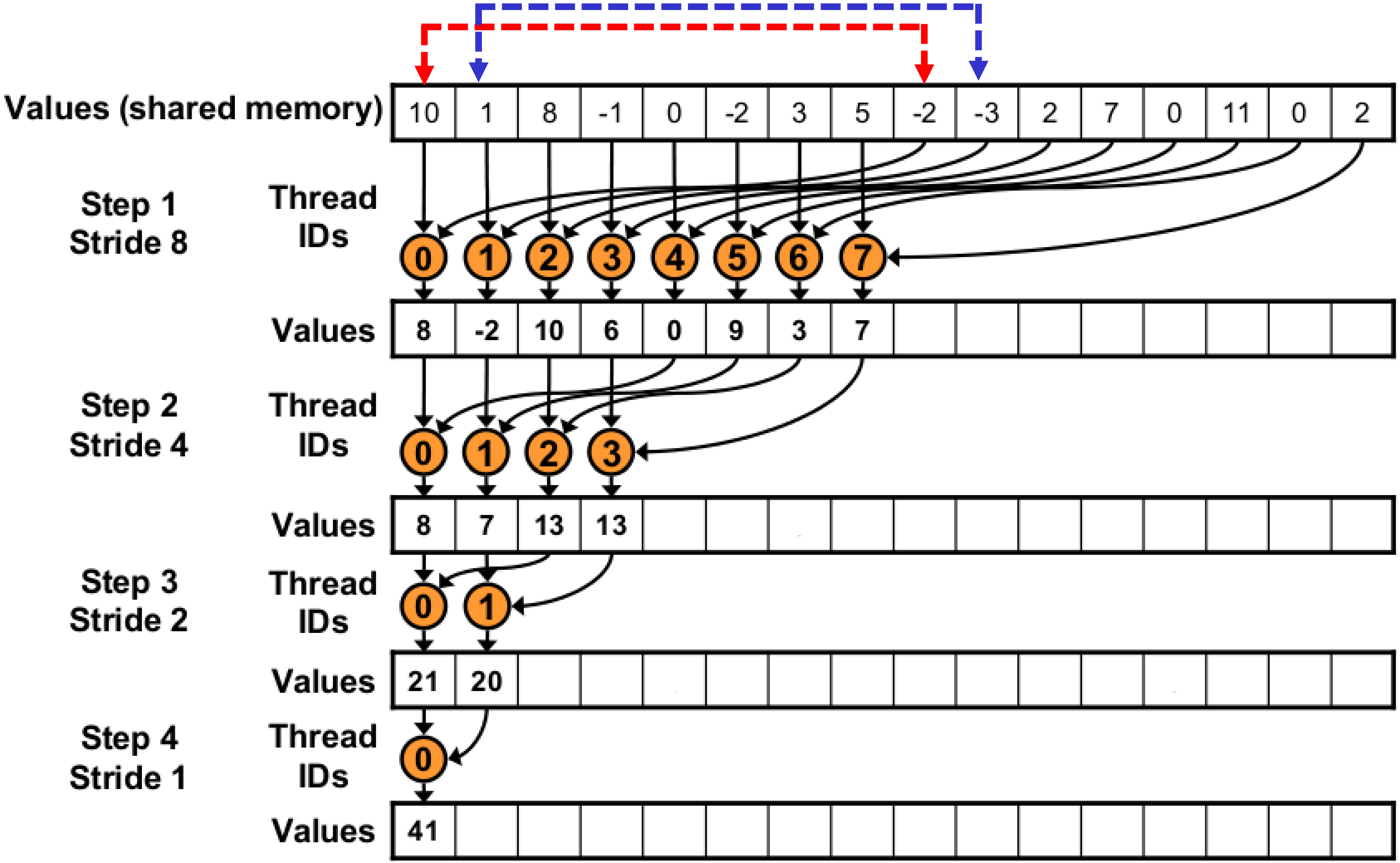
synchronization

synchronization

A Tiled Matrix-Matrix Multiplication Kernel



Array Reduction Kernel



Array Sum (Reduction) Kernel

```
__global__ void ArraySum(int *input, int *output)
{
  __shared__ int* partial_sums;
  int group_size = blockDim.x;
  int index = threadIdx.x + blockIdx.x * blockDim.x
  partial_sums[threadIdx.x] = input[index];

  for(int i = group_size/2; i>0; i/=2)
  {
      if(threadIdx.x < i)
         partial_sums[threadIdx.x] += partial_sums[threadIdx.x + i];
  }
  if(threadIdx.x == 0)
        output[index] = partial_sums[0];
}
```

