



Programming model CUDA

Ioannis E. Venetis

University of Piraeus



Introduction to CUDA



CUDA

Programming model for Nvidia GPUs

Extension to the C/C++ programming languages

- New keywords

- New predefined structs

- Functions that are called from the main program (kernels)

- Pre-defined macros

General purpose programming



Host and Device

There are two distinct entities

Host

The computer system to which the GPU is attached

Device

GPU

Very important to understand

The address space for the host and the device are separate

Although in the last few GPU generations and CUDA versions it is possible to define shared memory

The data must be transferred/copied explicitly between the host and the device

Towards both directions



CUDA application execution

The source code of a CUDA application is combined

The source code for the host and the device can exist in the same source file

Computational kernel

Basic element of CUDA

A function that will execute on the device

Each thread that executes on the device will execute that kernel

There are more levels of organization as we will see

Application execution

Starts serially on the host

Calls the computational kernel

Execution is asynchronous

The CPU continues immediately executing its own code



Grid and blocks of threads

Basic element of CUDA!

A computational kernel executes on a grid of threads

The grid is composed of blocks of threads

Block of threads can be organized on 1, 2 or 3 dimensions within the grid

Each block is composed of a number of threads

Threads can be organized on 1, 2 or 3 dimensions within the grid block

The number of dimensions of the grid don't necessarily match the number of dimensions of the blocks

E.g., we can have a 1-D grid that contains 2-D blocks

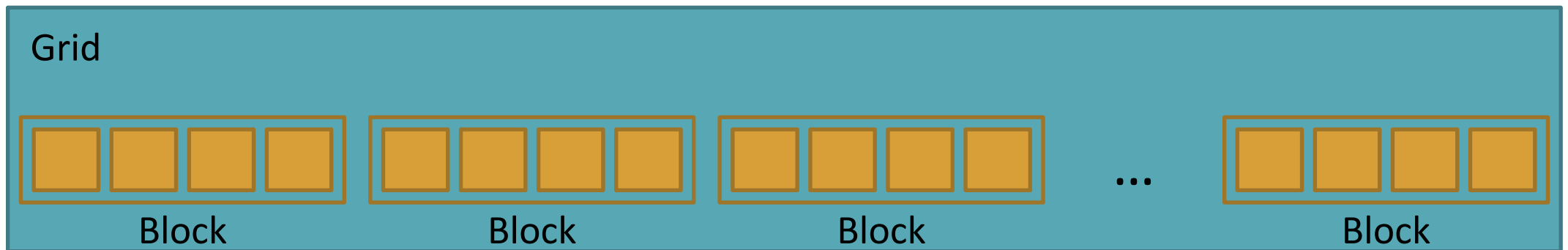
The number of blocks that compose the grid, the number of threads within each block and the organization into dimensions is defined by the programmer

All threads of the grid execute the same computational kernel

SIMT model: Single Instruction Multiple Threads

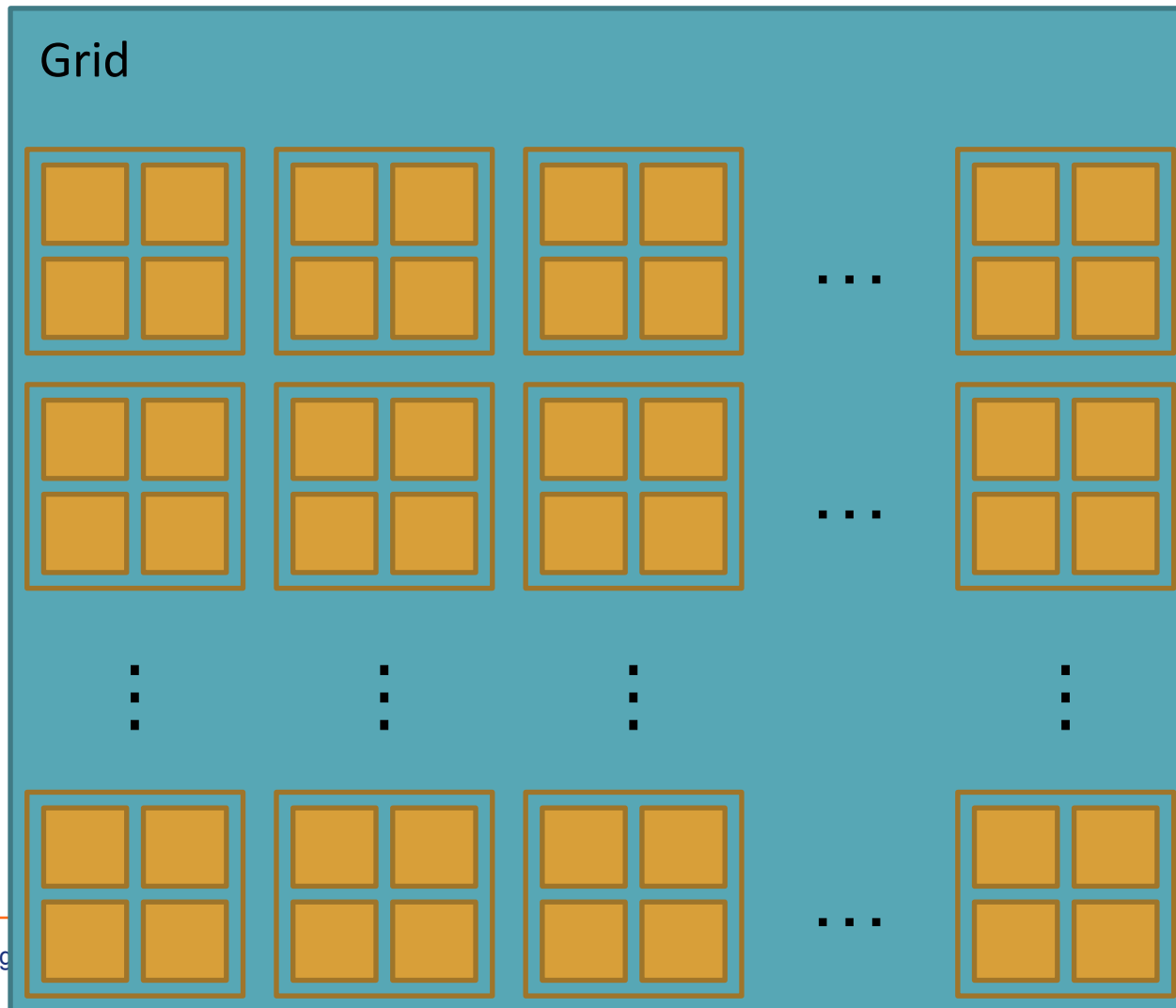
Examples of grid/blocks organization

1-D grid with 1-D blocks



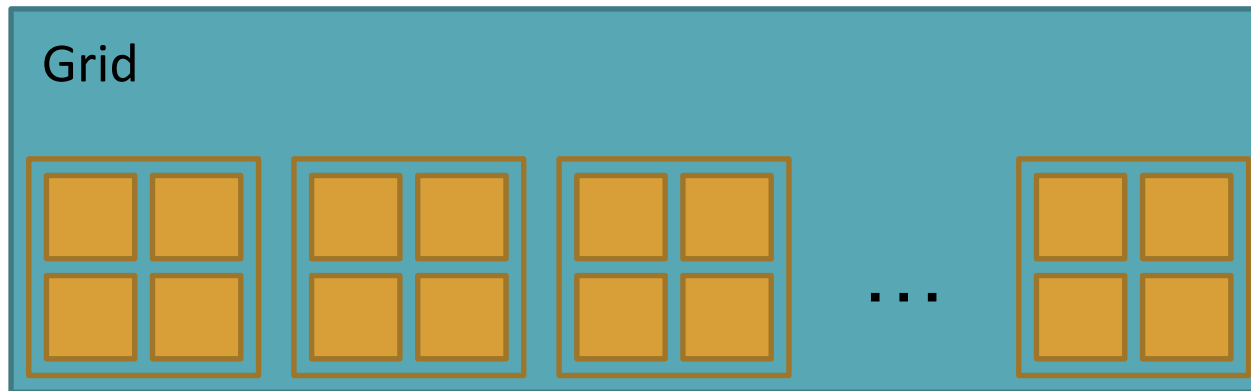
Examples of grid/blocks organization

2-D grid with 2-D blocks



Examples of grid/blocks organization

1-D grid with 2-D blocks



In the same sense we can define grids and blocks that will use the 3rd dimension too

CUDA application execution

Serial code (host)

Includes preparation for executing the computational kernel

Computational kernel (device)

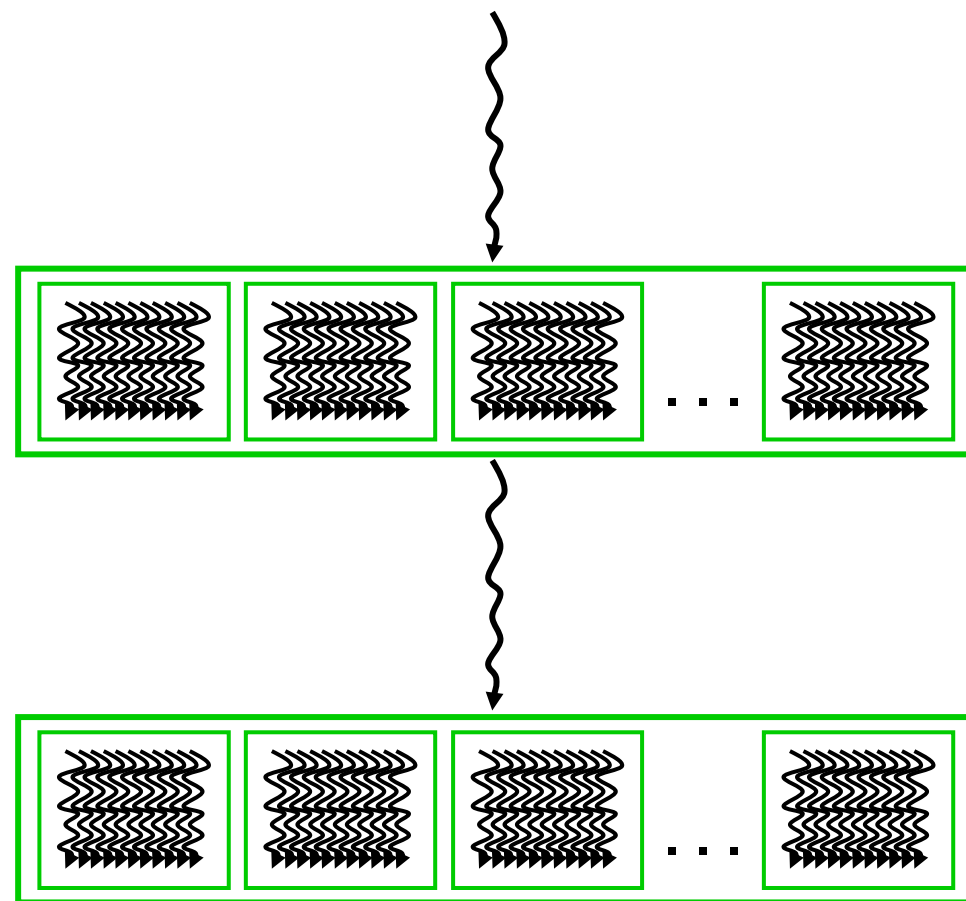
KernelA<<< nBlk, nTid >>>(args);

Serial code (host)

Includes preparation for executing the computational kernel

Computational kernel (device)

KernelB<<< nBlk, nTid >>>(args);





Restrictions

Where do they come from?

Do you remember Compute Capability?

Only 2-D grid

3-D grid also allowed

Technical specifications	Compute capability										
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	3.7	5.0	5.2	5.3
Maximum dimensionality of grid of thread blocks		2						3			
Maximum dimensionality of thread block						3					
Maximum x-dimension of a grid of thread blocks			65535						2 ³¹ - 1		
Maximum y-, or z-dimension of a grid of thread blocks						65535					
Maximum x- or y-dimension of a block			2						1024		
Maximum z-dimension of a block						64					
Maximum number of threads per block			12						1024		








Maximum allowed number of threads per block dimension

but it is not allowed to get over this limit!



Example

Assume Compute Capability $\geq 2.x$

-  Is it allowed to create a 1D block with 100 threads;
-  Is it allowed to create a 1D block with 2000 threads;
-  Is it allowed to create a 2D block with 32×32 threads;
-  Is it allowed to create a 2D block with 16×64 threads;
-  Is it allowed to create a 2D block with 32×64 threads;
-  Is it allowed to create a 3D block with $4 \times 32 \times 4$ threads;
-  Is it allowed to create a 3D block with $4 \times 32 \times 16$ threads;



Why not only threads but also blocks?

