

# Introduction to GPU computing (2)

Computing Methods for Experimental Physics and Data Analysis  
Lecture 4

[gianluca.lamanna@unipi.it](mailto:gianluca.lamanna@unipi.it)

# Characteristics of GPU we are using: GeForce GTX650

CUDA Driver Version / Runtime Version	9.1 / 9.1
CUDA Capability Major/Minor version number:	3.0
<b>Total amount of global memory:</b>	981 MBytes (1028915200 bytes)
<b>( 2) Multiprocessors, (192) CUDA Cores/MP:</b>	384 CUDA Cores
<b>GPU Max Clock rate:</b>	1058 MHz (1.06 GHz)
Memory Clock rate:	2500 Mhz
Memory Bus Width:	128-bit
L2 Cache Size:	262144 bytes
Maximum Texture Dimension Size (x,y,z)	1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
Maximum Layered 1D Texture Size, (num) layers	1D=(16384), 2048 layers
Maximum Layered 2D Texture Size, (num) layers	2D=(16384, 16384), 2048 layers
Total amount of constant memory:	65536 bytes
<b>Total amount of shared memory per block:</b>	49152 bytes
Total number of registers available per block:	65536
<b>Warp size:</b>	32
Maximum number of threads per multiprocessor:	2048
Maximum number of threads per block:	1024
Max dimension size of a thread block (x,y,z):	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z):	(2147483647, 65535, 65535)
Maximum memory pitch:	2147483647 bytes
Texture alignment:	512 bytes
Concurrent copy and kernel execution:	Yes with 1 copy engine(s)

...



# HelloWorld

- HelloWorld:

→ Try to change the kernel launch parameters

```
#include <cuda.h>
#include <stdio.h>

__global__ void mykernel(void) {
    printf( "Hello World from GPU! (block: %d thread:%d)\n" ,blockIdx.x,threadIdx.x);
}

int main(void) {
    mykernel <<<1,5>>>();
    cudaDeviceSynchronize();
    printf("Hello World from Host!\n");
    return 0;
}
```

# Vector Sum (Serial)

- We want to sum two vectors of 1048576 elements each
  - First we will try to write a «**serial**» version of the code
  - Due to the presence of cuda functions to measure the time, this code must be compiled with nvcc
- Time: 5.0 ms

```
#include <stdio.h>
#define N 1048576

void RandomVector(int *a, int nn){
    for (int i=0;i<nn;i++) {
        a[i]=rand()%100+1;
    }
}

//serial sum
void VecAddSerial(int *a, int *b, int *c){
    for (int i=0;i<N;i++){
        c[i] = a[i]+b[i];
    }
}

int main(void) {
    int *h_a, *h_b, *h_c;
    int size = N*sizeof(int);
    float time;
    cudaEvent_t start,stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);

    //Alloc in Host (and filling)
    h_a = (int *)malloc(size);
    h_b = (int *)malloc(size);
    h_c = (int *)malloc(size);
    RandomVector(h_a,N);
    RandomVector(h_b,N);
```

```
//start time
    cudaEventRecord(start);

    //Launch Serial Sum on CPU
    VecAddSerial(h_a,h_b,h_c);

    //stop time
    cudaEventRecord(stop);
    cudaEventSynchronize(stop);
    cudaEventElapsedTime(&time, start, stop);

    //Print Result
    // for(int i=0;i<N;i++){
    //     printf ("%d) h_a:%d h_b:%d
h_c:%d\n",
    //     i,h_a[i],h_b[i],h_c[i]);
    // }

    //print time
    printf("Time: %3.5f ms\n",time);

    //Cleanup
    free(h_a);
    free(h_b);
    free(h_c);
    return(0);
}
```

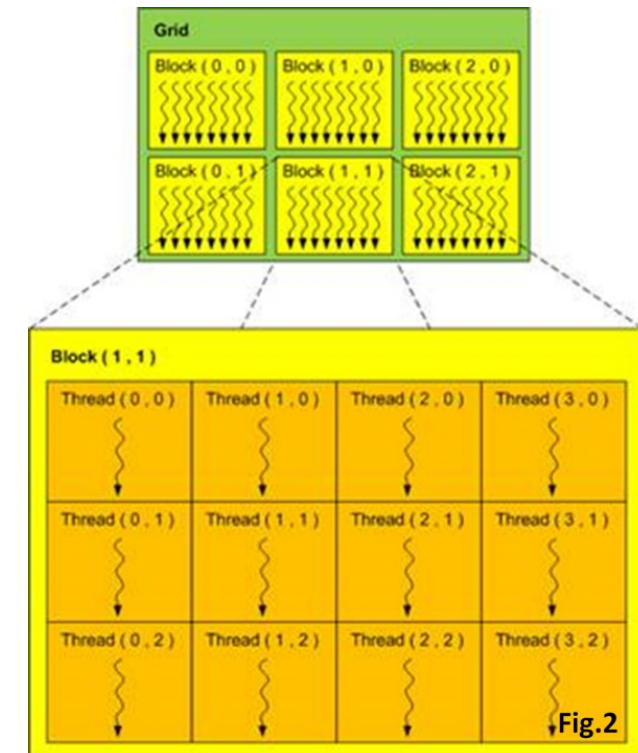
# Vector Sum (parallel)

- Let's try to parallelize, by using several blocks
- Remember to copy data from host to device and results back
- The results is not what we expect → Time: 17 ms !!!
- Why????

```
<skip>
//kernel
__global__ void VecAddGpu(int *a, int *b, int *c){
    c[blockIdx.x] = a[blockIdx.x]+b[blockIdx.x];
}
<skip>
//Alloc in Device
cudaMalloc((void **)&d_a, size);
cudaMalloc((void **)&d_b, size);
cudaMalloc((void **)&d_c, size);

//Copy input vectors form host to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
<skip>
```

```
<skip>
//Launch Kernel on GPU
VecAddGpu<<<N,1>>>(d_a,d_b,d_c);
cudaDeviceSynchronize();
<skip>
//Copy back the results
cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);
<skip>
//Cleanup
free(h_a);
free(h_b);
free(h_c);
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);
```



## Vector Sum (parallel): 2<sup>o</sup> attempt

- Then let's try to use one single block and N Threads
- Time=0.007 ms
- SpeedUp = 714 !!!
- Uhmmmmmmmmmmmmmmmmmm
- A reasonable speedup is around 100 or less
- Try to print something:
  - ➔ The results
  - ➔ The error code
- Try to have a look to the maximum size of threads per block

*<skip>*

```
//Launch Kernel on GPU
```

```
VecAddGpu<<<1,N>>>(d_a,d_b,d_c);
cudaDeviceSynchronize();
```

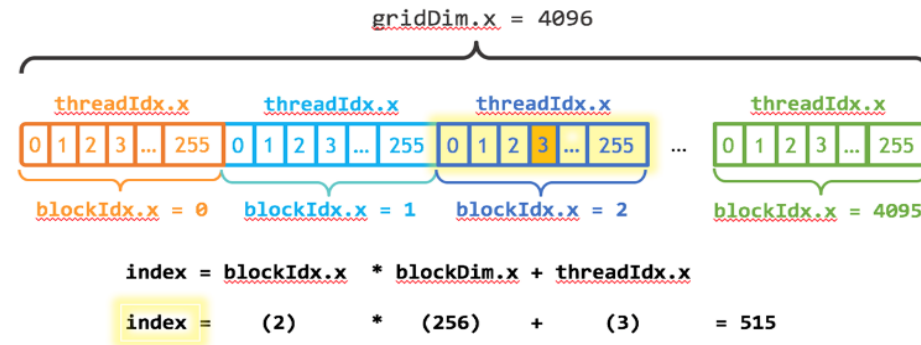
# Vector Sum (parallel): final attempt

- Use both threads and blocks
  - The total number of threads must be equal to the number of elements in the vectors
  - Define an «index» by using the block/thread identifier
  - The kernel must be adapted to this structure
- Time=0.45 ms
- Without errors

```
<skip>
#define THREADS_PER_BLOCK 128

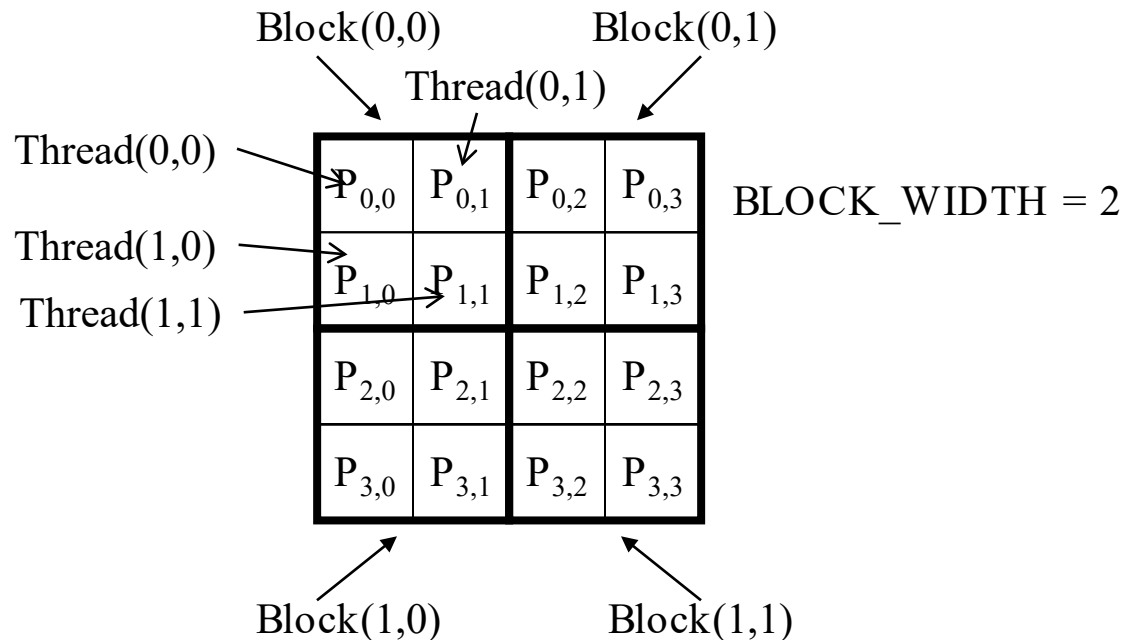
<skip>
//kernel
__global__ void VecAddGpu(int *a, int *b, int *c){
    int index = threadIdx.x + blockIdx.x*blockDim.x;
    c[index] = a[index]+b[index];
}

<skip>
//Launch Kernel on GPU
VecAddGpu<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>>(d_a,d_b,d_c);
cudaDeviceSynchronize();
```