Array Sum (Reduction) Kernel

```
__global__ void ArraySum(int *input, int *output)
{
  __shared__ int* partial_sums;
  int group_size = blockDim.x;
  int index = threadIdx.x + blockIdx.x * blockDim.x
  partial_sums[threadIdx.x] = input[index];

  for(int i = group_size/2; i>0; i/=2)
  {
      if(threadIdx.x < i)
         partial_sums[threadIdx.x] += partial_sums[threadIdx.x + i];
  }
  if(threadIdx.x == 0)
        output[index] = partial_sums[0];
}
```

There are some problems with this code!

Array Sum (Reduction) Kernel

```
__global__ void ArraySum(int *input, int *output)
{
  __shared__ int* partial_sums;
  int group_size = blockDim.x;
  int index = threadIdx.x + blockIdx.x * blockDim.x
  partial_sums[threadIdx.x] = input[index];
  __synchthreads();

  for(int i = group_size/2; i>0; i/=2)
  {
      if(threadIdx.x < i)
         partial_sums[threadIdx.x] += partial_sums[threadIdx.x + i];

      __synchthreads();
  }
  if(threadIdx.x == 0)
        output[index] = partial_sums[0];
}
```

Array Sum (Reduction) Kernel

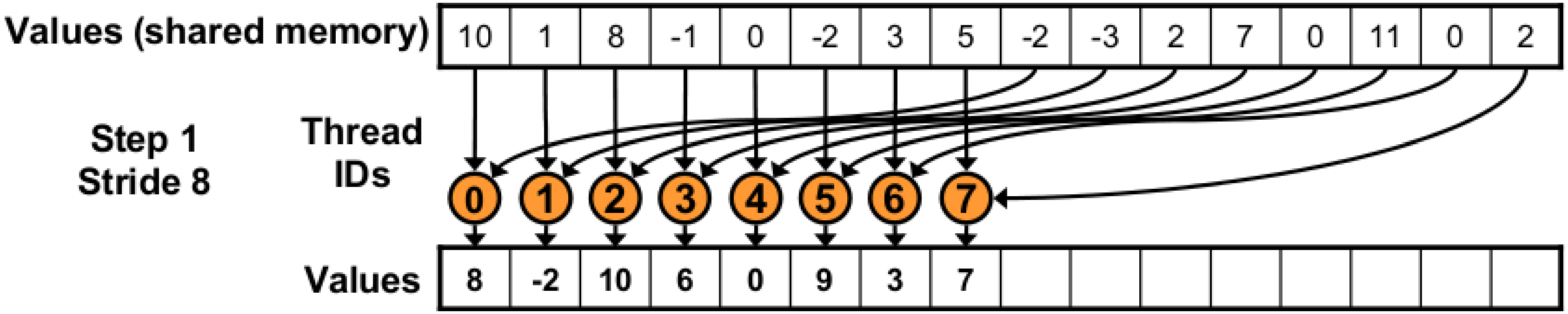
```
void main()
{
  …
  ArraySum<<<1, 16>>>(input, output)
  …
}
```

Values (shared memory)