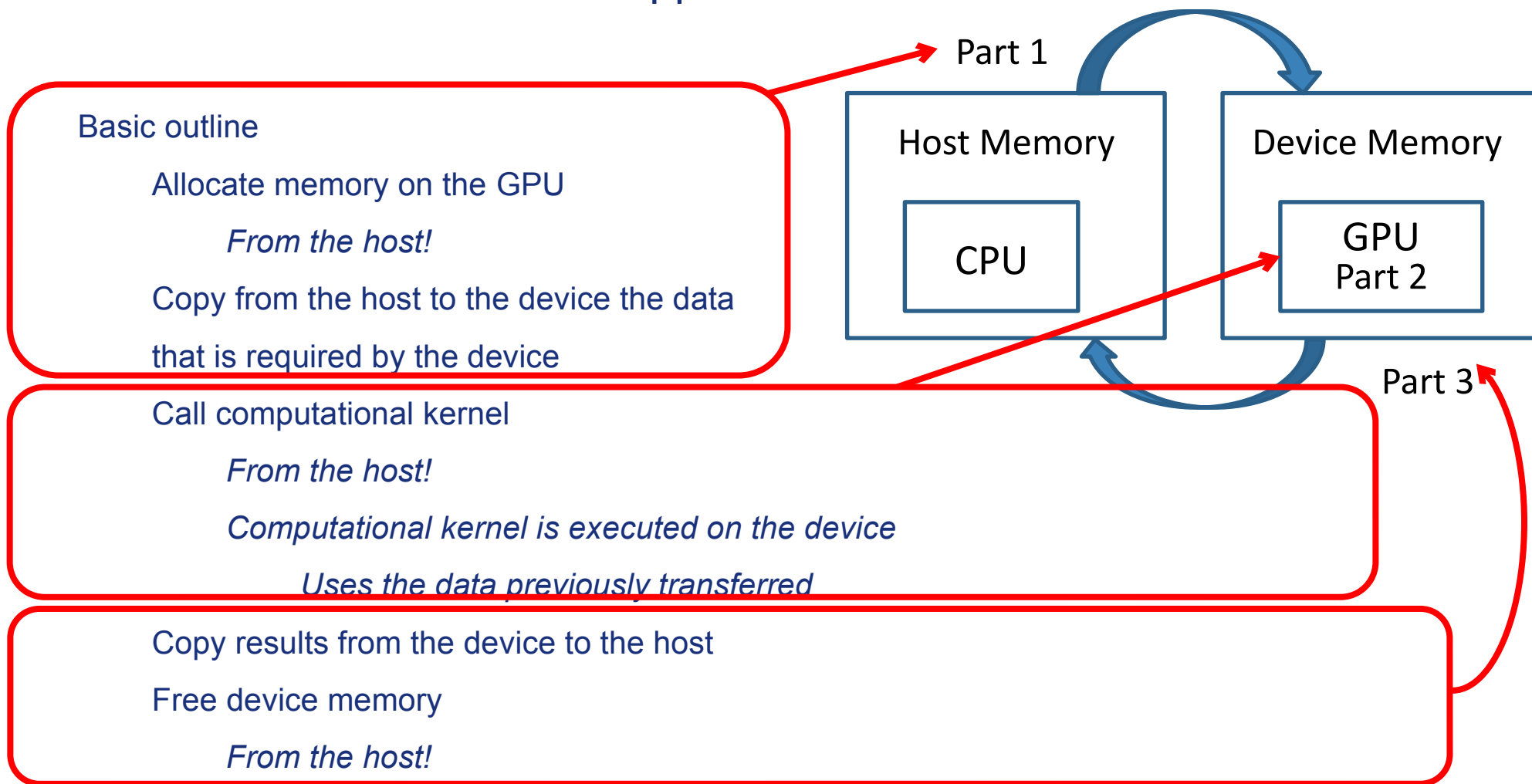
1024 (or 512) threads are too few

To map larger problems

To fully exploit the computational resources of a GPU

Technical specifications	Compute capability										
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	3.7	5.0	5.2	5.3
Maximum dimensionality of grid of thread blocks	2				3						
Maximum dimensionality of thread block	3										
Maximum x-dimension of a grid of thread blocks	65535					$2^{31} - 1$					
Maximum y-, or z-dimension of a grid of thread blocks	65535										
Maximum x- or y-dimension of a block	512				1024						
Maximum z-dimension of a block	64										
Maximum number of threads per block	512				1024						

Flow of execution of a CUDA application





Predefined variables (1/2)

Each thread executes the computational kernel

How do they differentiate among each other, so that they can decide what data each one will process?

OpenMP

Unique thread number

Acquired through `omp_get_thread_num()`

MPI

Unique rank

Acquired through `MPI_Comm_rank()`



Predefined variables (2/2)

CUDA includes a number of predefined variables

Each thread has access to them simply by referencing their name

They contain (possibly) different values for different blocks and threads

Predefined variable	Description
gridDim	Grid size (= number of blocks) on each dimension
blockIdx	Coordinates of a block within the grid
blockDim	Block size (= number of threads) on each dimension
threadIdx	Coordinates of a thread within a block

1 variable provides information for up to 3 dimensions?

They are structs that contain multiple members

The members of the struct are “x”, “y”, “z”

E.g., gridDim.x, gridDim.y, gridDim.z

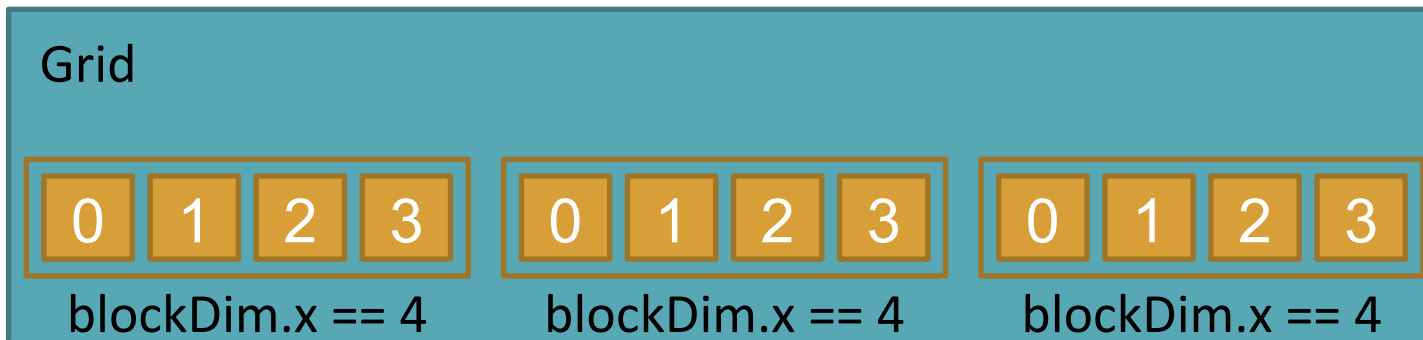
Examples of grids/blocks

1-D grid with 1-D blocks

`gridDim.x == 3`

`gridDim.y == 1`

`gridDim.z == 1`



`threadIdx.x == 0`

`threadIdx.x == 1`

`threadIdx.x == 2`

`threadIdx.x == 3`

Examples of grids/blocks

2-D grid with 2-D blocks

`gridDim.x == 4`

`gridDim.y == 3`

`gridDim.z == 1`

For all blocks

`blockDim.x == 2`

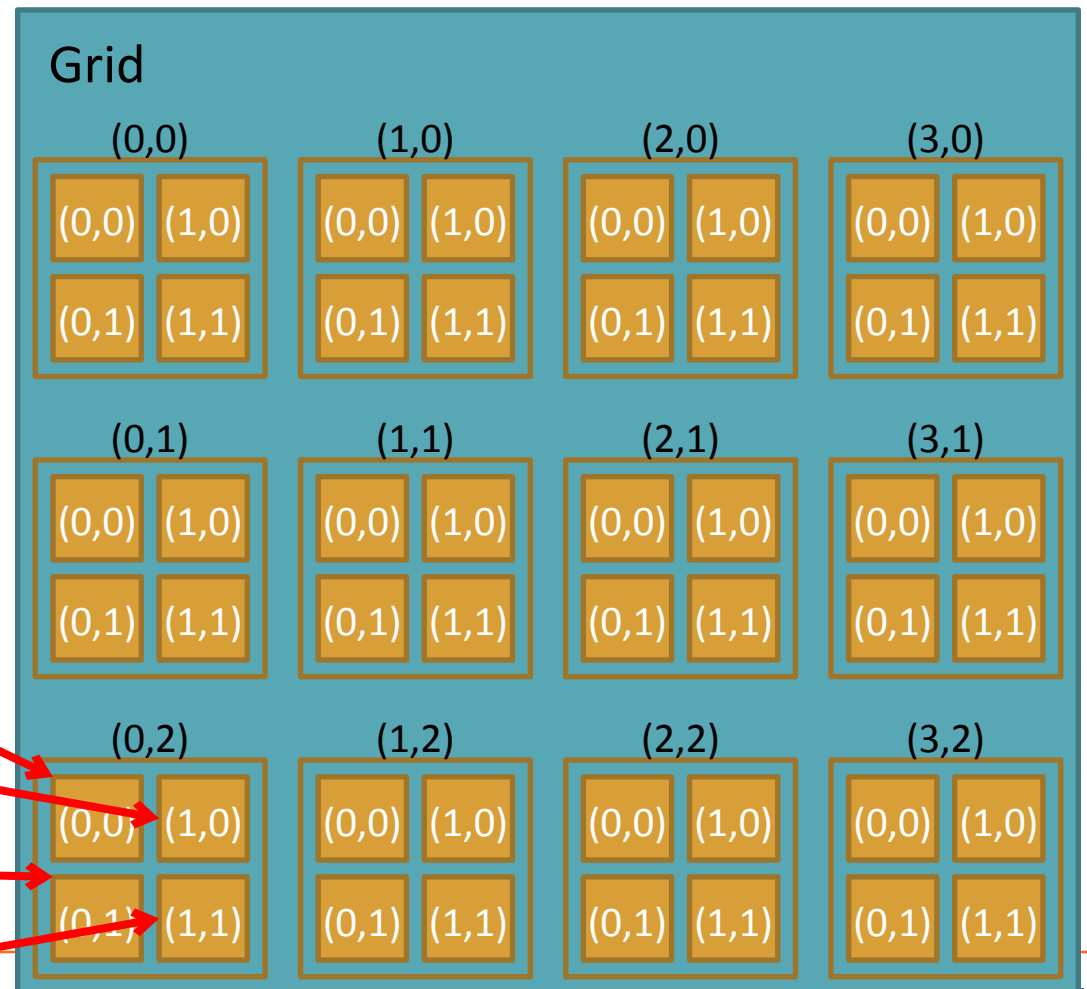
`blockDim.y == 2`

`threadIdx.x == 0`
`threadIdx.y == 0`

`threadIdx.x == 1`
`threadIdx.y == 0`

`threadIdx.x == 0`
`threadIdx.y == 1`

`threadIdx.x == 1`
`threadIdx.y == 1`





Examples of grids/blocks

1-D grid with 2-D blocks

`gridDim.x == 3`

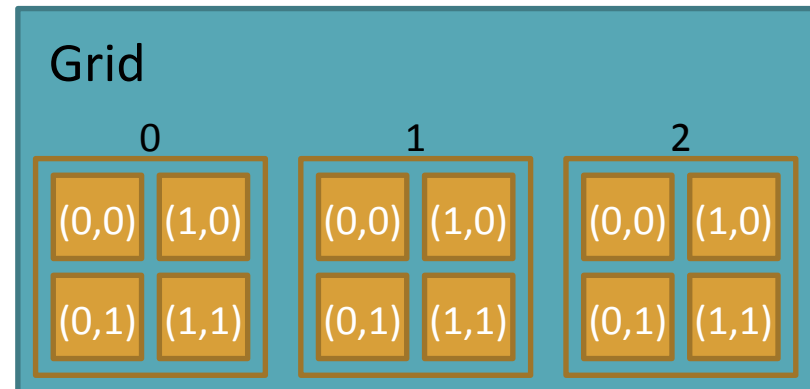
`gridDim.y == 1`

`gridDim.z == 1`

For all blocks

`blockDim.x == 2`

`blockDim.y == 2`





Example

Let's try to understand how these variables are used

Adding two vectors

1-D problem

Start with an easy example on how to use the predefined variables

Vectors with only a few elements

We will discuss what we mean with "few elements"