# Matrix Multiplication

- Assume you want to multiply two large matrices
- 2D structure of threads and blocks
- Each thread computes one element of the matrix
- Use the blocks to subdivide the matrix in sub-blocks

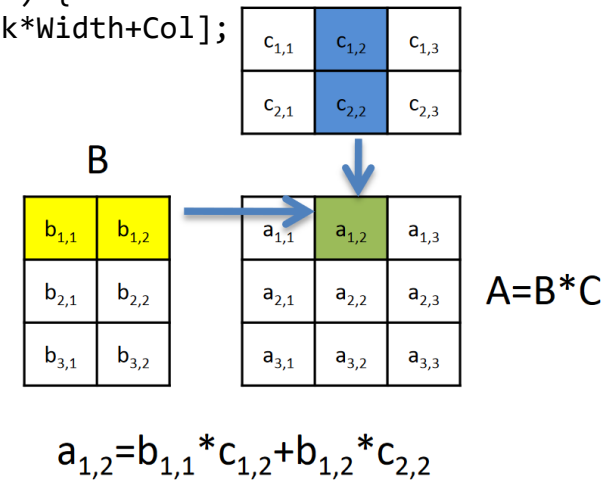


```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int
Width) {
```

```
    // Calculate the row index of the P element and M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
```

```
    // Calculate the column index of P and N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;
```

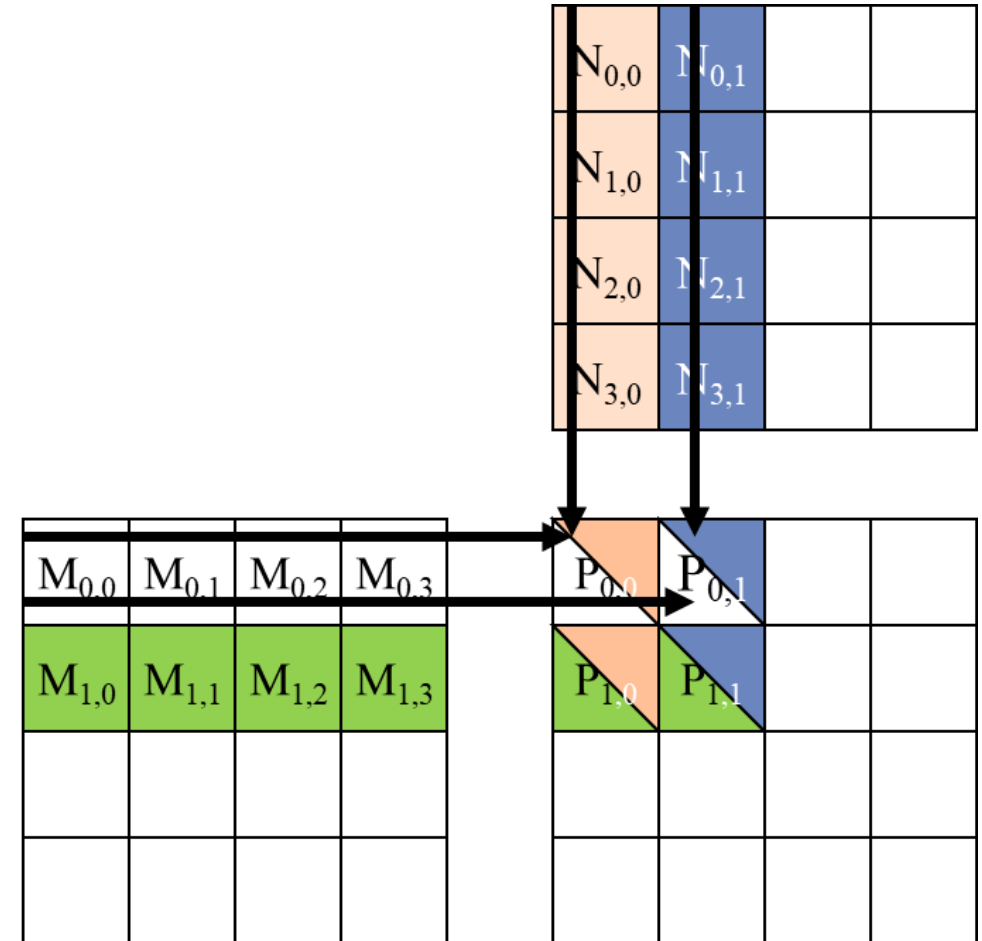
```
    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k) {
            Pvalue += M[Row*Width+k]*N[k*Width+Col];
        }
        P[Row*Width+Col] = Pvalue;
    }
```





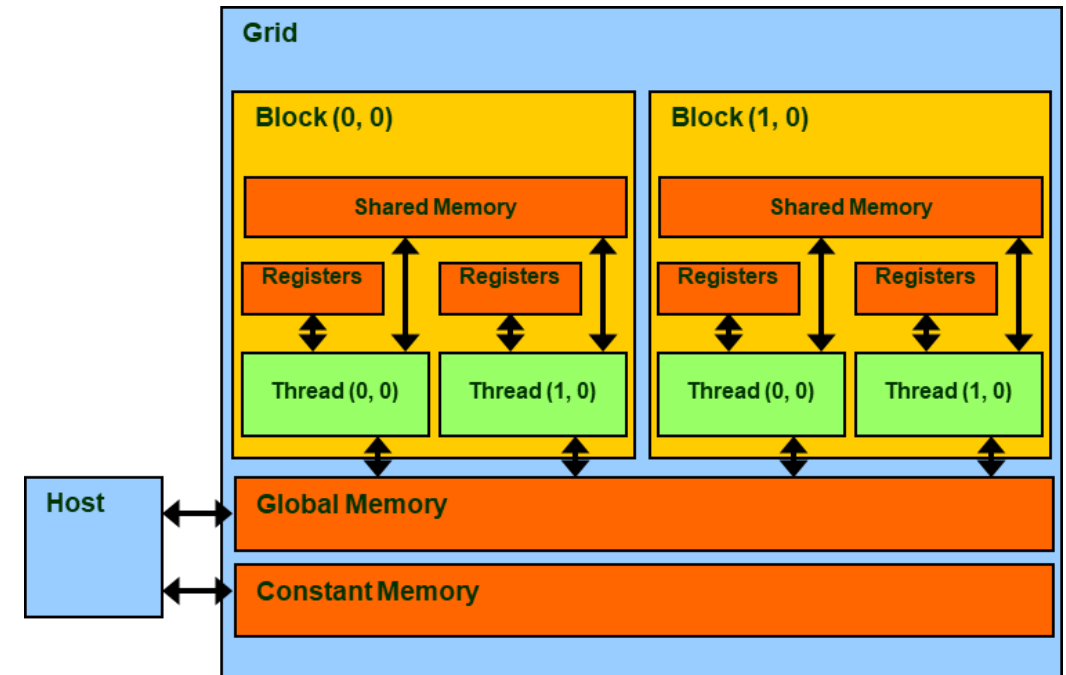
# Limitations to computing power

- A lot of access to memory
  - For each element computed we need  $2N$  global memory access
  - For each element computed we need  $2N$  operations ( $N$  multiplications and  $N$  sums)
  - The compute-to-global-memory-access is  $1:1=1$
- In the GTX650 the memory bandwidth is 7GB/s
  - Assume  $100 \times 100$  matrices
  - How many operands per seconds we can load?  $7\text{GB}/(2N \cdot 4\text{bytes}) = 8.75 \text{ Moperands/s}$
  - Being the computer-to-global-memory-access limited to 1 this means that the computing throughput is 8.75 MFlops
  - Very far from the 800 Gflops of the board!



# Shared Memory

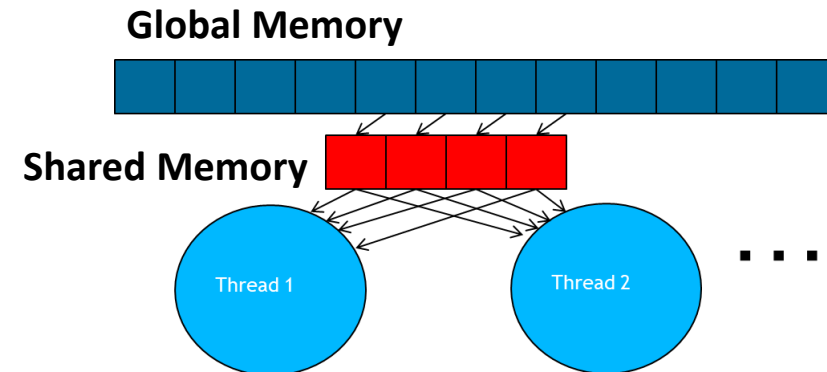
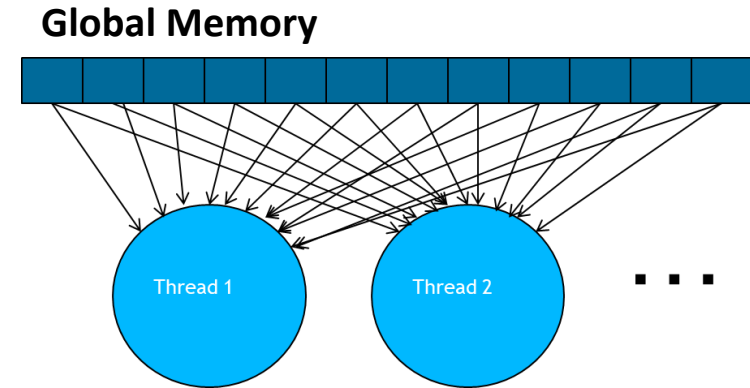
- A special type of memory whose contents are explicitly defined and used in the kernel source code
  - One in each SM
  - Accessed at much higher speed (in both latency and throughput) than global memory
  - Scope of access and sharing - thread blocks
  - Lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution
  - Accessed by memory load/store instructions
  - A form of scratchpad memory in computer architecture