10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2

size = 16

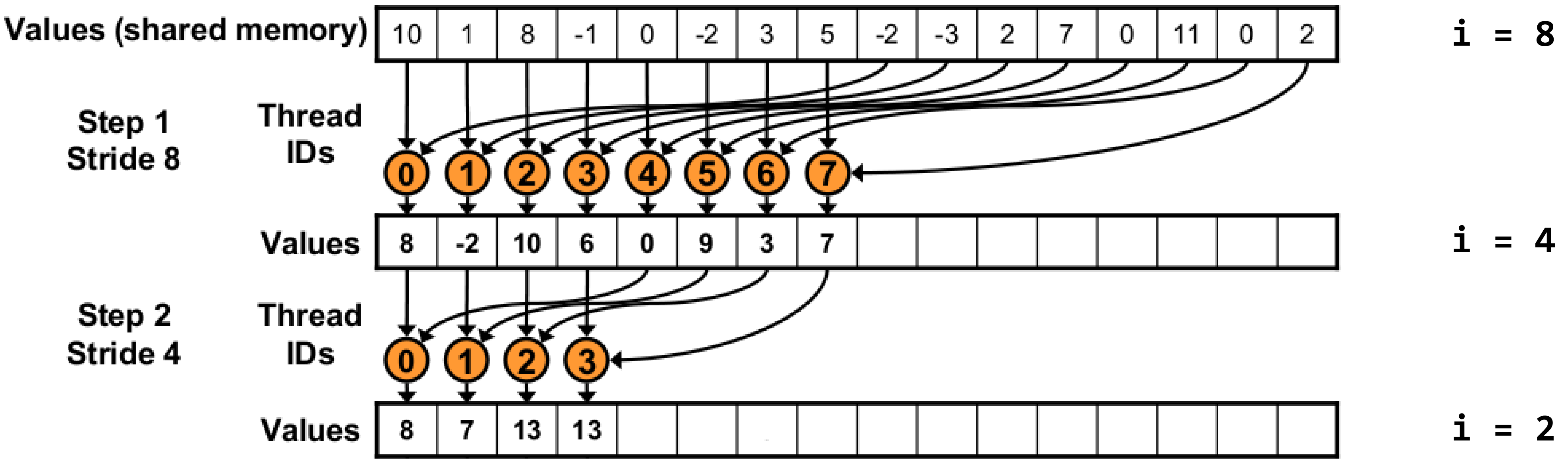
```
int group_size = blockDim.x;
int index = threadIdx.x + blockIdx.x * blockDim.x
partial_sums[threadIdx.x] = input[index];
__synchthreads();
```