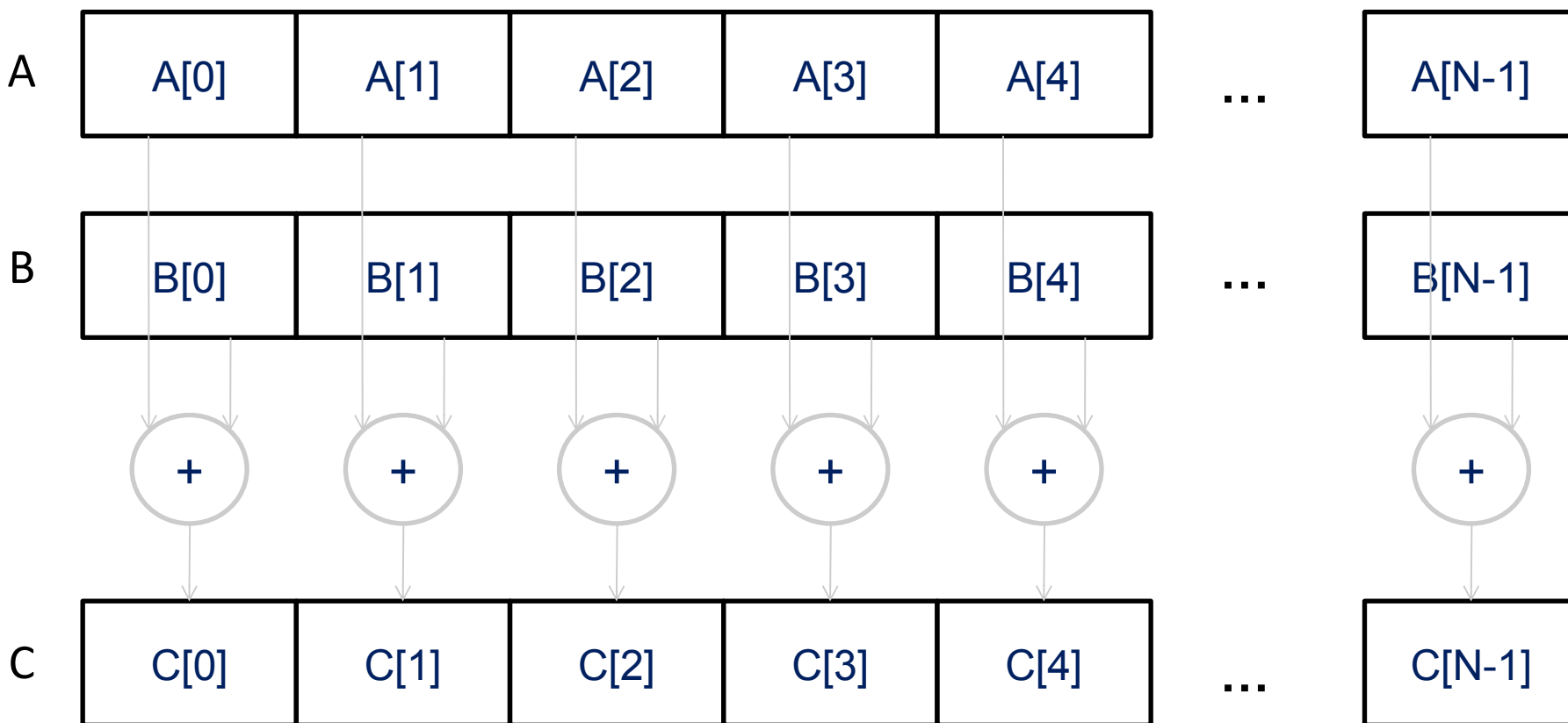
We will discuss how to expand on that and handle larger vectors

Our approach will be for each CUDA thread to add one element of vector A with one element of vector B

This is not a strict requirement in CUDA

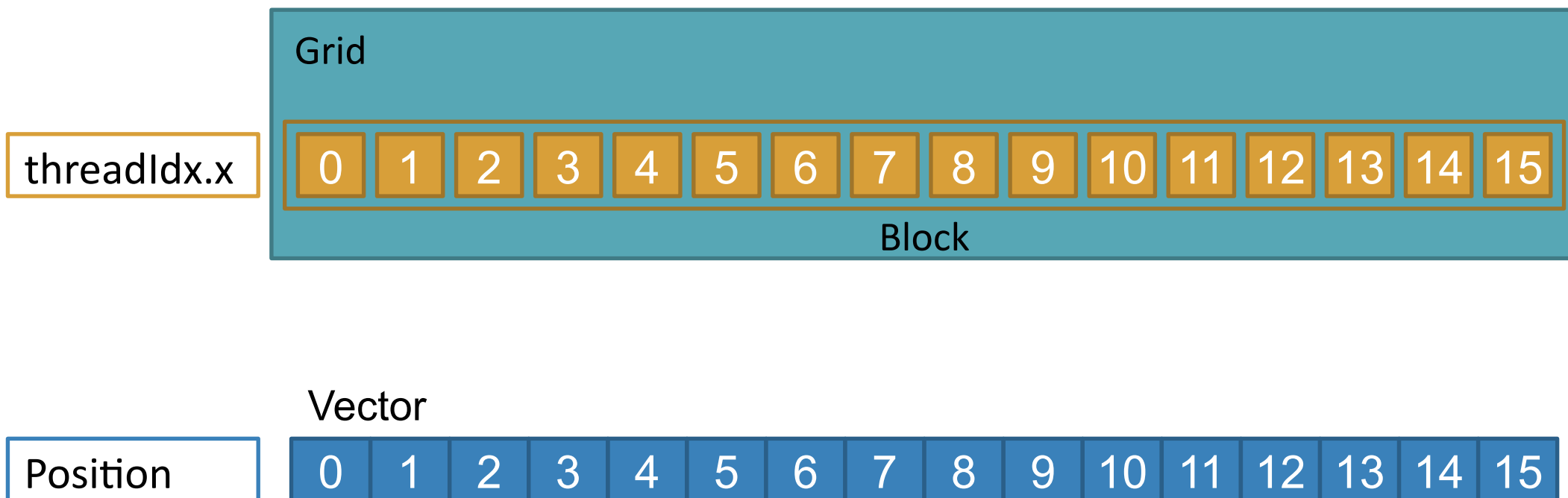
But it is the typical approach to most problems

Example





Correlating threads of a block to memory locations





Vector addition – C code (1/3)

```
/*
 * Addition of vectors A and B
 * Result is stored in vector C
 *
 * We isolate the code within a function,
 * so that only the function will require changes
 * when porting the code to CUDA.
 */
void vecAdd(float *A_d, float *B_d, float *C_d, int n)
{
    for (int i = 0; i < n; i++) {
        C_d[i] = A_d[i] + B_d[i];
    }
}
```



Vector addition – C code (2/3)

```
int main(int argc, char *argv[])
{
    float *A_h, *B_h, *C_h; // Pointers to host memory
    int N;                  // Size of vectors

    if (argc != 2) {
        printf("Provide the problem size \n");
        exit(0);
    }

    N = atoi(argv[1]);
    if (N > 1024) {
        printf("Problem size too large.\n");
        exit(0);
    }

    A_h = (float *)malloc(N * sizeof(float));
    B_h = (float *)malloc(N * sizeof(float));
    C_h = (float *)malloc(N * sizeof(float));
    if ((A_h == NULL) || (B_h == NULL) || (C_h == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```

Different between GPU generations ⇒
Should be found from within the program ⇒
See the function `cudaGetDeviceProperties()`



Vector addition – C code (3/3)

```
/*  
 * Here we should initialize vectors A and B.  
 */  
  
/*  
 * Call function to add vectors.  
 */  
vecAdd(A_h, B_h, C_h, N);  
  
printf("C = ");  
for (i = 0; i < N; i++) {  
    printf("%d ", C_h[i]);  
}  
  
return(0);  
} /* main() ends here */
```

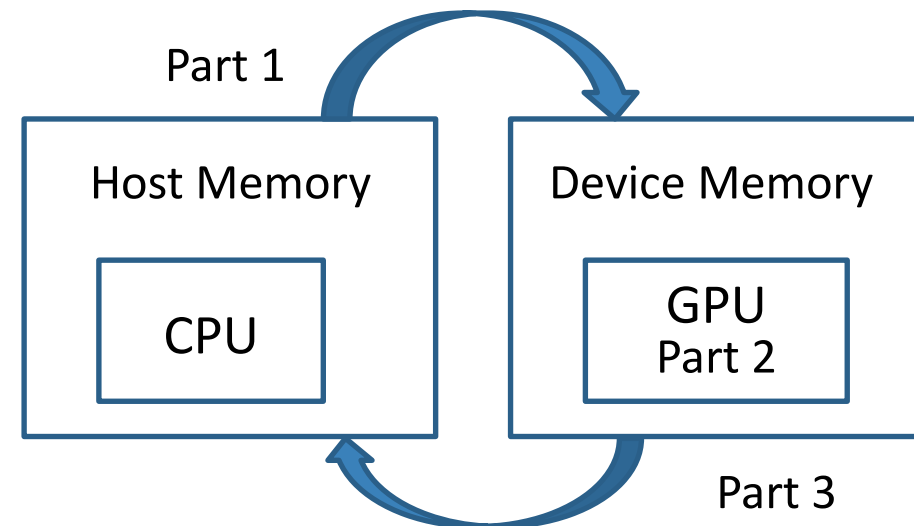
How to port the code to CUDA?

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

```
void vecAdd(float *A_h, float *B_h, float *C_h, int n)
{
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;
    ...
    1. // Allocate memory on the device for the vectors
       // Copy A and B to the device

    2. // Call computational kernel –
       // execute vector addition on the device