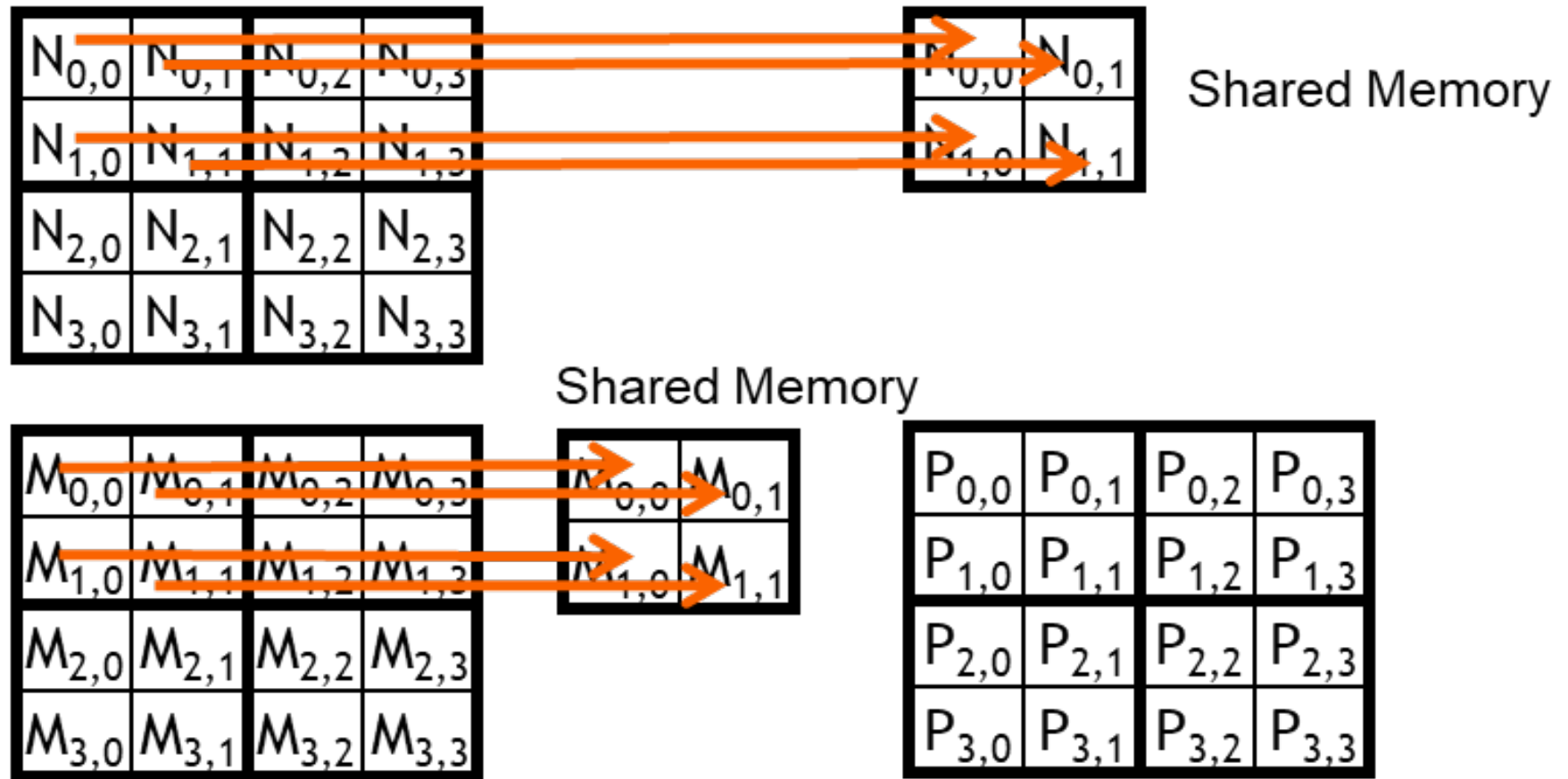
Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
__device__ __shared__ int SharedVar;	shared	block	block
__device__ int GlobalVar;	global	grid	application
__device__ __constant__ int ConstantVar;	constant	grid	application

# Shared memory for Matrix Multiplication

- Identify a “tile” of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Use barrier synchronization to make sure that all threads are ready to start the phase
- Have the multiple threads to access their data from the on-chip memory
- Use barrier synchronization to make sure that all threads have completed the current phase
- Move on to the next tile



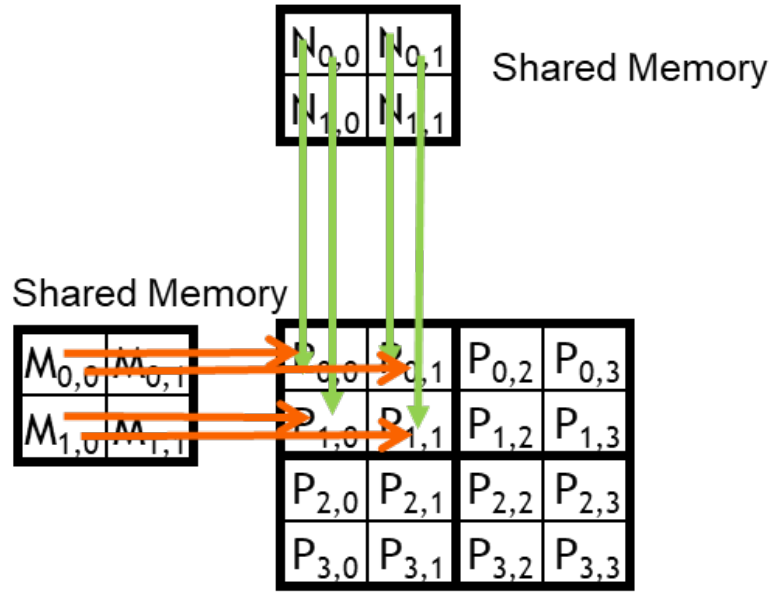
# Shared memory: phase 0 load for block (0,0)



# Shared memory: phase 0 use block (0,0)

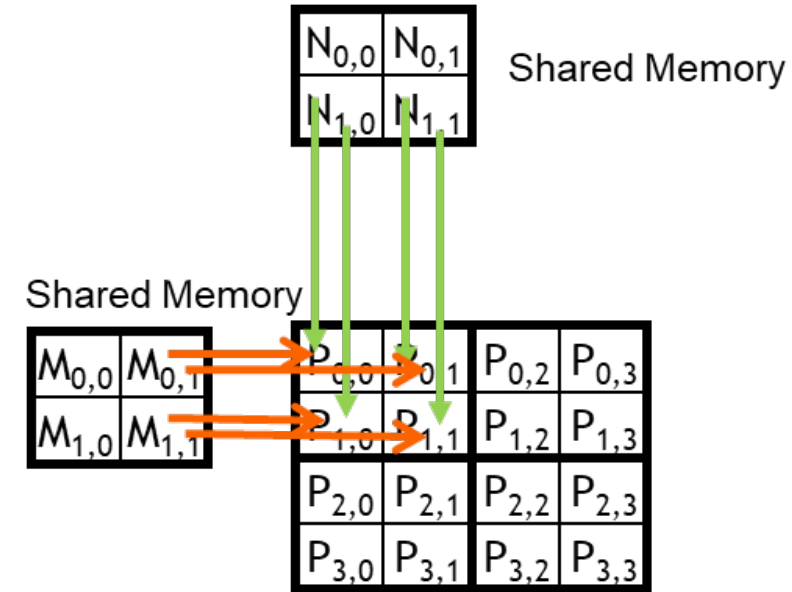
$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$




$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



# Execution phases

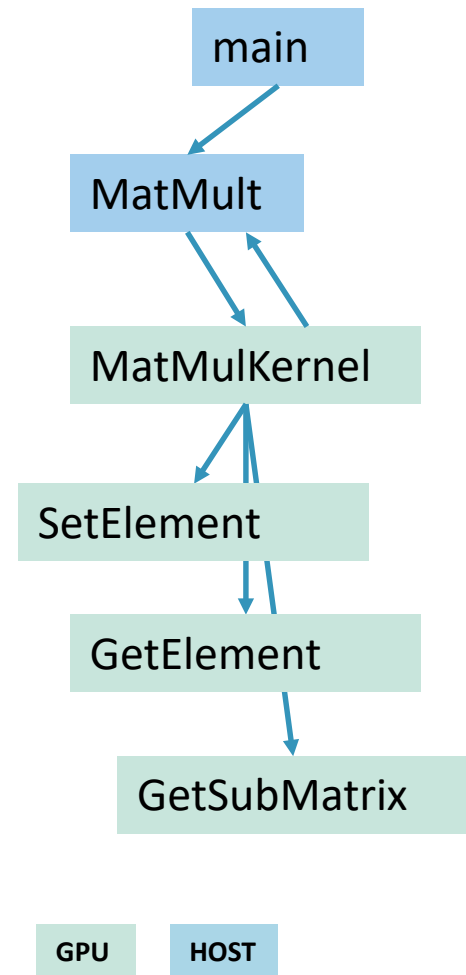
	Phase 0			Phase 1		
thread <sub>0,0</sub>	<b>M<sub>0,0</sub></b> ↓ Mds <sub>0,0</sub>	<b>N<sub>0,0</sub></b> ↓ Nds <sub>0,0</sub>	PValue <sub>0,0</sub> += Mds <sub>0,0</sub> *Nds <sub>0,0</sub> + Mds <sub>0,1</sub> *Nds <sub>1,0</sub>	<b>M<sub>0,2</sub></b> ↓ Mds <sub>0,0</sub>	<b>N<sub>2,0</sub></b> ↓ Nds <sub>0,0</sub>	PValue <sub>0,0</sub> += Mds <sub>0,0</sub> *Nds <sub>0,0</sub> + Mds <sub>0,1</sub> *Nds <sub>1,0</sub>
thread <sub>0,1</sub>	<b>M<sub>0,1</sub></b> ↓ Mds <sub>0,1</sub>	<b>N<sub>0,1</sub></b> ↓ Nds <sub>1,0</sub>	PValue <sub>0,1</sub> += Mds <sub>0,0</sub> *Nds <sub>0,1</sub> + Mds <sub>0,1</sub> *Nds <sub>1,1</sub>	<b>M<sub>0,3</sub></b> ↓ Mds <sub>0,1</sub>	<b>N<sub>2,1</sub></b> ↓ Nds <sub>0,1</sub>	PValue <sub>0,1</sub> += Mds <sub>0,0</sub> *Nds <sub>0,1</sub> + Mds <sub>0,1</sub> *Nds <sub>1,1</sub>
thread <sub>1,0</sub>	<b>M<sub>1,0</sub></b> ↓ Mds <sub>1,0</sub>	<b>N<sub>1,0</sub></b> ↓ Nds <sub>1,0</sub>	PValue <sub>1,0</sub> += Mds <sub>1,0</sub> *Nds <sub>0,0</sub> + Mds <sub>1,1</sub> *Nds <sub>1,0</sub>	<b>M<sub>1,2</sub></b> ↓ Mds <sub>1,0</sub>	<b>N<sub>3,0</sub></b> ↓ Nds <sub>1,0</sub>	PValue <sub>1,0</sub> += Mds <sub>1,0</sub> *Nds <sub>0,0</sub> + Mds <sub>1,1</sub> *Nds <sub>1,0</sub>
thread <sub>1,1</sub>	<b>M<sub>1,1</sub></b> ↓ Mds <sub>1,1</sub>	<b>N<sub>1,1</sub></b> ↓ Nds <sub>1,1</sub>	PValue <sub>1,1</sub> += Mds <sub>1,0</sub> *Nds <sub>0,1</sub> + Mds <sub>1,1</sub> *Nds <sub>1,1</sub>	<b>M<sub>1,3</sub></b> ↓ Mds <sub>1,1</sub>	<b>N<sub>3,1</sub></b> ↓ Nds <sub>1,1</sub>	PValue <sub>1,1</sub> += Mds <sub>1,0</sub> *Nds <sub>0,1</sub> + Mds <sub>1,1</sub> *Nds <sub>1,1</sub>