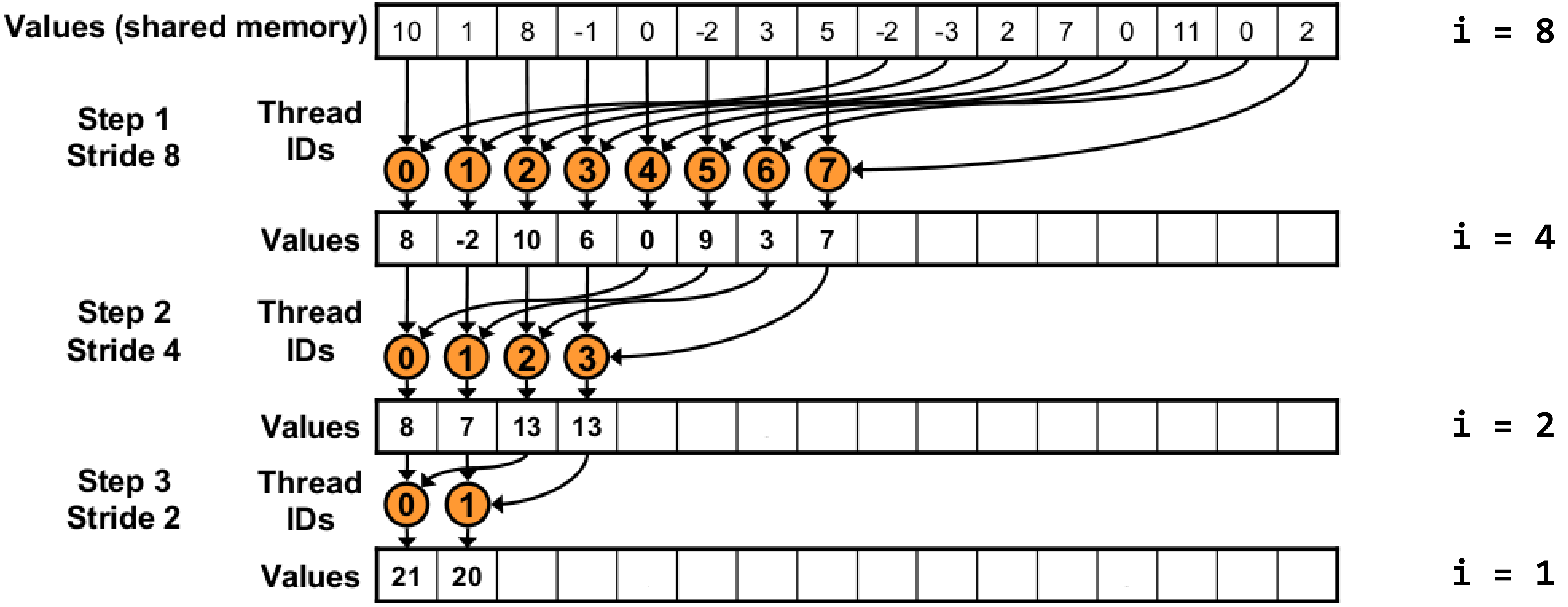
i = 8

i = 4

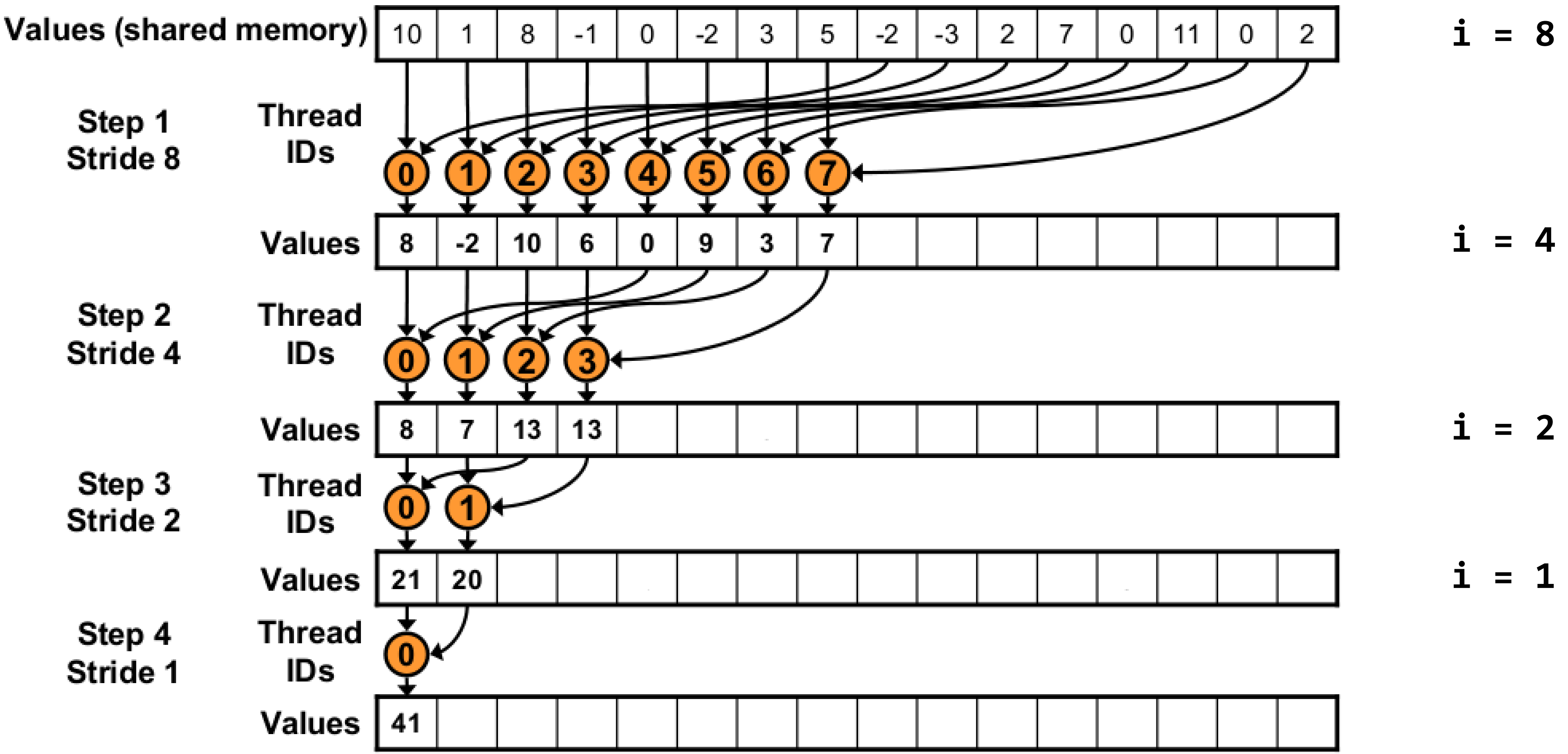
```
for(int i = group_size/2; i>0; i/=2)
  {
      if(threadIdx.x < i)
         partial_sums[threadIdx.x] += partial_sums[threadIdx.x + i];

      __synchthreads();
  }
```




```
for(int i = group_size/2; i>0; i/=2)
  {
      if(threadIdx.x < i)
         partial_sums[threadIdx.x] += partial_sums[threadIdx.x + i];

      __synchthreads();
  }
```




```
for(int i = group_size/2; i>0; i/=2)
  {
      if(threadIdx.x < i)
         partial_sums[threadIdx.x] += partial_sums[threadIdx.x + i];

      __synchthreads();
  }
```