    3. // Copy results to the host
       // Free memory on the device
}
```



Quick (and incomplete) overview of the memory hierarchy on CUDA

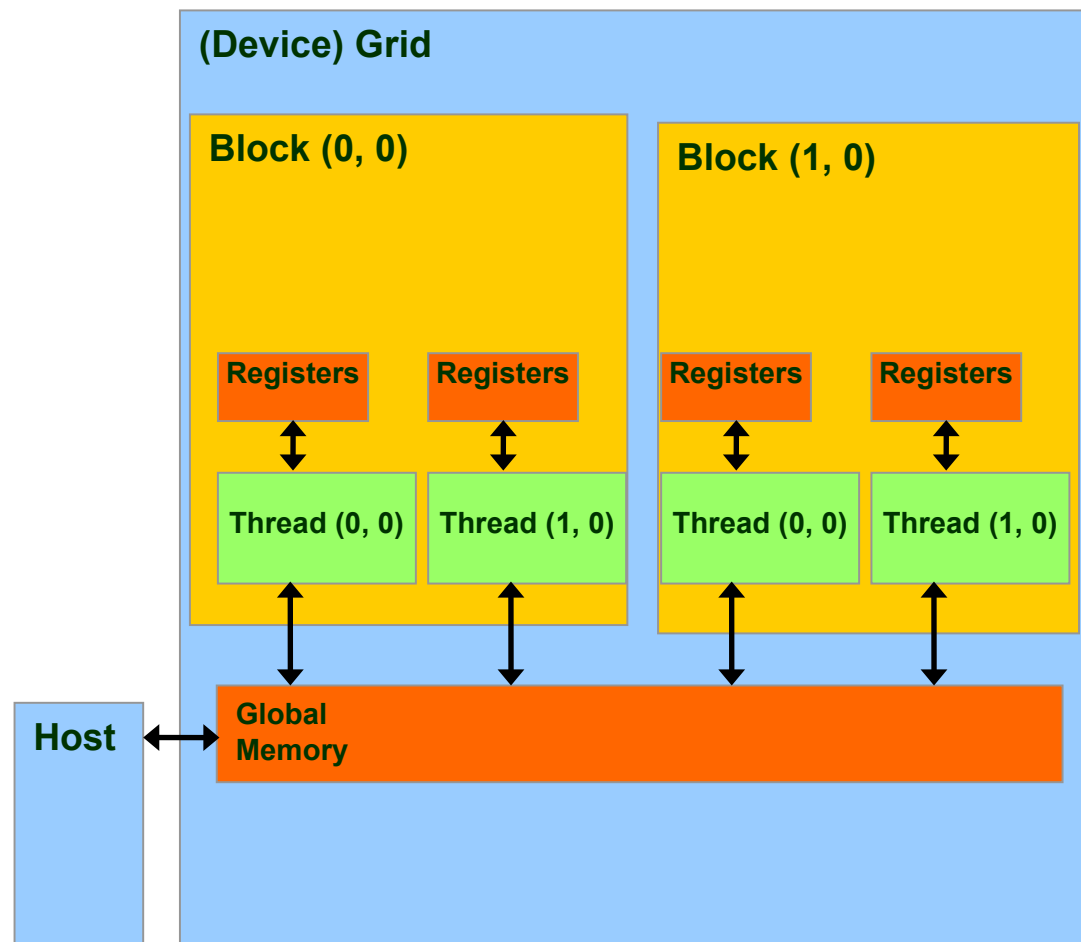
Code executing on the device:

- Reads/Writes registers per thread

- Reads/Writes global memory of the GPU per grid

Code on the host:

- Transfers data from/to global memory of the GPU per grid



CUDA API for basic memory management on the device

cudaMalloc()

Allocates an object in the global memory of the device

Two parameters

Pointer to the object to be allocated

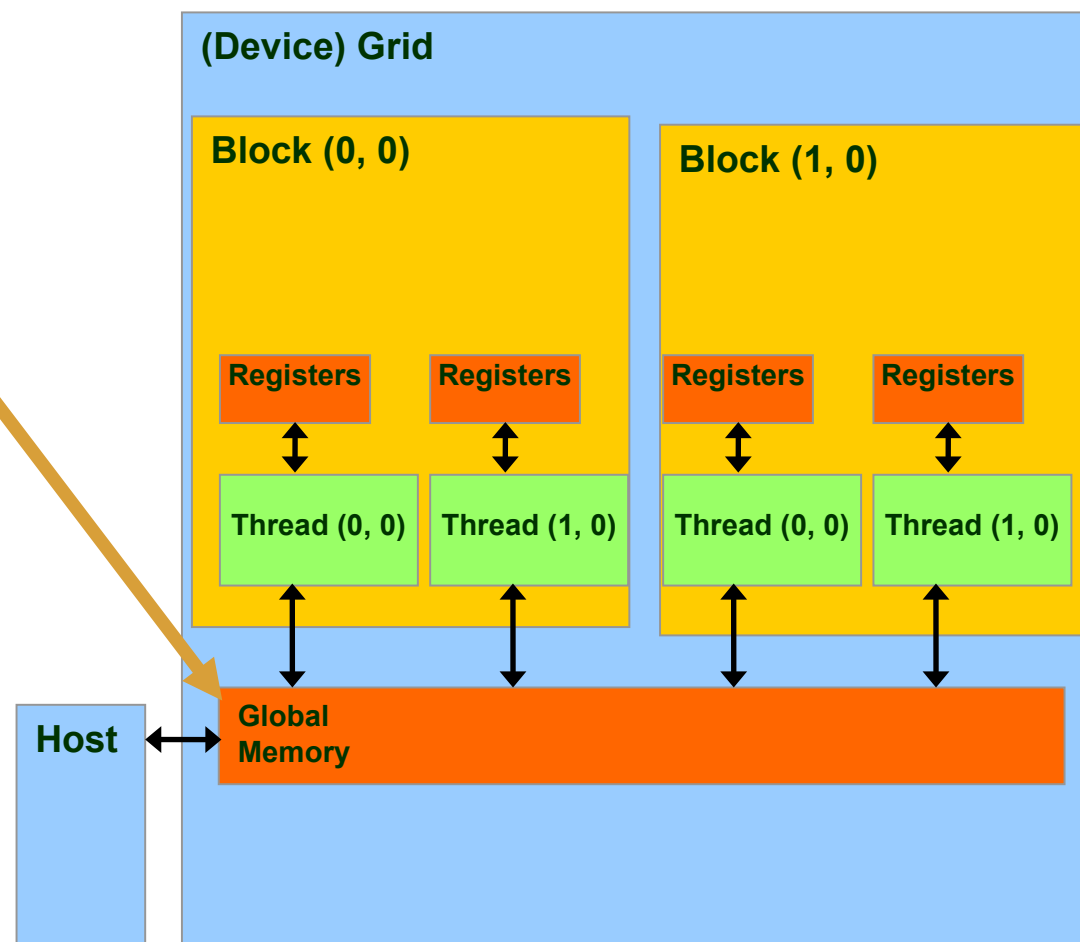
Object size in bytes

cudaFree()

Free an object from the global memory of the device

One parameter

Pointer to the object



API for data transfer

`cudaMemcpy()`

Data transfer

4 parameters

Pointer to the object-target (device)

Pointer to the object-source (host)

Bytes to transfer

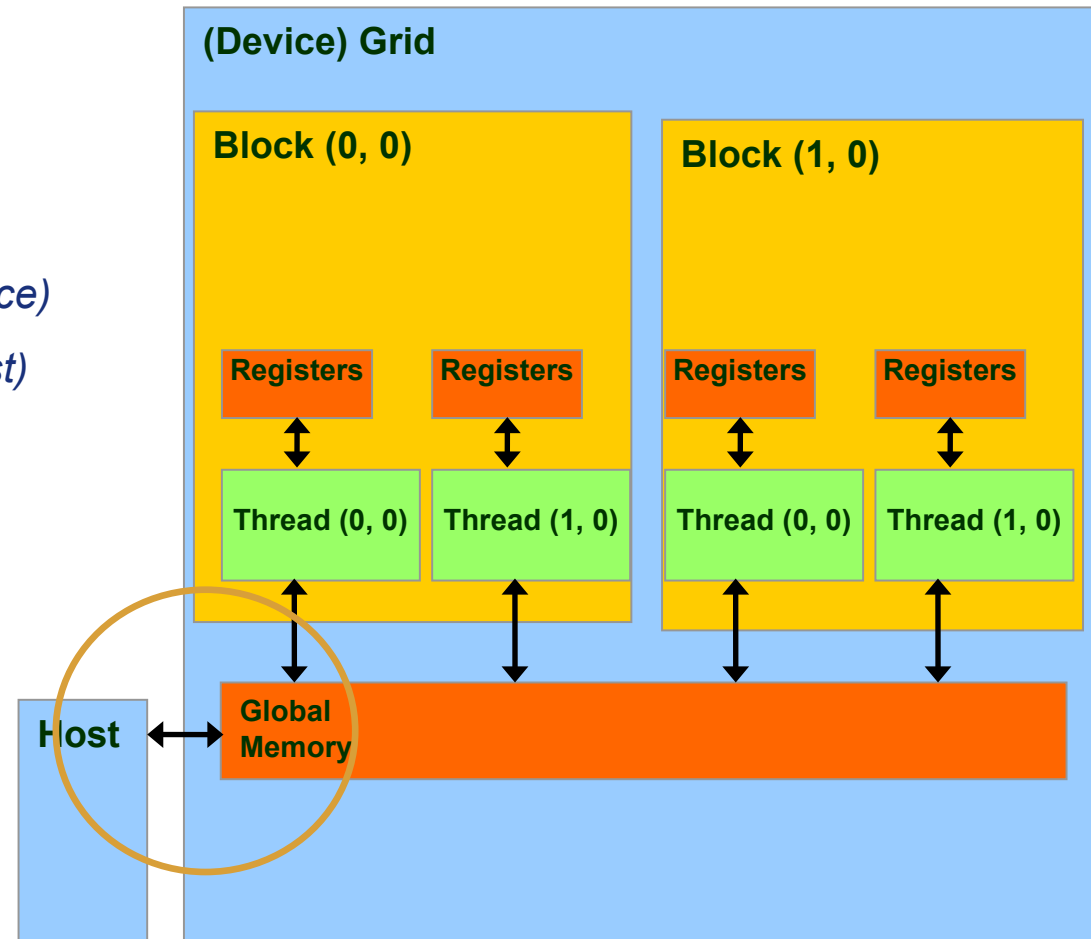
Direction of transfer

Data transfer to the device is

asynchronous

The host program continues

immediately execution





```
void vecAdd(float *A_h, float *B_h, float *C_h, int n)
{
    int size = n * sizeof(float);
    float *A_d, *B_d, *C_d;

    1. // Allocate memory on the device for the vectors
    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

    // Copy vectors A and B to the device
    cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

    // Code executed on the host

    2. // Call computational kernel - More details follow

    // Other code on the host

    3. // Copy results to the host
    cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

    // Free memory on the device
    cudaFree(A_d);
    cudaFree(B_d);
    cudaFree(C_d);
}
```



Computational kernel

```
global void vecAddKernel(float *A_d, float *B_d, float *C_d, int n)
{
    int i = threadIdx.x;
    C_d[i] = A_d[i] + B_d[i];
}
```

Device Code

```
int vecAddKernel_h(float *A_h, float *B_h, float *C_h, int n)
{
    // Initialization code we presented earlier

    // Call of computational kernel
    vecAddKernel<<<1, n>>>(A_d, B_d, C_d, n);

    // Copy results to the host
    // Free memory on device
}
```

Number of blocks

Number of threads

Host Code

Overview of computational kernel execution

```

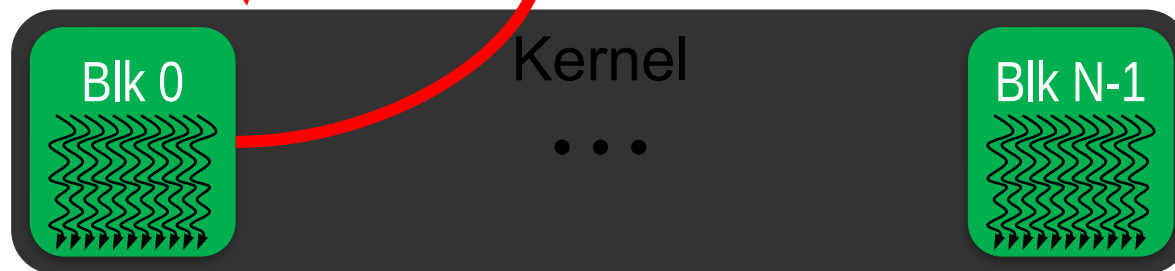
__host__
void vecAdd()
{
vecAddKernel<<<1,n>>>(A_d,B_d,C_d,n);
}

```

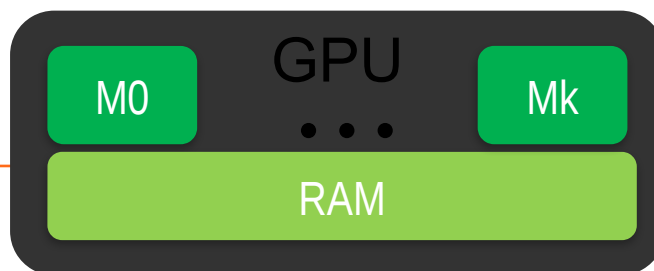
```

__global__
void vecAddKernel(float *A_d,
float *B_d, float *C_d, int n)
{
int i = threadIdx.x;
C_d[i] = A_d[i] + B_d[i];
}

```



Schedule onto multiprocessors





Function declarations in CUDA

`__global__` defines a computational kernel

Each “`__`” is composed of two underscores

A computational kernel does not return a value (return type must be void)

`__host__` defines a function that execute on the host

A normal C/C++ function

Using `__host__` is optional

`__device__` and `__host__` can be used together in a function declaration

The compiler will create two versions of the code

One that runs on the host and one that runs on the device

	Executes on	Can be called only from
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc</code>	host	host

CUDA program compilation

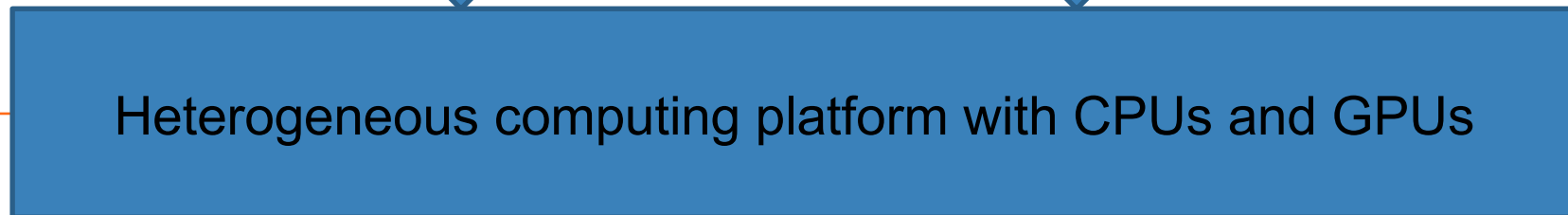
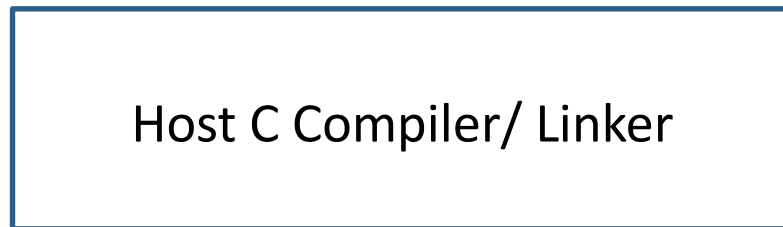
C/C++ program with CUDA extensions



Host Code



Device Code (PTX)





Let's see the complete example



```
__global__ void vecAdd(float *A_d, float *B_d, float *C_d, int n)
{
    int i = threadIdx.x;
    C_d[i] = A_d[i] + B_d[i];
}
```



```
int main(int argc, char *argv[])
{
    float *A_h, *B_h, *C_h; // Pointers to host memory
    float *A_d, *B_d, *C_d; // Pointers to device memory
    int i, size, N; // N: Vector size

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);
    if (N > 1024) {
        printf("Problem size too large.\n");
        exit(0);
    }

    size = N * sizeof(float);

    A_h = (float *)malloc(size);
    B_h = (float *)malloc(size);
    C_h = (float *)malloc(size);
    if ((A_h == NULL) || (B_h == NULL) || (C_h == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



```
for (i = 0; i < N; i++) {
    A_h[i] = (rand() % 100) / 100.00;
    B_h[i] = (rand() % 100) / 100.00;
}

// Allocate memory on the device for the vectors
cudaMalloc((void **) &A_d, size);
cudaMalloc((void **) &B_d, size);
cudaMalloc((void **) &C_d, size);

// Copy vectors A and B to the device
cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

// Call computational kernel
vecAdd<<<1, N>>>(A_d, B_d, C_d, N);

// Copy result to the host
cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

// Free memory on the device
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
}
```



Working with larger problem sizes



Vector addition (reloaded)

There exists a very important restriction

We used only 1 block of threads

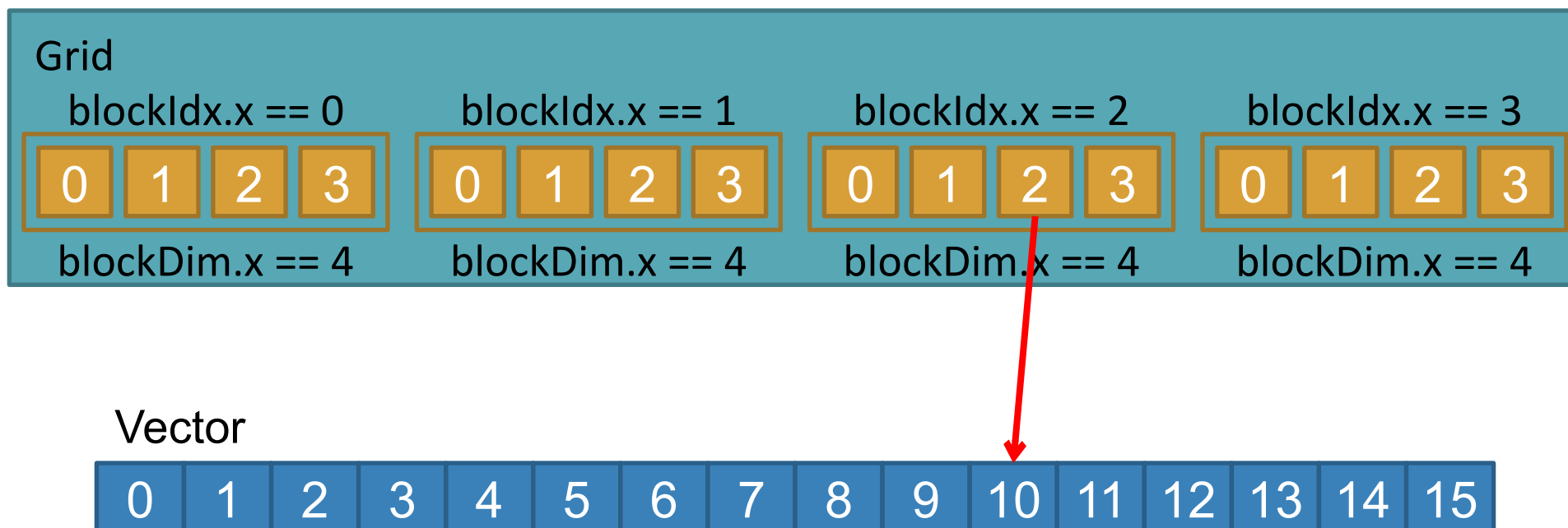
Vectors up to 1024 elements

Or 512 elements on older GPUs

How can we lift this restriction?