time 

# Synchronization

- Synchronize all threads in a block  
→ `__syncthreads()`
- All threads in the same block must reach the `__syncthreads()` before any of the them can move on
- Best used to coordinate the phased execution tiled algorithms
  - To ensure that all elements of a tile are loaded at the beginning of a phase
  - To ensure that all elements of a tile are consumed at the end of a phase

# MatrixMultiplication code



```
#include "MatrixMultiplication_shared.h"

// Matrix multiplication - HOST CODE
// (Matrix dimensions are assumed to be multiples of BLOCK_SIZE)
void MatMul(const Matrix A, const Matrix B, Matrix C) {

    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width;
    d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaError_t err = cudaMalloc(&d_A.elements, size);
    printf("CUDA malloc A: %s\n", cudaGetErrorString(err));
    cudaMemcpy(d_A.elements, A.elements, size, cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width;
    d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    err = cudaMalloc(&d_B.elements, size);
    printf("CUDA malloc B: %s\n", cudaGetErrorString(err));
    cudaMemcpy(d_B.elements, B.elements, size, cudaMemcpyHostToDevice);
    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width;
    d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    err = cudaMalloc(&d_C.elements, size);
    printf("CUDA malloc C: %s\n", cudaGetErrorString(err));

    float time;
    cudaEvent_t start, stop;
    cudaEventCreate(&start);
    cudaEventCreate(&stop);
    //start time
    cudaEventRecord(start);
```

```
// Invoke kernel
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
err = cudaThreadSynchronize();
//stop time
cudaEventRecord(stop);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time, start, stop);

printf("Run kernel: %s\n", cudaGetErrorString(err));
//print time
printf("Time: %3.5f ms\n", time);

// Read C from device memory
err = cudaMemcpy(C.elements, d_C.elements, size, cudaMemcpyDeviceToHost);
printf("Copy C off of device: %s\n", cudaGetErrorString(err));
// Free device memory
cudaFree(d_A.elements);
cudaFree(d_B.elements);
cudaFree(d_C.elements);
} //END HOST FUNCTION

//-----
// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col) {
    return A.elements[row * A.stride + col];
}

//-----
// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col, float value) {
    A.elements[row * A.stride + col] = value;
}
```



# MatrixMultiplication code

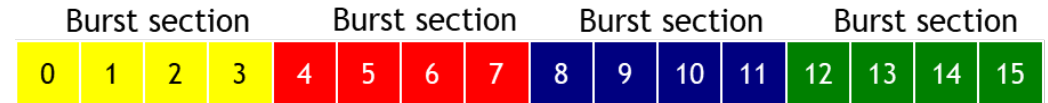
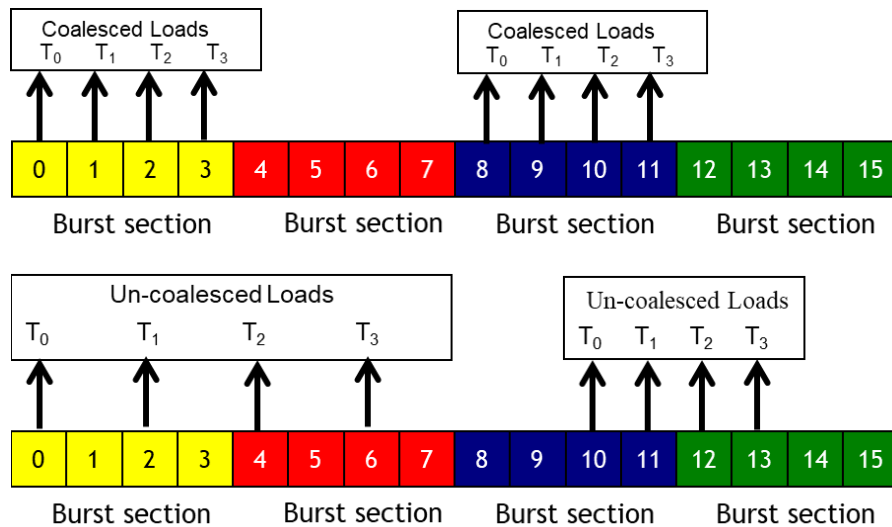
```
//-----  
// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is  
// located col sub-matrices to the right and row sub-matrices  
// down  
// from the upper-left corner of A  
__device__ Matrix GetSubMatrix(Matrix A, int row, int col) {  
    Matrix Asub;  
    Asub.width = BLOCK_SIZE;  
    Asub.height = BLOCK_SIZE;  
    Asub.stride = A.stride;  
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row +  
    BLOCK_SIZE * col];  
    return Asub;  
}  
//-----  
// Matrix multiplication kernel called by MatMul()  
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C) {  
    // Block row and column  
    int blockRow = blockIdx.y;  
    int blockCol = blockIdx.x;  
    // Each thread block computes one sub-matrix Csub of C  
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);  
    // Each thread computes one element of Csub  
    // by accumulating results into Cvalue  
    float Cvalue = 0.0;  
    // Thread row and column within Csub  
    int row = threadIdx.y;  
    int col = threadIdx.x;  
    // Loop over all the sub-matrices of A and B that are  
    // required to compute Csub  
    // Multiply each pair of sub-matrices together  
    // and accumulate the results  
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {  
        // Get sub-matrix Asub of A  
        Matrix Asub = GetSubMatrix(A, blockRow, m);
```

```
        // Get sub-matrix Bsub of B  
        Matrix Bsub = GetSubMatrix(B, m, blockCol);  
        // Shared memory used to store Asub and Bsub  
        // respectively  
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];  
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];  
        // Load Asub and Bsub from device memory to shared  
        // memory  
        // Each thread loads one element of each sub-matrix  
        As[row][col] = GetElement(Asub, row, col);  
        Bs[row][col] = GetElement(Bsub, row, col);  
        // Synchronize to make sure the sub-matrices are  
        // loaded  
        // before starting the computation  
        __syncthreads();  
        // Multiply Asub and Bsub together  
        for (int e = 0; e < BLOCK_SIZE; ++e)  
            Cvalue += As[row][e] * Bs[e][col];  
        // Synchronize to make sure that the preceding  
        // computation is done before loading two new  
        // sub-matrices of A and B in the next iteration  
        __syncthreads();  
    }  
    // Write Csub to device memory  
    // Each thread writes one element  
    SetElement(Csub, row, col, Cvalue);  
} //end kernel  
  
int main(int argc, char* argv[]){  
    Matrix A, B, C;  
    int a1, a2, b1, b2;  
    a1 = atoi(argv[1]); /* Height of A */  
    a2 = atoi(argv[2]); /* Width of A */  
    b1 = a2; /* Height of B */  
    b2 = atoi(argv[3]); /* Width of B */  
    A.height = a1;  
    A.width = a2;
```

```
    A.elements = (float*)malloc(A.width * A.height *  
    sizeof(float));  
    B.height = b1;  
    B.width = b2;  
    B.elements = (float*)malloc(B.width * B.height *  
    sizeof(float));  
    C.height = A.height;  
    C.width = B.width;  
    C.elements = (float*)malloc(C.width * C.height *  
    sizeof(float));  
    for(int i = 0; i < A.height; i++)  
        for(int j = 0; j < A.width; j++)  
            A.elements[i*A.width + j] = (random() % 3);  
    for(int i = 0; i < B.height; i++)  
        for(int j = 0; j < B.width; j++)  
            B.elements[i*B.width + j] = (random() % 2);  
    MatMul(A, B, C);  
  
    /*  
    for(int i = 0; i < min(10, A.height); i++){  
        for(int j = 0; j < min(10, A.width); j++)  
            printf("%f ", A.elements[i*A.width + j]);  
        printf("\n");  
    }  
    printf("\n");  
    for(int i = 0; i < min(10, B.height); i++){  
        for(int j = 0; j < min(10, B.width); j++)  
            printf("%f ", B.elements[i*B.width + j]);  
        printf("\n");  
    }  
    printf("\n");  
    for(int i = 0; i < min(10, C.height); i++){  
        for(int j = 0; j < min(10, C.width); j++)  
            //printf("%f ", C.elements[i*C.width + j]);  
            //printf("\n");  
        }  
    printf("\n");  
    */
```