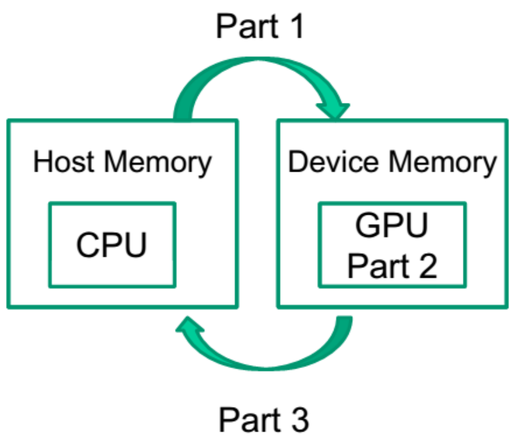



```
for(int i = group_size/2; i>0; i/=2)
  {
      if(threadIdx.x < i)
         partial_sums[threadIdx.x] += partial_sums[threadIdx.x + i];

      __synchthreads();
  }
```

Kernel Execution

- As described before, a kernel is launched similar to a function call in C with additional execution configuration parameters in <<< >>>.

```
#include <cuda.h>
…
void vecAdd(float* A, float*B, float* C, int n)
{
   int size = n* sizeof(float);
    float *A_d, *B_d, *C_d;
   …
1. // Allocate device memory for A, B, and C
   // copy A and B to device memory

2. // Kernel launch code – to have the device
   // to perform the actual vector addition

3. // copy C from the device memory
   // Free device vectors
}
```



Kernel Execution

- It should be noted that kernel launch commands are asynchronous to the host code.
- When a kernel is launched, the control returns back to the host immediately and the GPU starts its work in parallel to the host.

Kernel Execution

- What is wrong with this code?

```
...
float* d_output;
cudaMalloc(d_output);
clock_t start = clock();
myKernel<<<gs,bs>>>(d_input, d_output);
clock_t end = clock();
float seconds = (float)(end - start) / CLOCKS_PER_SEC;
...
```

Kernel Execution

- The host should be synchronized with the asynchronous kernel execution.

```
…
float* d_output;
cudaMalloc(d_output);
clock_t start = clock();
myKernel<<<gs,bs>>>(d_input, d_output);
cudaDeviceSynchronize();
clock_t end = clock();
float seconds = (float)(end - start) / CLOCKS_PER_SEC;
…
```

synchronization

Kernel Execution

- **cudaDeviceSynchronize()** Blocks until the device has completed all preceding requested tasks.
- This is a synchronization mechanism between the device and the host.
- What are its differences with **__synchthread**()?

Kernel Execution

- In the following example, kernel **`myKernel`** reads data from its first argument and writes to the second one.
- Therefore, is there any problem with this code?

```
...
float* d_input, d_output1, d_output2;
cudaMalloc(d_input); // and d_output1, d_output2
myKernel<<<gs1,bs1>>>(d_input, d_output1);
myKernel<<<gs2,bs2>>>(d_output1, d_output2);
...
```

Kernel Execution

- All CUDA commands including kernel launches and memory copies are executed serially, unless you explicitly specify to be executed in parallel.
- They are inserted into the same queue.

Summary

- Mapping threads to multi dimensional data
- Matrix-matrix multiplication as an example of the mapping
- Strategies to reduce the access to global memory
- Using shared memory to increase the performance
- Synchronization between threads and between CPU and GPU

Advanced Topics in GPU Computing

- Event recording and timing
- Streaming
- Unified memory
- __device__ functions

- Working with NVIDIA Visual Profiler

- Working with benchmarks
  - CUDA Sample SDK
  - Rodinia
  - Parboil