Correlating threads of multiple blocks to memory locations

How will each thread figure out which element in the vector it has to process?



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



```
__global__ void vecAdd(float *A_d, float *B_d, float *C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C_d[i] = A_d[i] + B_d[i];
}
```



```
int main()
{
    float *A_h, *B_h, *C_h; // Pointers to host memory
    float *A_d, *B_d, *C_d; // Pointers to device memory
    int i, size, N;         // N: Size of vectors
    int threadsPerBlock, numOfBlocks;

    if (argc != 2) {
        printf("Provide the problem size.\n");
        exit(0);
    }

    N = atoi(argv[1]);

    size = N * sizeof(float);

    A_h = (float *)malloc(size);
    B_h = (float *)malloc(size);
    C_h = (float *)malloc(size);
    if ((A_h == NULL) || (B_h == NULL) || (C_h == NULL)) {
        printf("Could not allocate memory.\n");
        exit(0);
    }
}
```



```
for (i = 0; i < N; i++) {
    A_h[i] = (rand() % 100) / 100.00;
    B_h[i] = (rand() % 100) / 100.00;
}

// Allocate memory on the host for the vectors
cudaMalloc((void **) &A_d, size);
cudaMalloc((void **) &B_d, size);
cudaMalloc((void **) &C_d, size);

// Copy vectors A and B to the device
cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

// Call computational kernel
threadsPerBlock = 4;
numOfBlocks = N / 4;
vecAdd<<<numOfBlocks, threadsPerBlock>>>(A_d, B_d, C_d, N);

// Copy results to the host
cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

// Free memory on the device
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
}
```



Maximum size of vectors?

Technical specifications	Compute capability										
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	3.7	5.0	5.2	5.3
Maximum x-dimension of a grid of thread blocks	65535					$2^{31} - 1$					
Maximum number of threads per block	512				1024						

Compute capability

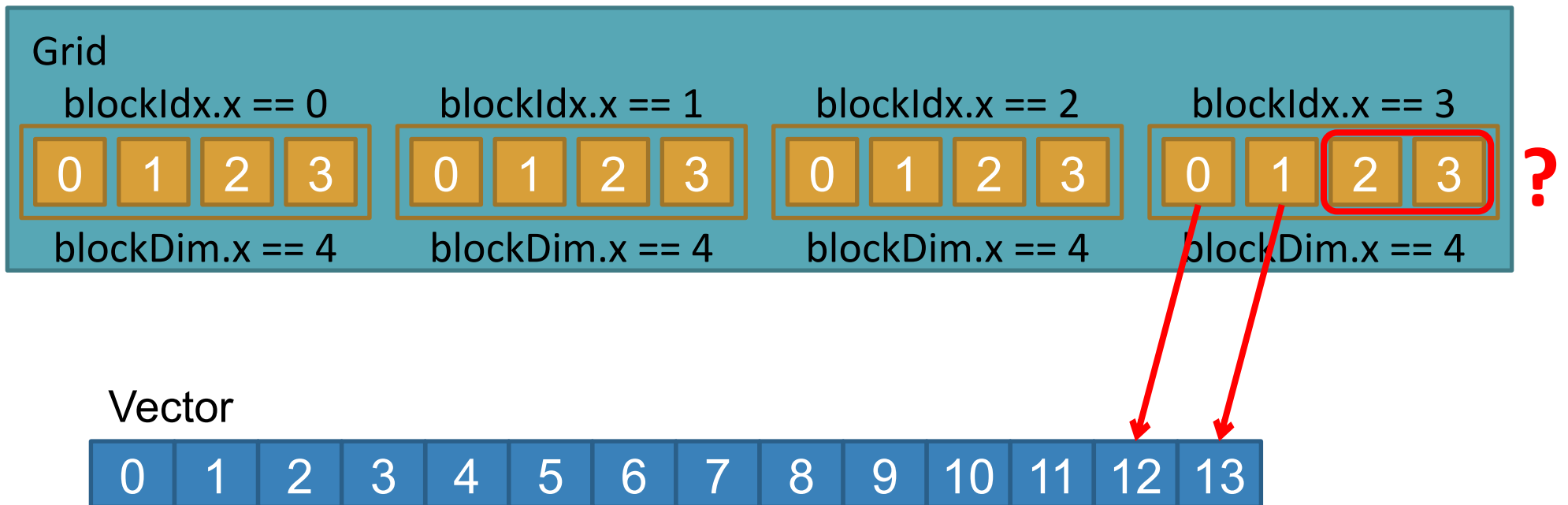
1.x: $65535 * 512$

2.x: $65535 * 1024$

$\geq 3.x: (2^{31} - 1) * 1024$

There is a last issue

What if the block size does not divide exactly the vector size?





Modified computational kernel

```
__global__ void vecAdd(float *A_d, float *B_d, float *C_d, int n)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        C_d[i] = A_d[i] + B_d[i];
    }
}
```



Alternative parallelization schemes

How we parallelized

One vector element was assigned for processing to each CUDA thread

There is no restriction in using another approach

Processing multiple elements in each CUDA thread

Not necessarily adjacent

As long as it is correct!

Ideally more efficient and/or having other characteristics that interest us



Addition of very large vectors

Important issues as we have seen

Especially on GPUs with compute capability 1.x

*Maximum 65535 blocks * 512 threads/block = 33.553.920 threads*

But these GPUs have up to 4GB memory

Can hold 1G elements of single precision

≈333.000.000 elements per vector (we need 3 vectors in our example)

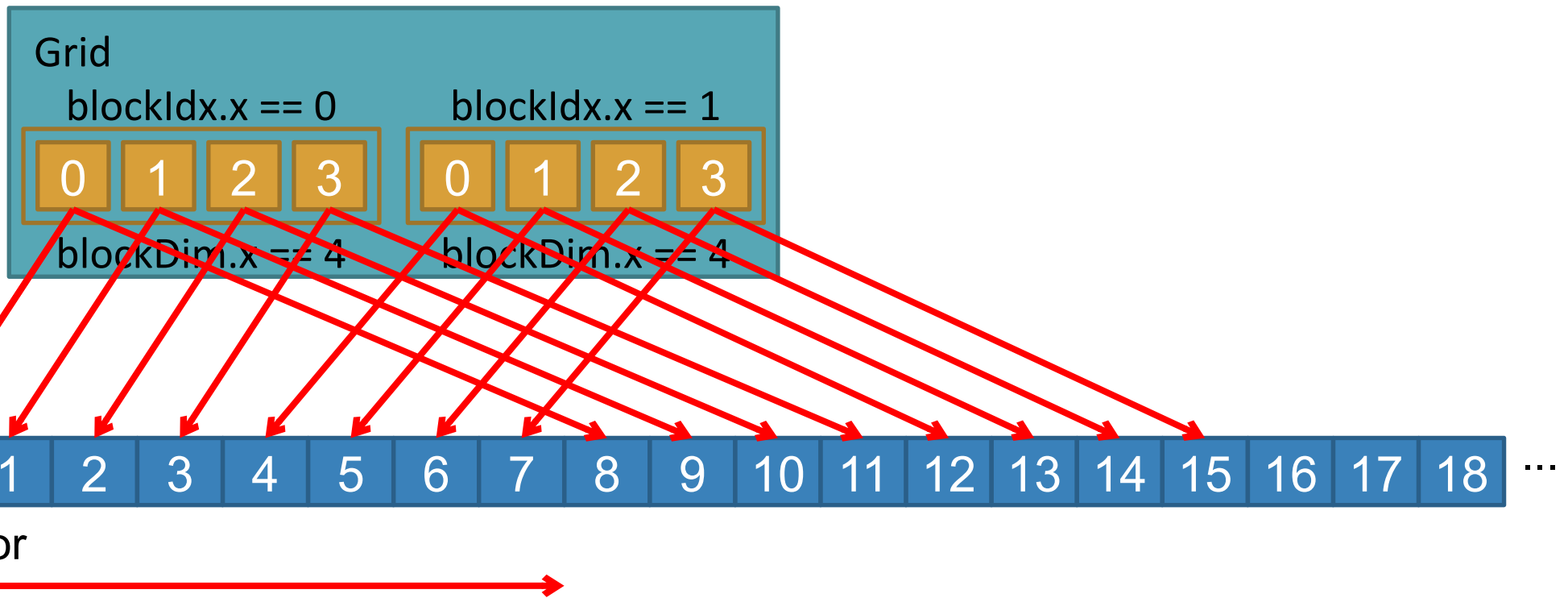
How can we add such large vectors on these GPUs?

Alternative approach

We define a grid with a constant number of blocks and threads within a block

Independent of the size of the vectors

Each thread will calculate more elements of the result



$$8 = 4 * 2 = blockDim.x * gridDim.x$$



```
__global__ void vecAdd(float *A_d, float *B_d, float *C_d, int n)
{
    int firstElement = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;
    int i;

    for (i = firstElement; i < n; i += stride) {
        C_d[i] = A_d[i] + B_d[i];
    }
}
```



2-D problems



Addition of 2-D matrices

Let's expand the problem to 2 dimensions

We will assume that each CUDA thread will add 1 element from each matrix

What changes with respect to the calculation of indices?



Correlating threads to memory locations

We use blocks of size 4×4

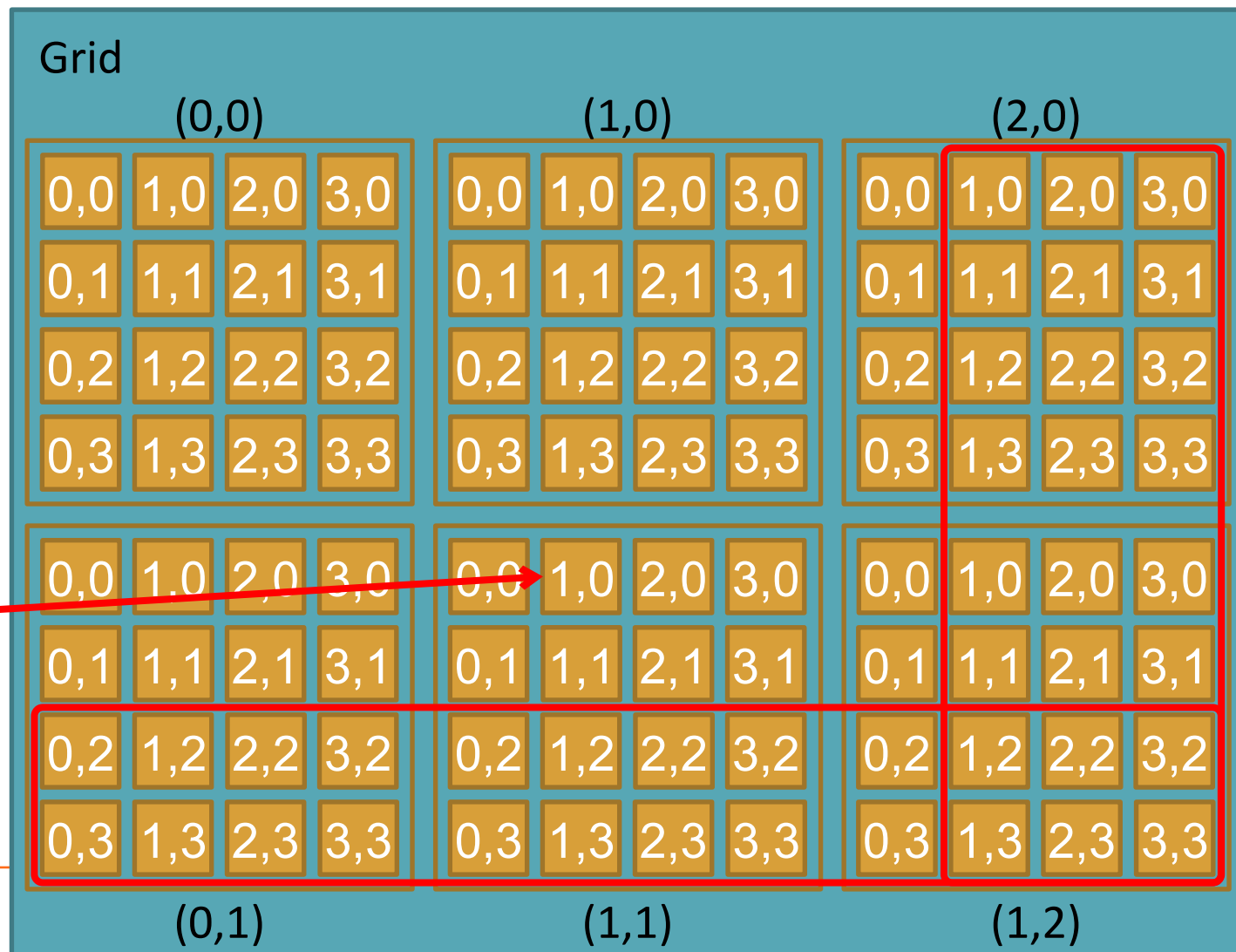
Matrix is 6×9

Matrix

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7	0,8
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7	1,8
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7	2,8
3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7	3,8
4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7	4,8
5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7	5,8

```
x = blockIdx.x * blockDim.x +  
  threadIdx.x;
```

```
y = blockIdx.y * blockDim.y +  
  threadIdx.y
```