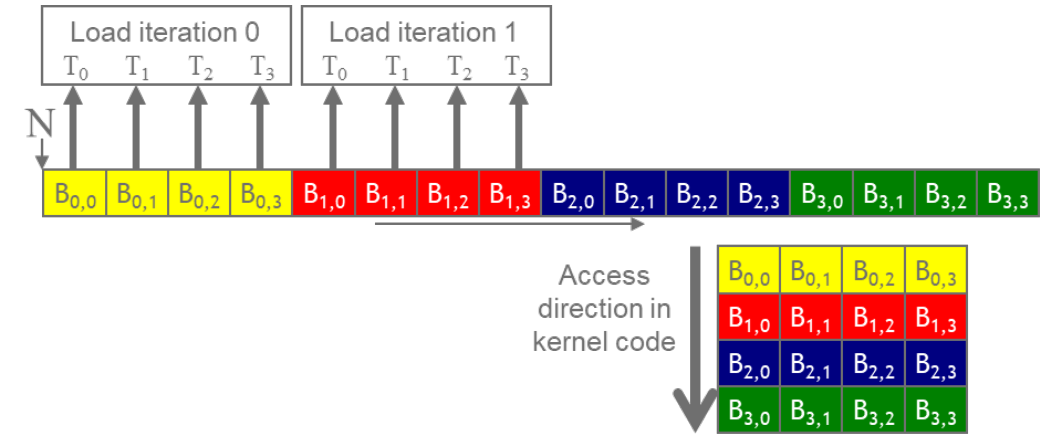
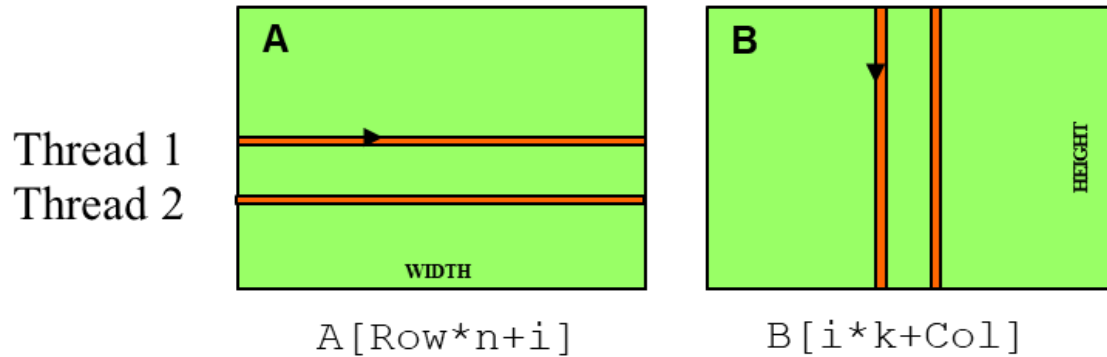
# Memory coalescing

- Each address space is partitioned into burst sections
  - Whenever a location is accessed, all other locations in the same section are also delivered to the processor
- Basic example: a 16-byte address space, 4-byte burst sections
  - In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more

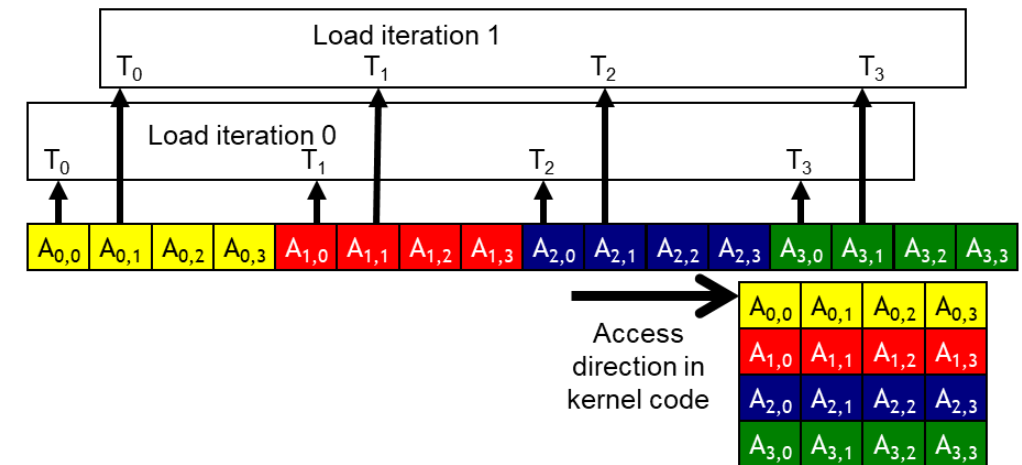


- If all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.
- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
  - Some of the bytes accessed and transferred are not used by the threads

# Memory access in Matrix Multiplication



- Access to matrix B is coalesced
- Access to matrix A isn't coalesced



# For Hands-on this afternoon

---

- <https://colab.research.google.com>  
→ Google account needed
- Based on Jupyter Notebook
- Read the introduction in the first page
- Colab allows to use machines with GPU (and also TPU)

# Bibliography

- **Parallel programming**
  - «The source book of parallel computing» (Dongarra et al.) – Morgan Kaufmann ed. (2003)
  - «Introduction to parallel programming» (Grama-Grupta-Karypsis-Kumar)
- **Parallel processing in python**
  - <https://docs.python.org/2/library/multiprocessing.html>
  - <https://docs.python.org/3/library/threading.html>
- **GPU online resources**
  - <https://docs.nvidia.com/cuda/index.html>
  - [https://docs.nvidia.com/pdf/CUDA\\_C\\_Programming\\_Guide.pdf](https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf)
  - [https://docs.nvidia.com/pdf/CUDA\\_C\\_Best\\_Practices\\_Guide.pdf](https://docs.nvidia.com/pdf/CUDA_C_Best_Practices_Guide.pdf)
  - <https://hgpu.org/> (collection of tutorials and articles)
  - «GPU Computing and Applications» (Yiju Cai) – Springer (Free from library <https://onesearch.unipi.it>)
  - “Multicore and gpu programming : an integrated approach” (Barlas) – Morgan Kaufmann (Free from library <https://onesearch.unipi.it>)
- **GPU in Python**
  - <https://wish4book.com/education/5410-hands-on-gpu-computing-with-python.html> (free book)
  - <https://documen.tician.de/pycuda/>
- **GPU books**
  - «Cuda by Example» (Sanders) – Addison-Wesley
  - «Professional Cuda C programming» (Cheng et al.) - Wrox

# Appendix

---

# Purpose of this addendum

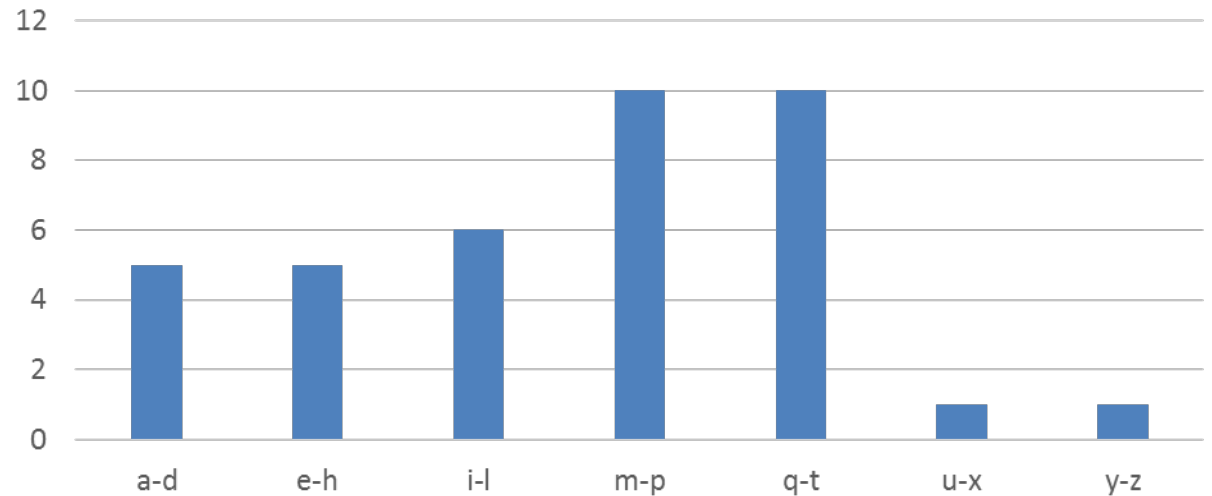
- The purpose of this addendum is to present some classic problems that can be addressed with GPU
- The kernels proposed are deliberately incompleted (and sometimes also wrong)
- As an assignment try to complete one or more of the proposed solution to have a working kernel
- Then try to include this kernel in a python code by using pyCUDA
- Try to measure the timing of your implementation with respect to some simpler implementantion or with respect to the serial version to show the speedup introduced by your work
- If you want to discuss send me e-mail: [gianluca.lamanna@unipi.it](mailto:gianluca.lamanna@unipi.it)

# Histograms

- Building histograms is a method for extracting features and patterns from large data sets
  - Standard decryption methods
  - Feature extraction for object recognition in images
  - Search for decays in High Energy Physics
  - Correlating heavenly object movements in astrophysics
- for each element in the data set, use the value to identify a “bin counter” to increment

- Example:

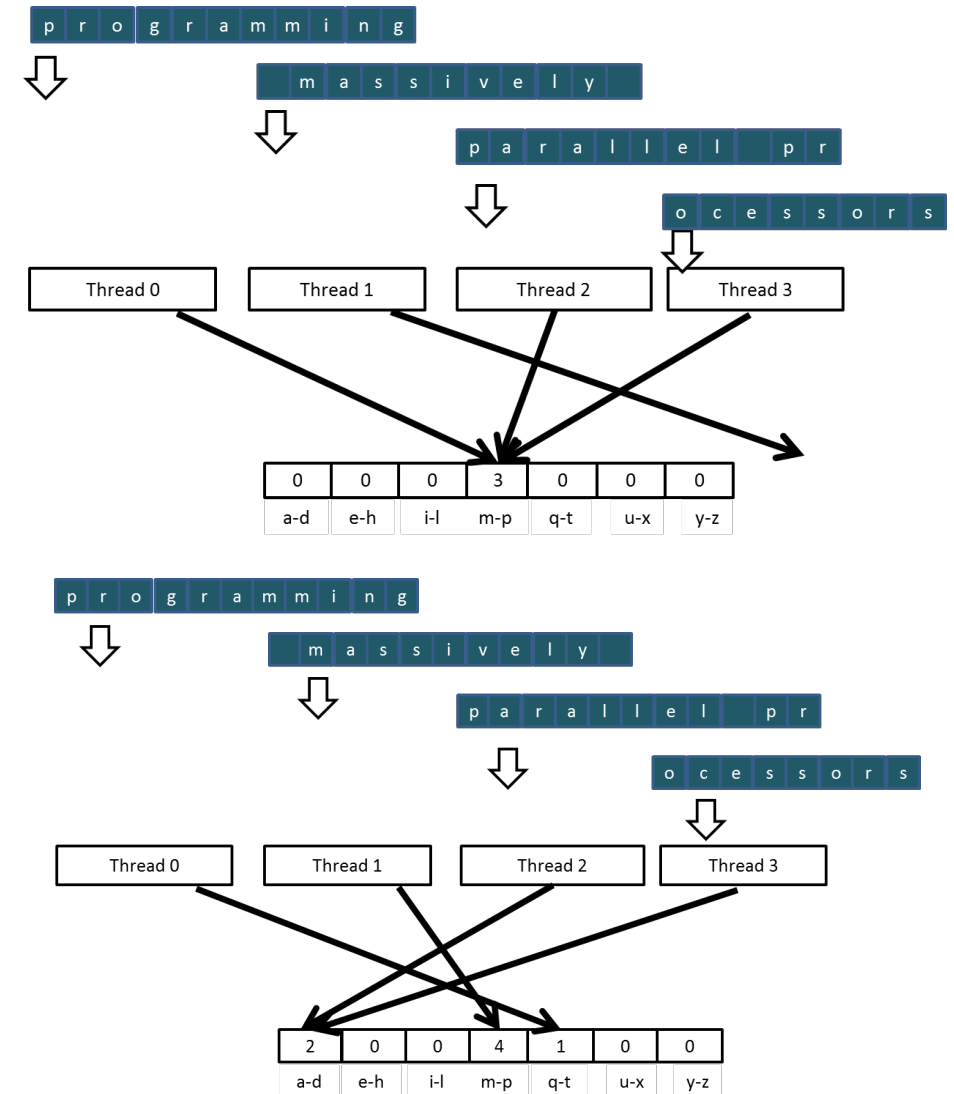
- Define the bins as four-letter sections of the alphabet: a-d, e-h, i-l, n-p, ...
- For each character in an input string, increment the appropriate bin counter.
- In the phrase “Programming Massively Parallel Processors” the output histogram is shown below:





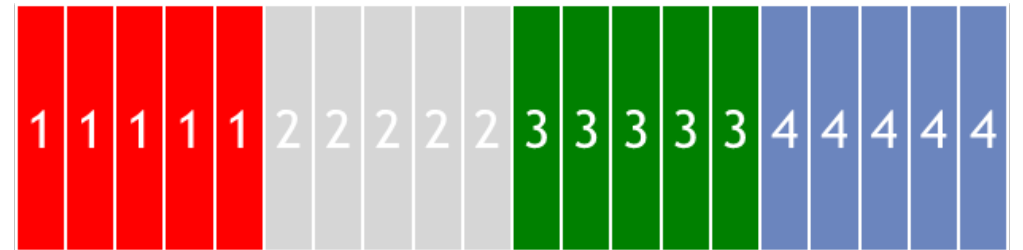
# Simple Histograms algorithm

- Partition the input into sections
- Have each thread to take a section of the input
- Each thread iterates through its section.
- For each letter, increment the appropriate bin counter



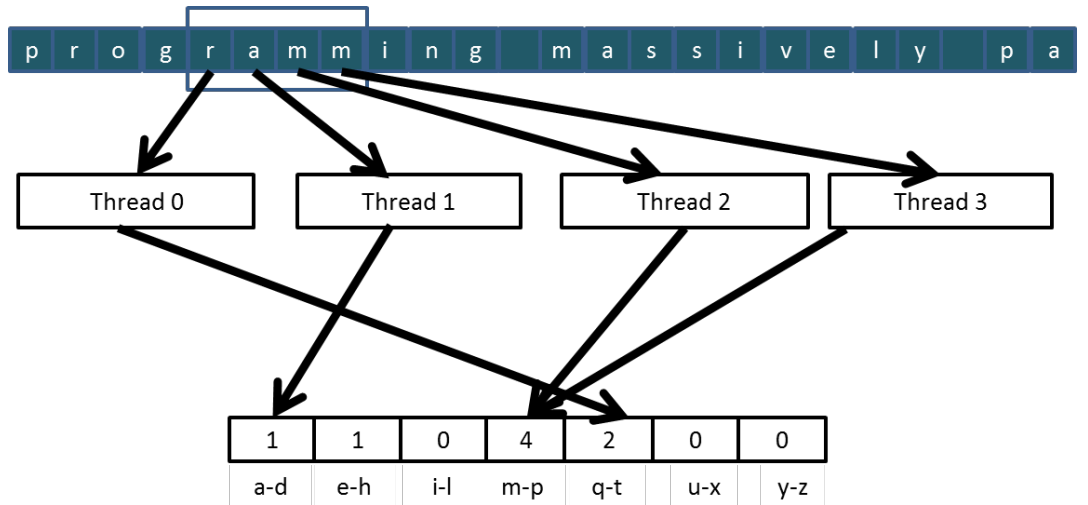
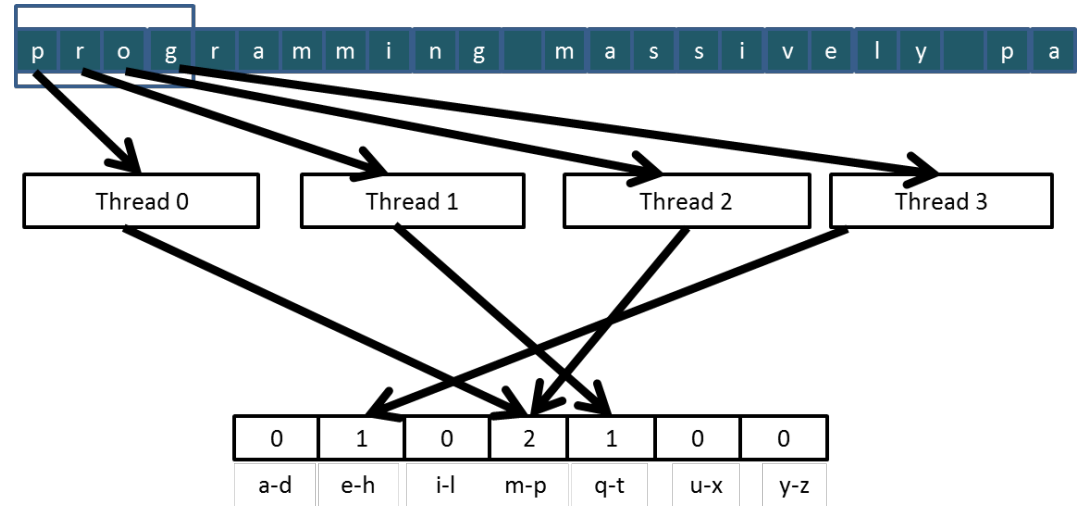
# Limits to «partitioning»

- Sectioned partitioning results in poor memory access efficiency
  - Adjacent threads do not access adjacent memory locations
  - Accesses are not coalesced
  - DRAM bandwidth is poorly utilized
- Change to interleaved partitioning
  - All threads process a contiguous section of elements
  - They all move to the next section and repeat
  - The memory accesses are coalesced



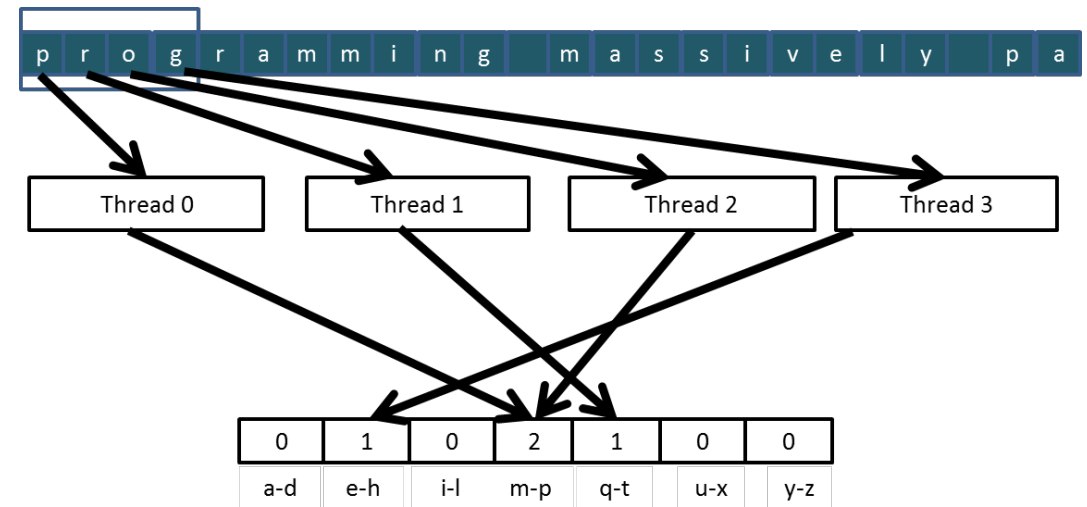
# Interleaved input access

- In this case the logic partitioning is a little bit more complicated (just a little) but the access to the memory is very efficient



# Data races

- Interleaving solve the problem of access to the memory in input
- Writing the histogram is still problematic
  - Different threads write at the same time in the same location in output memory space
- We have already see the problem of data races in the multiprocessing in python
  - Use the locks (both in multiprocessing and in multithreading)
- In GPU programming it is not «natural» to use semaphores
  - The code is essentially SIMD



# Atomic Operations

- A read-modify-write operation performed by a single hardware instruction on a memory location address
  - Read the old value, calculate a new value, and write the new value to the location
  - The hardware ensures that no other threads can perform another read-modify-write operation on the same location until the current atomic operation is complete
- Any other threads that attempt to perform an atomic operation on the same location will typically be held in a queue
  - All threads perform their atomic operations serially on the same location
- Performed by calling functions that are translated into single instructions (a.k.a. intrinsic functions or intrinsics)
  - Atomic add, sub, inc, dec, min, max, exch (exchange), CAS (compare and swap)
  - Read CUDA C programming Guide 4.0 or later for details
- Atomic Add
  - `int atomicAdd(int* address, int val);`
  - reads the 32-bit word old from the location pointed to by address in global or shared memory, computes (old + val), and stores the result back to memory at the same address. The function returns old+val.

# Histograms kernel

- Assignment:

- Follow the example:

- Notice the stride and the use of `atomicAdd()`

- Write the host code in Python and include the kernel by using `SourceModule` of `pyCUDA`.

- Define the «histogram» with the correct dimensions

- Plot the histogram with `matplotlib`

```
__global__ void histo_kernel(unsigned char *buffer,
                             long size, unsigned int *histo)
{
    int i = threadIdx.x + blockIdx.x * blockDim.x;

    // stride is total number of threads
    int stride = blockDim.x * gridDim.x;

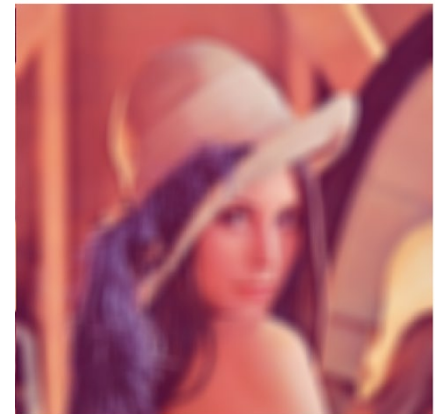
    // All threads handle blockDim.x * gridDim.x
    // consecutive elements
    while (i < size) {
        int alphabet_position = buffer[i] - "a";
        if (alphabet_position >= 0 && alphabet_position < 26)
            atomicAdd(&histo[alphabet_position/4], 1);
        i += stride;
    }
}
```

# Convolution

- The convolution is widely used in several field
  - Audio, image processing
- Base of «stencil computing»
- Convolution is a filter that transforms signal or pixel values into more desirable values.
  - Gaussian filters can be used to sharpen boundaries and edges of objects in images
  - Some filters smooth out the signal values so that one can see the big-picture trend

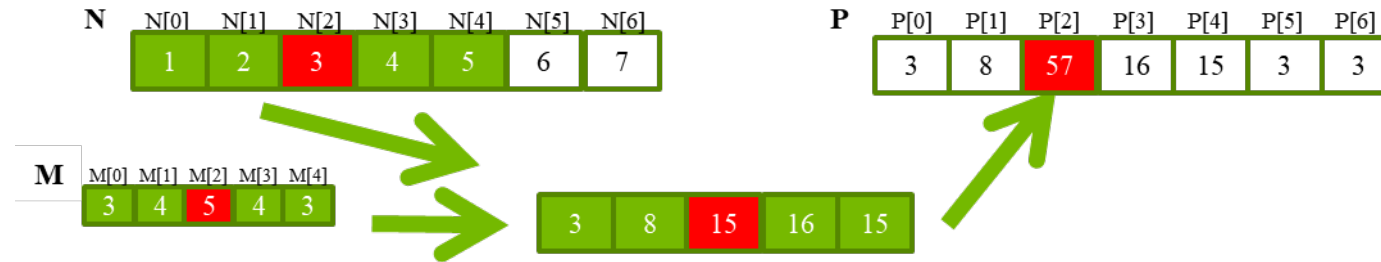
# Computational definition of Convolution

- An array operation where each output data element is a weighted sum of a collection of neighboring input elements
- The weights used in the weighted sum calculation are defined by an input mask array, we call this “convolution mask” (some time it’s called convolution kernel... but isn’t necessarily the GPU kernel)
  - The value pattern of the mask array elements defines the type of filtering done
  - Image blurring



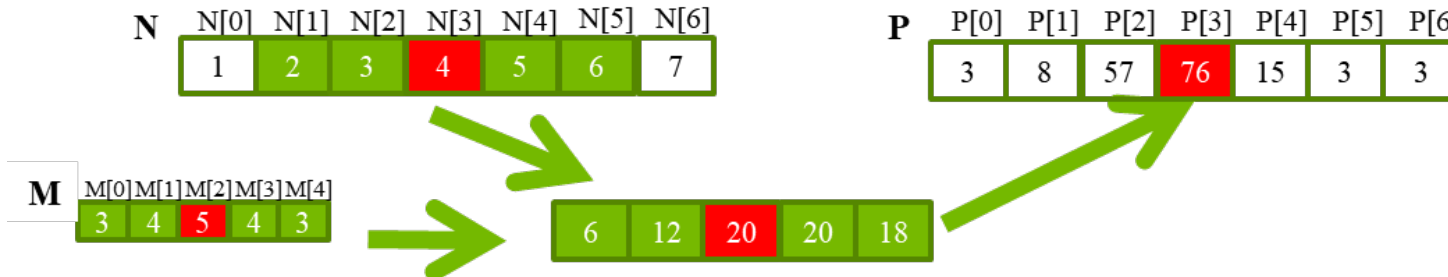


# 1D Convolution example

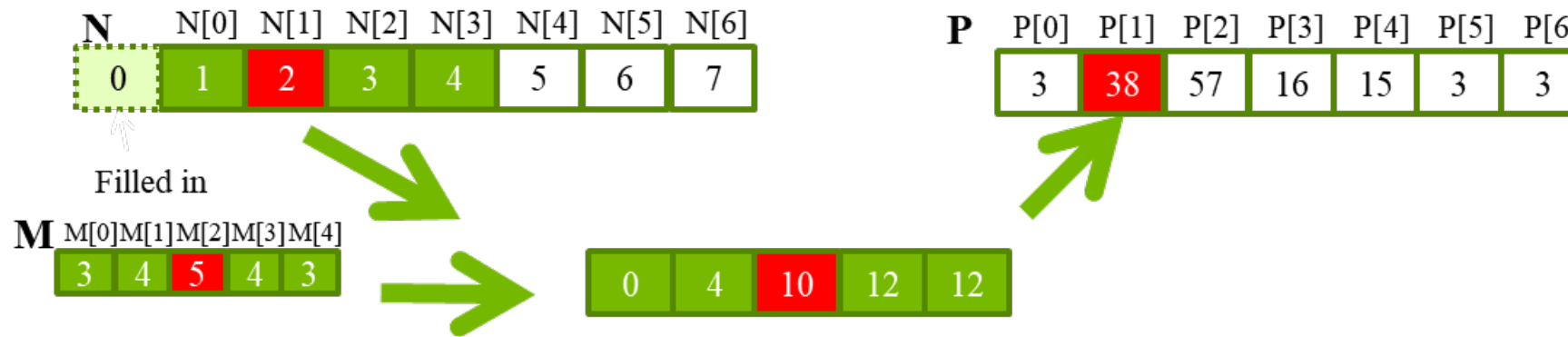


- $$P[2] = N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4]$$

- $$P[3] = N[1]*M[0] + N[2]*M[1] + N[3]*M[2] + N[4]*M[3] + N[5]*M[4]$$



# 1D convolution: boundaries



- Calculation of output elements near the boundaries (beginning and end) of the array need to deal with “ghost” elements  
→ Different policies (0, replicates of boundary values, etc.)

# A simple stencil kernel

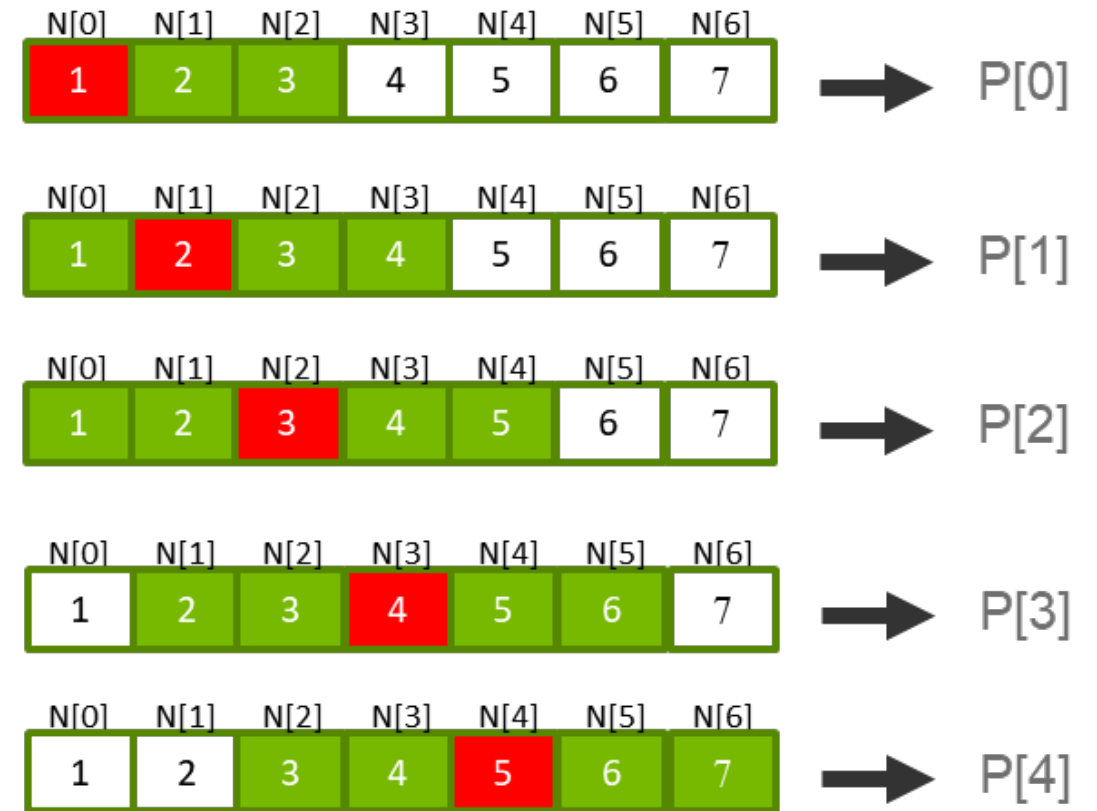
```
__global__ void convolution_1D_basic_kernel(float *N, float *M,
                                           float *P, int Mask_Width, int Width)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    float Pvalue = 0;
    int N_start_point = i - (Mask_Width/2);

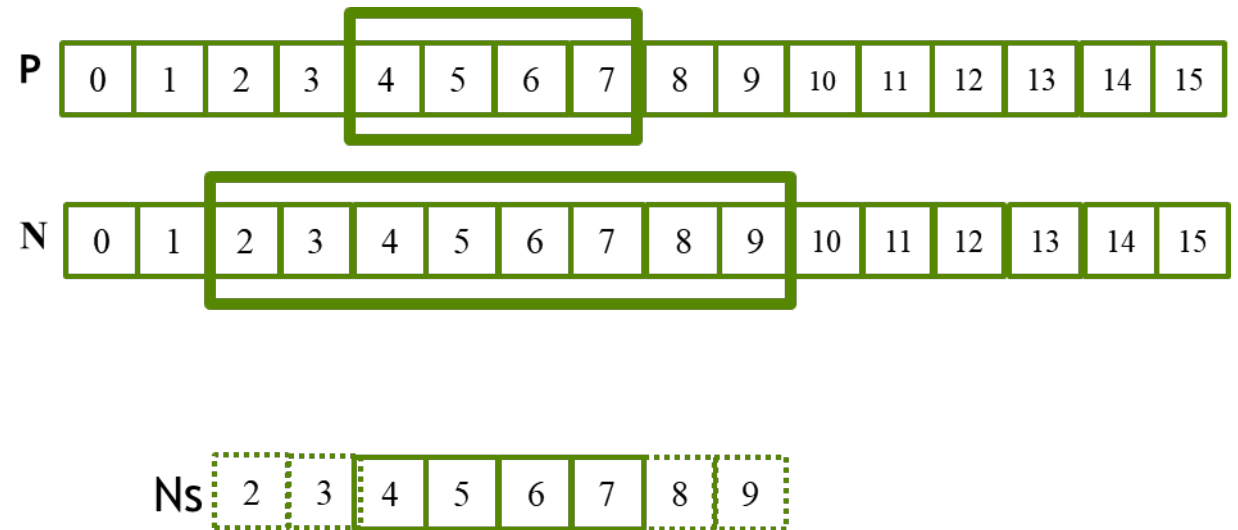
    for (int j = 0; j < Mask_Width; j++) {
        if (N_start_point + j >= 0 && N_start_point + j < Width) {
            Pvalue += N[N_start_point + j]*M[j];
        }
    }

    P[i] = Pvalue;
}
```