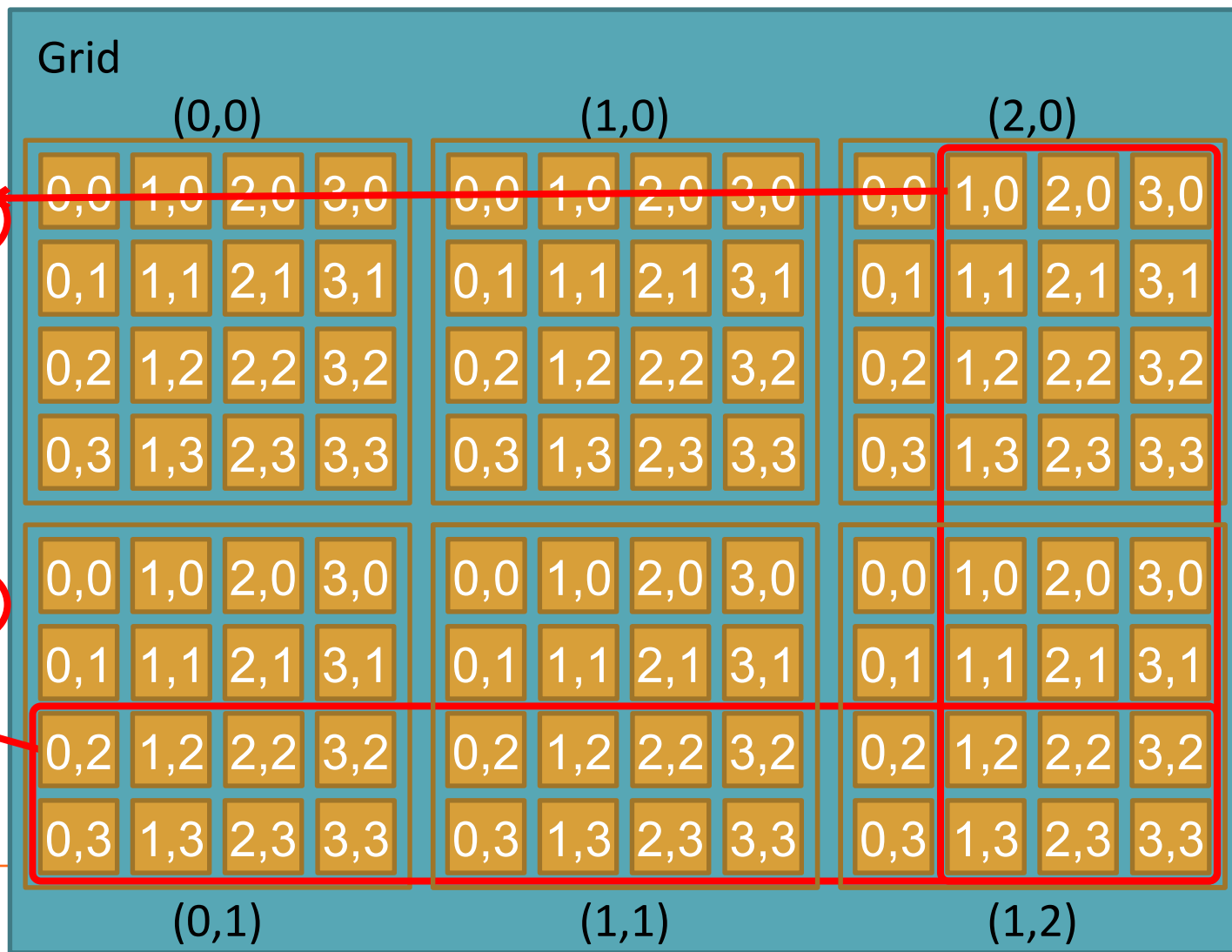
Threads exceeding matrix limits

We use blocks of size 4x4

Matrix is 6x9

$$x = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x} = 2 * 4 + 1 = 9$$

$$y = \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y} = 1 * 4 + 2 = 6$$





1st version of computational kernel

```
__global__ void matAdd(float *A_d, float *B_d, float *C_d, int width, int height)
{
    x = blockIdx.x * blockDim.x + threadIdx.x;
    y = blockIdx.y * blockDim.y + threadIdx.y;

    if ((x < width) && (y < height)) {
        C_d[?] = A_d[?] + B_d[?];
    }
}
```

Typically, matrices of any number of dimensions are passed simply as pointers to functions

How will we calculate the position of the matrices that each thread has to add together?

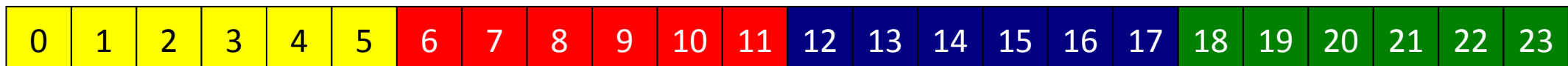
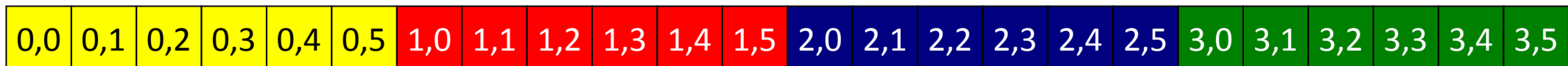
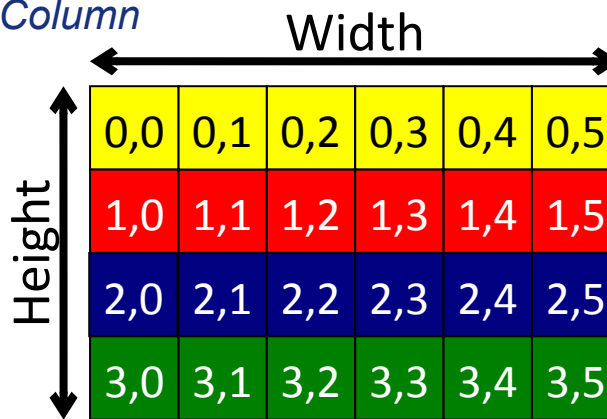


Row-major representation of matrices in C/C++

2-D matrices are stored in memory row-after-row

Position in this 1-D representation:

$Row * Width + Column$





2nd version of computational kernel

```
__global__ void matAdd(float *A_d, float *B_d, float *C_d, int width, int height)
{
    x = blockIdx.x * blockDim.x + threadIdx.x;
    y = blockIdx.y * blockDim.y + threadIdx.y;

    if ((x < width) && (y < height)) {
        C_d[y * width + x] = A_d[y * width + x] + B_d[y * width + x];
    }
}
```



Main program (1/2)

```
#define WIDTH 200
#define HEIGHT 100
float A_h[HEIGHT][WIDTH], B_h[HEIGHT][WIDTH], C_h[HEIGHT][WIDTH];

int main()
{
    float *A_d, *B_d, *C_d;    // Pointers to memory in device
    int i, j, size;

    size = WIDTH * HEIGHT * sizeof(float);

    for (i = 0; i < HEIGHT; i++) {
        for (j = 0; j < WIDTH; j++) {
            A_h[i][j] = (rand() % 100) / 100.00;
            B_h[i][j] = (rand() % 100) / 100.00;
        }
    }

    // Allocate memory on the device for the matrices
    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);
}
```



Main program (1/2)

```
// Copy matrices A and B to device
cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

// Call computational kernel
matAdd<<<?, ?>>>(A_d, B_d, C_d, WIDTH, HEIGHT);

// Copy result to host
cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

// Free memory on the device
cudaFree(A_d);
cudaFree(B_d);
cudaFree(C_d);
}
```



Number of block per grid dimension Number of threads per block dimension

For a 1-D grid and/or block

We provide the size of the grid and the size of the block simply as numbers when we call the computational kernel

```
vecAdd<<<numOfBlocks, threadsPerBlock>>>(A_d, B_d, C_d, N);
```

How to do it when we have more dimensions?

We have to define variables of the type **dim3**

A struct that can store the size per dimension

The predefined variables we have discussed earlier are also of this type



Calling the computational kernel

```
// We define each block to have a size of 32x32
unsigned int BLOCK_SIZE_PER_DIM = 32;

// Round up the number of blocks per dimension
unsigned int numBlocksX = (WIDTH - 1) / BLOCK_SIZE_PER_DIM + 1;
unsigned int numBlocksY = (HEIGHT - 1) / BLOCK_SIZE_PER_DIM + 1;

// Define the size of the grid for each dimension
dim3 dimGrid(numBlocksX, numBlockY, 1);

// Define the size of the block for each dimension
dim3 dimBlock(BLOCK_SIZE_PER_DIM, BLOCK_SIZE_PER_DIM, 1);

// Call the computational kernel
matAdd<<<dimGrid, dimBlock>>>(A_d, B_d, C_d, WIDTH, HEIGHT);
```



Multiplication of 2-D matrices



2-D matrices multiplication

Let's have a look at another 2-D problem

A bit more complicated than 2-D matrix addition

Allows us to show another way to assign calculations to CUDA threads

For simplicity we assume square matrices



Multiplication of square matrices

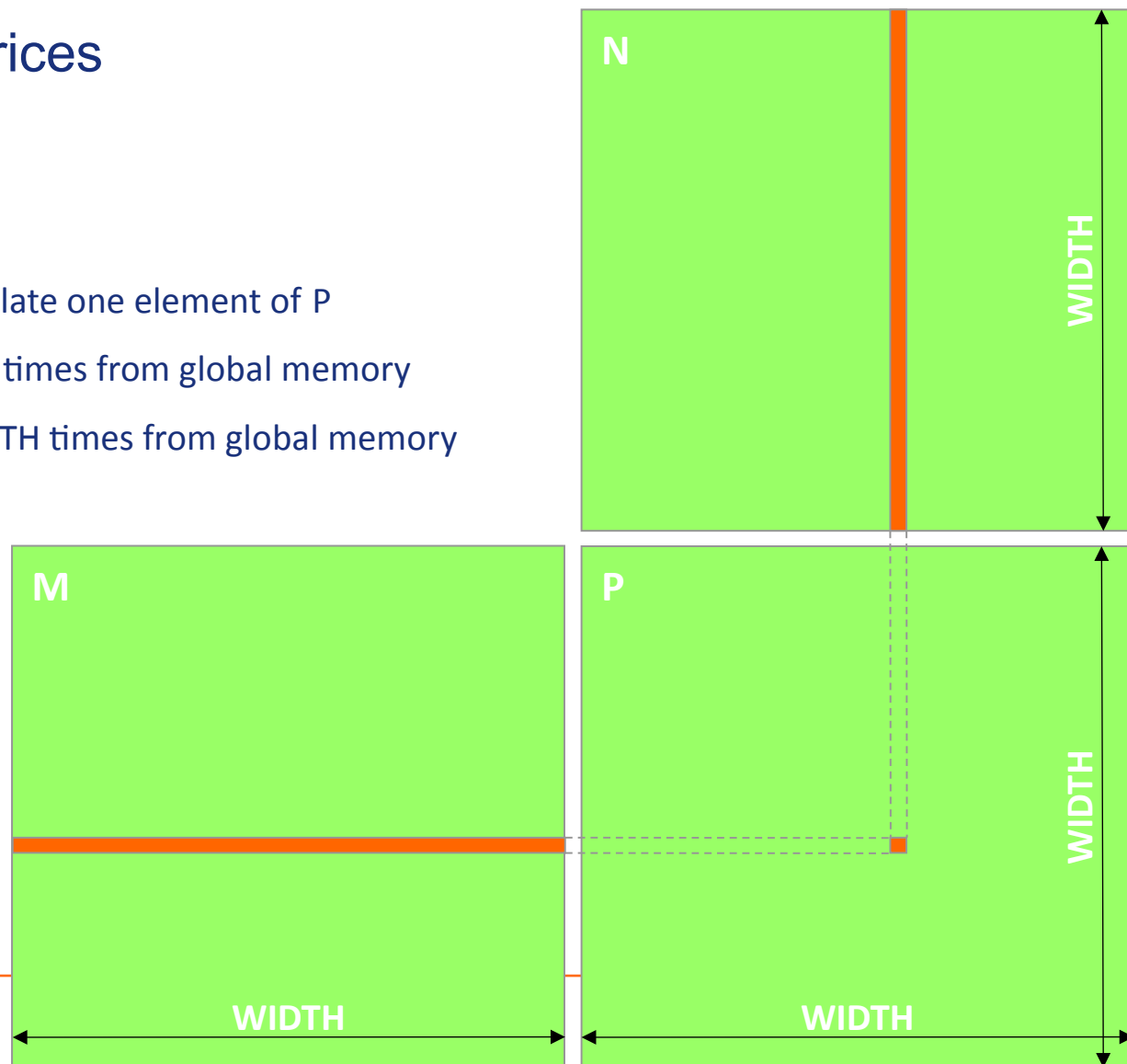
$$P = M * N$$

Size is WIDTH x WIDTH

Our strategy will be for each thread to calculate one element of P

Notice that each row of M is loaded WIDTH times from global memory

Notice that each column of N is loaded WIDTH times from global memory

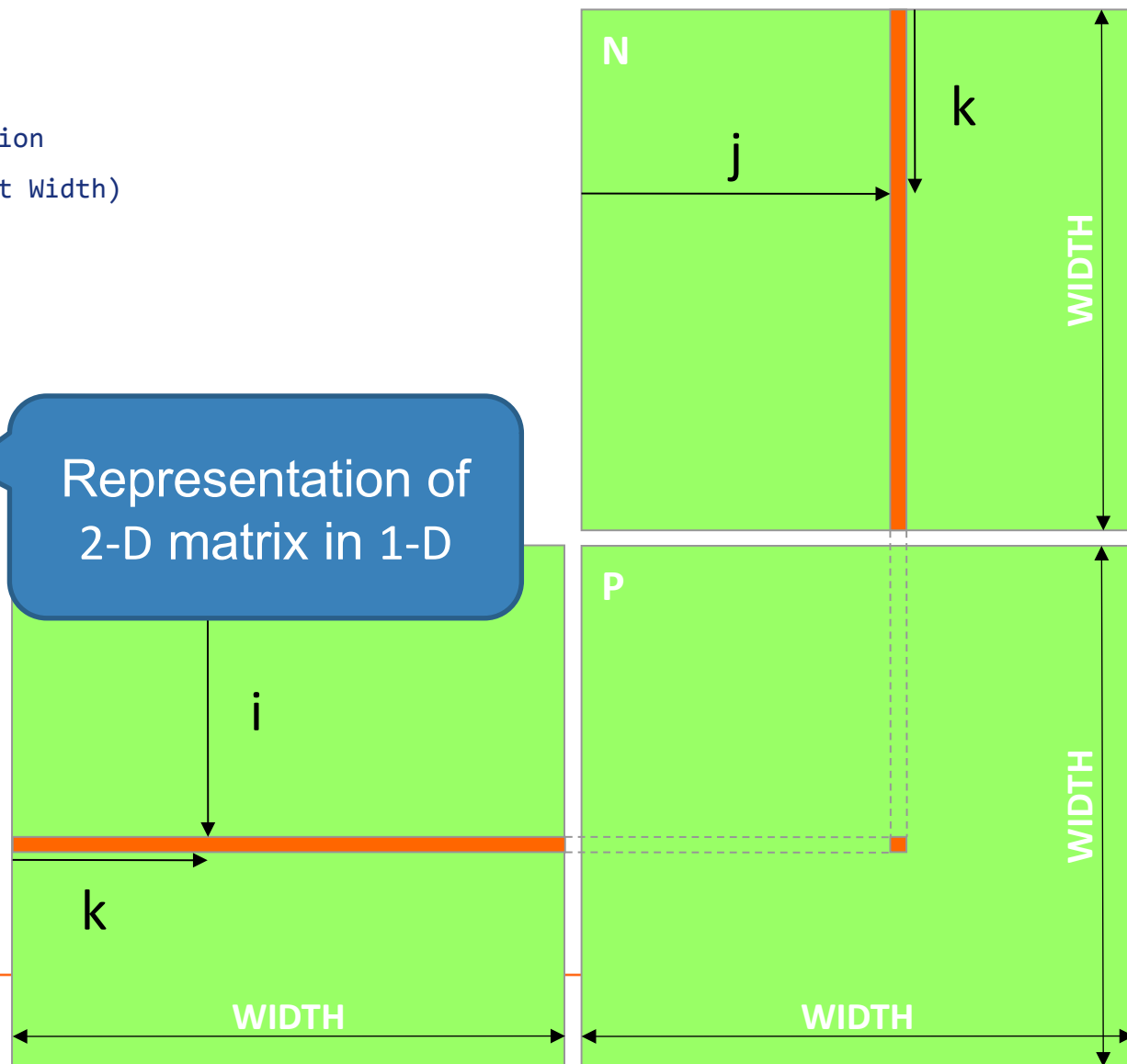




2-D matrix multiplication CPU code

```
// Matrix multiplication on the host in single precision
void MatrixMulOnHost(float *M, float *N, float *P, int Width)
{
    for (int i = 0; i < Width; i++)
        for (int j = 0; j < Width; j++) {
            double sum = 0.0;
            for (int k = 0; k < Width; k++) {
                double a = M[i * Width + k];
                double b = N[k * Width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```

Representation of
2-D matrix in 1-D

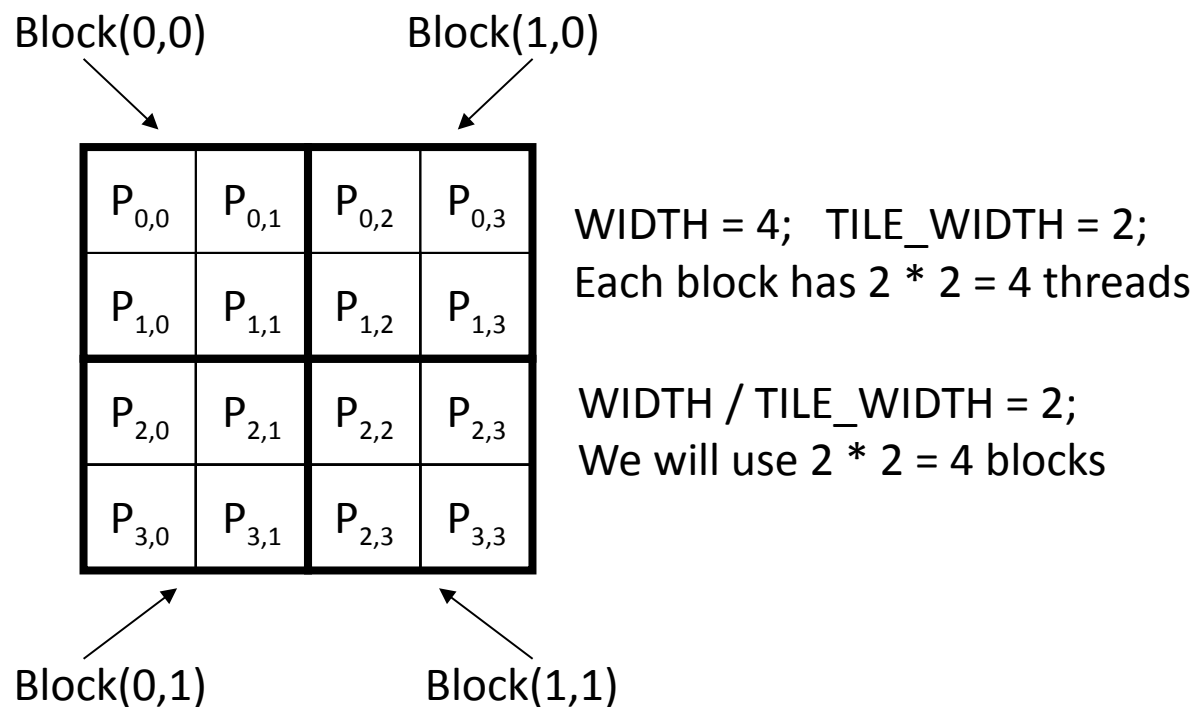


How to organize calculations

We will create blocks of threads of size $TILE_WIDTH \times TILE_WIDTH$

Each block will calculate a tile of size $TILE_WIDTH \times TILE_WIDTH$ of the result matrix P

The 2-D grid must have size $(WIDTH/TILE_WIDTH) \times (WIDTH/TILE_WIDTH)$



A larger example

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

WIDTH = 8; TILE_WIDTH = 2;
Each block has $2 * 2 = 4$ threads

WIDTH / TILE_WIDTH = 4;
We will use $4 * 4 = 16$ blocks

Using a different block size

$P_{0,0}$	$P_{0,1}$	$P_{0,2}$	$P_{0,3}$	$P_{0,4}$	$P_{0,5}$	$P_{0,6}$	$P_{0,7}$
$P_{1,0}$	$P_{1,1}$	$P_{1,2}$	$P_{1,3}$	$P_{1,4}$	$P_{1,5}$	$P_{1,6}$	$P_{1,7}$
$P_{2,0}$	$P_{2,1}$	$P_{2,2}$	$P_{2,3}$	$P_{2,4}$	$P_{2,5}$	$P_{2,6}$	$P_{2,7}$
$P_{3,0}$	$P_{3,1}$	$P_{3,2}$	$P_{3,3}$	$P_{3,4}$	$P_{3,5}$	$P_{3,6}$	$P_{3,7}$
$P_{4,0}$	$P_{4,1}$	$P_{4,2}$	$P_{4,3}$	$P_{4,4}$	$P_{4,5}$	$P_{4,6}$	$P_{4,7}$
$P_{5,0}$	$P_{5,1}$	$P_{5,2}$	$P_{5,3}$	$P_{5,4}$	$P_{5,5}$	$P_{5,6}$	$P_{5,7}$
$P_{6,0}$	$P_{6,1}$	$P_{6,2}$	$P_{6,3}$	$P_{6,4}$	$P_{6,5}$	$P_{6,6}$	$P_{6,7}$
$P_{7,0}$	$P_{7,1}$	$P_{7,2}$	$P_{7,3}$	$P_{7,4}$	$P_{7,5}$	$P_{7,6}$	$P_{7,7}$

WIDTH = 8; TILE_WIDTH = 4;
Each block has $4 * 4 = 16$ threads

WIDTH / TILE_WIDTH = 2;
We will use $2 * 2 = 4$ blocks



Calling the computational kernel from the host

```
// Setup the execution configuration
// TILE_WIDTH is a #define constant
dim3 dimGrid(Width / TILE_WIDTH, Width / TILE_WIDTH, 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(M_d, N_d, P_d, Width);
```



Computational kernel

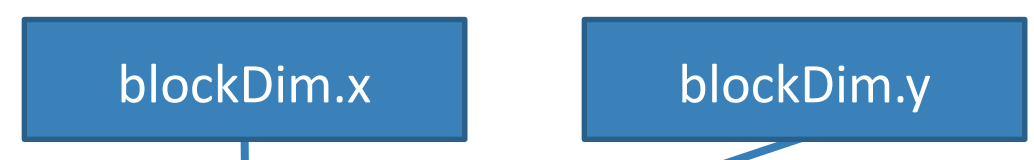
```
/*  
 * Computational kernel for 2-D matrix multiplication.  
 * The code executes on each CUDA thread.  
 */  
__global__ void MatrixMulKernel(float *M_d, float *N_d, float *P_d, int Width)  
{  
    /*  
     * The following variable is used as a temporary variable  
     * that accumulates the final value of the matrix element  
     * that is calculated on a specific thread.  
     */  
    float Pvalue = 0.0;
```

Calculations on block (0, 0) for TILE_WIDTH = 2

Col = 0
Col = 1

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}

P _{0,0}	P _{0,1}	P _{0,2}	P _{0,3}
P _{1,0}	P _{1,1}	P _{1,2}	P _{1,3}
P _{2,0}	P _{2,1}	P _{2,2}	P _{2,3}
P _{3,0}	P _{3,1}	P _{3,2}	P _{3,3}



$$\text{Col} = 0 * 2 + \text{threadIdx.x}$$

$$\text{Row} = 0 * 2 + \text{threadIdx.y}$$



Row = 0
Row = 1

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}



Calculations on block (0, 1) for TILE_WIDTH = 2

$$\text{Col} = 1 * 2 + \text{threadIdx.x}$$

$$\text{Row} = 0 * 2 + \text{threadIdx.y}$$

blockIdx.x

blockIdx.y

Col = 2
Col = 3

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

Row = 0

Row = 1

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

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



A first, simple computational kernel for 2-D matrix multiplication