- Calculation of adjacent output elements involve shared input elements  
→ E.g.,  $N[2]$  is used in calculation of  $P[0]$ ,  $P[1]$ ,  $P[2]$ .  $P[3]$  and  $P[5]$  assuming a 1D convolution Mask\_Width of width 5
- We can load all the input elements required by all threads in a block into the shared memory to reduce global memory accesses



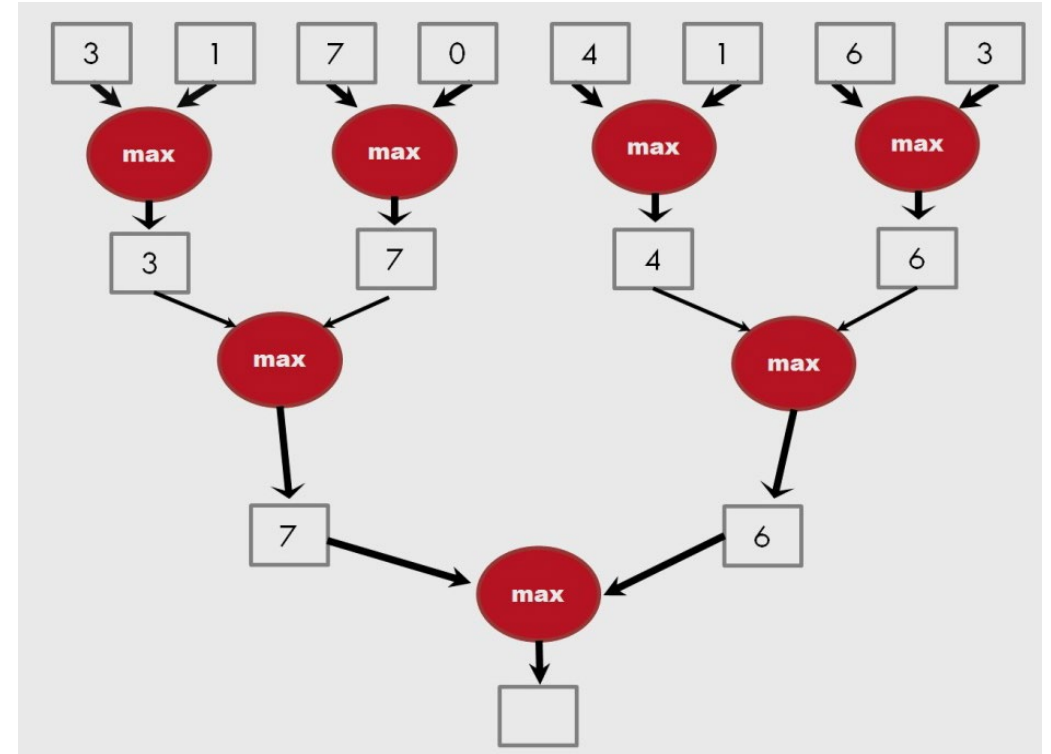
- Subdivide the input array in tiles
  - Each tile is large  $nP$  elements
  - $nP$  is the number of threads in a block (blockdim.x)
- Cache Data in shared memory
  - All the threads in a block will copy one element in the shared memory from the global memory
  - $2 \times \text{radius}$  elements (where radius is the width of the halo around the  $nP$  elements) are included



```
__global__ void stencil_1d(int *in, int *out) {
__shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
int gindex = threadIdx.x + blockIdx.x * blockDim.x;
int lindex = threadIdx.x + radius;
// Read input elements into shared memory
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
temp[lindex - RADIUS] = in[gindex - RADIUS];
temp[lindex + BLOCK_SIZE] = in[gindex +
BLOCK_SIZE];
}
// Synchronize (ensure all the data is available)
__syncthreads();
// Apply the stencil
int result = 0;
int offset = -RADIUS ; offset <= RADIUS ;
offset++)
result += temp[lindex + offset];
// Store the result
out[gindex] = result;
}
```

# Parallel reduction

- A large class of algorithm on large data set are based on the idea to reduce the amount of interesting information
  - Google and Hadoop MapReduce frameworks support this strategy
- There is no required order of processing elements in a data set (associative and commutative)
- Partition the data set into smaller chunks
  - Have each thread to process a chunk
- Use a reduction tree to summarize the results from each chunk into the final answer
- Summarize a set of input values into one value using a “reduction operation”
- Max, Min, Sum, Product



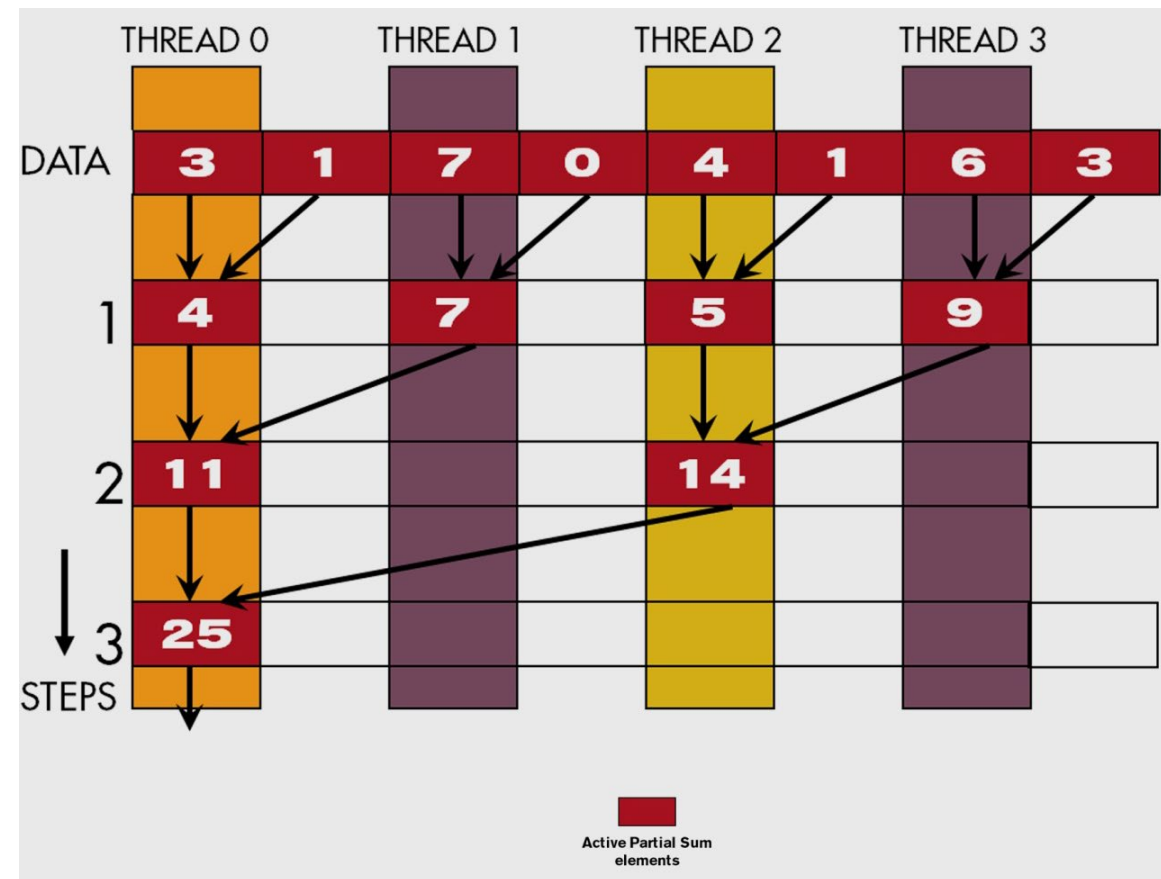
# Simple Parallel reduction tree

- Simple parallelization
  - Each operation is performed by a different task
- For  $N$  initial values the tree performs:
  - $1/2 N + 1/4 N + 1/8 N + \dots = (N-1)$  operations (if  $N=8$  we have 7 operations)
  - In  $\log(N)$  parallel steps (if  $N=8$  then steps=3)
- The average parallelization per step is  $N-1/\log(N)$ 
  - Assume  $N=1000000$ , we have in average 50000 parallel workers
  - But the peak is still 500000 parallel workers (in step 1), while the last step needs only one worker
  - Very inefficient use of parallel resources (specially if SIMD)
- It's a good idea to try to re-think the parallel structure in order to reuse efficiently the computing units.



# Parallel sum with shared memory

- Recursively halve # of threads, add two values per thread in each step
  - Takes  $\log(n)$  steps for  $n$  elements, requires  $n/2$  threads
- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is used to hold a partial sum vector
- Each step brings the partial sum vector closer to the sum
  - The final sum will be in element 0 of the partial sum vector
- Reduces global memory traffic due to partial sum values
- Thread block size limits  $n$  to be less than or equal to 2,048



# Additional assignment

---

- Try to write the GPU version of the assignment proposed in the processing in python lecture
- Measure the speedup with respect to the multithreading and multiprocessing and produce a plot to compare the results

# License

- Part of the material used for this presentation comes from the «GPU Teaching KIT» licensed by NVIDIA and University of Illinois, and has been used and modified according to Creative Commons attribution not-commercial 4.0 (<http://creativecommons.org/licenses/by-nc/4.0/legalcode>)