```
__global__ void MatrixMulKernel(float *M_d, float *N_d, float *P_d, int Width)
{
    // Calculate the row index of the P_d element and M_d
    int Row = blockIdx.y * blockDim.y + threadIdx.y;

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

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0.0;
        // Each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; k++) {
            Pvalue += M_d[Row * Width + k] * N_d[k * Width + Col];
        }
        P_d[Row * Width + Col] = Pvalue;
    }
}
```



Execution of applications on the GPU

Each block of threads is scheduled for execution on a single SM

To fully exploit a GPU, an application must create at least as many blocks as there are SMs on the GPU

However, each SM can have more blocks active and ready for execution

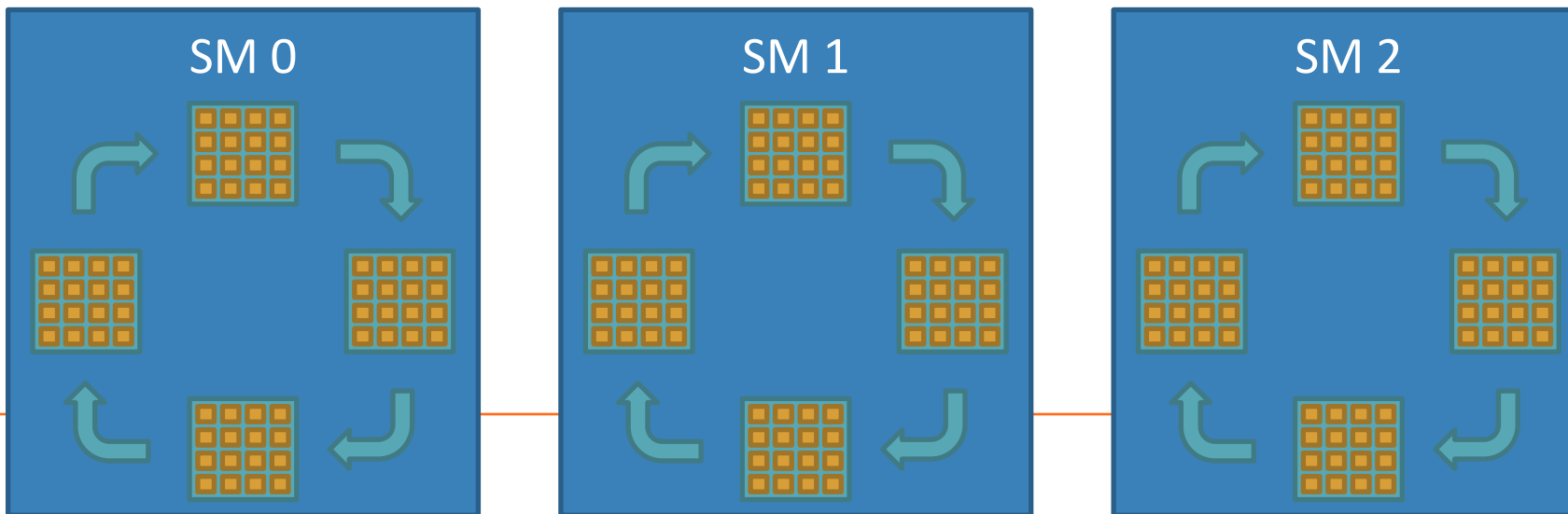
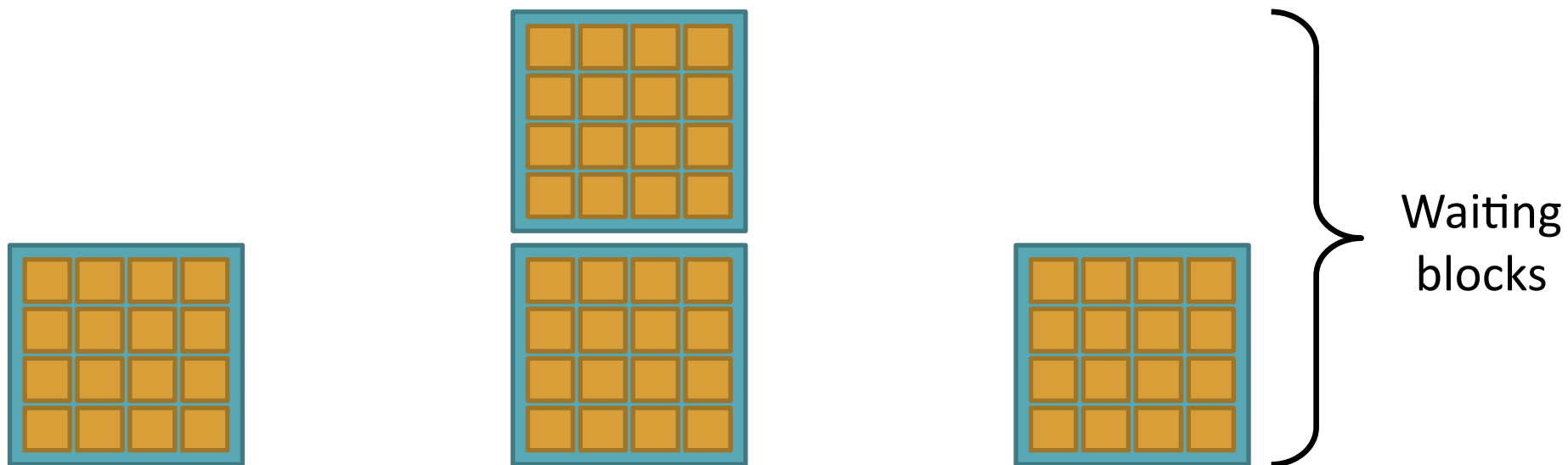
Technical specifications	Compute capability										
	1.0	1.1	1.2	1.3	2.x	3.0	3.5	3.7	5.0	5.2	5.3
Maximum number of resident blocks per multiprocessor			8			16			32		

The CUDA run-time system schedules blocks onto SMs

If the application creates more blocks than the number of blocks that can be active on a GPU?

They wait for blocks to finish execution to become active

A schematic depiction





Remarks

Due to the fact that a block of threads executes on an SM

Threads of a block can share data and can synchronize while they execute on the SM

Threads of different blocks cannot cooperate

Each block can execute in any arbitrary order in relation to the rest of the blocks

We cannot assume any order in block execution!



Restrictions on computational kernel execution

Each computational kernel

- Uses a number of registers

 - Decided by the compiler during compilation*

 - Although a maximum number can be defined by the programmer*

 - If the code requires more registers “spilling” occurs*

 - Can hurt performance*

- Is executed by every thread of a block

- As a result, execution of a kernel requires a number of registers that can be calculated as:

 - (Number of registers of kernel) * (threads per block)*

- If an SM does not have that many registers the computational kernel will not execute!

 - We have always to check for errors!

- Size of shared memory that a block can request

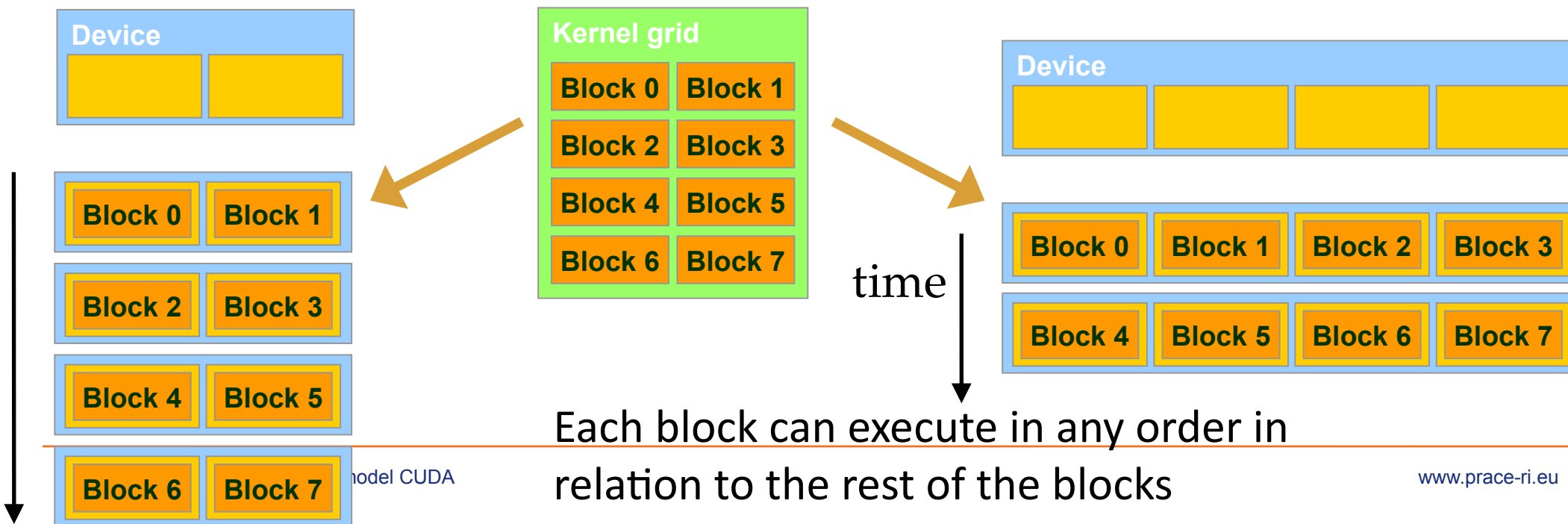
 - Cannot exceed the size of the shared memory of an SM



Transparent scalability

The hardware can execute any block of threads on any SM, at any time

A computational kernel is scalable to any number of SMs





Selecting the right size of block

For a 2-D matrix multiplication, which one is the best choice of block size on a Fermi GPU?

Blocks of size 8x8, 16x16 or 32x32;

For 8x8, we will have 64 threads per block

Each SM can have up to 1536 threads active, therefore we need 24 blocks to fully exploit an SM

But each SM can have only 8 blocks active, therefore only 512 threads will be active on each SM!

We exploit only 1/3 of the GPU!

For 16x16, we will have 256 threads per block

Each SM can have up to 1536 threads active, therefore we need 6 blocks to fully exploit an SM

We fully exploit each SM (except if there are other issues in resource allocation, like number of registers, etc.)

For 32x32, we will have 1024 threads per block

Each SM can handle only a single block

We exploit only 2/3 of the number of active threads that an SM can handle



Occupancy calculator

How do we calculate the optimal size of a block, the number of registers per thread, etc?

NVidia provides the Occupancy calculator

Excel file

http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls

CUDA Occupancy Calculator

You select basic features of the GPU

- Compute capability
- Size of shared memory

Then you change values here...

...to achieve optimal results here.

1.) Select Compute Capability (click): **3.5**
 1.b) Select Shared Memory Size Config (bytes) **49152**

2.) Enter your resource usage:
 Threads Per Block **1024**
 Registers Per Thread **32**
 Shared Memory Per Block (bytes) **0**

3.) GPU Occupancy Data is displayed here and in the graphs:
 Active Threads per Multiprocessor **2048**
 Active Warps per Multiprocessor **64**
 Active Thread Blocks per Multiprocessor **2**
 Occupancy of each Multiprocessor **100%**

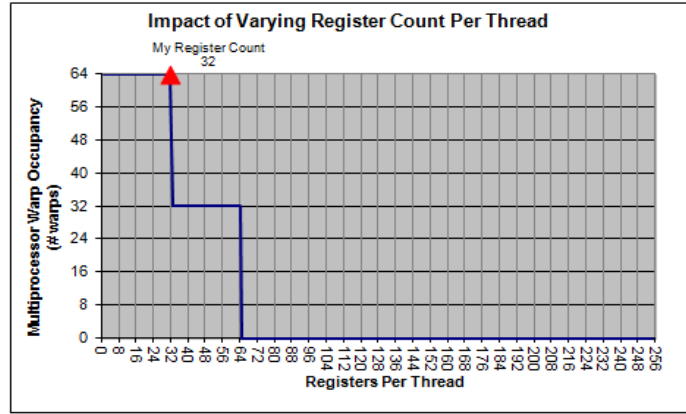
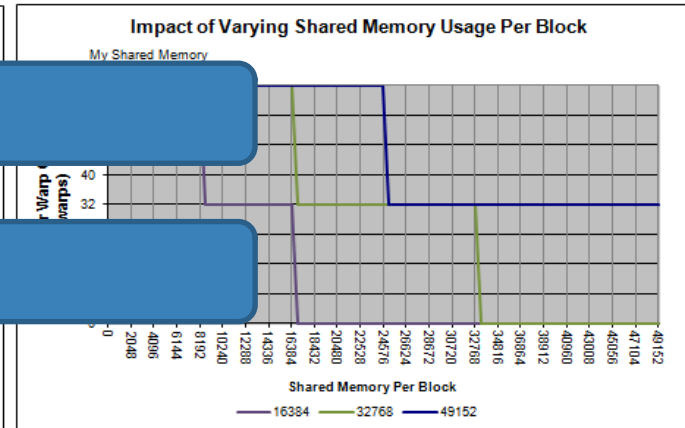
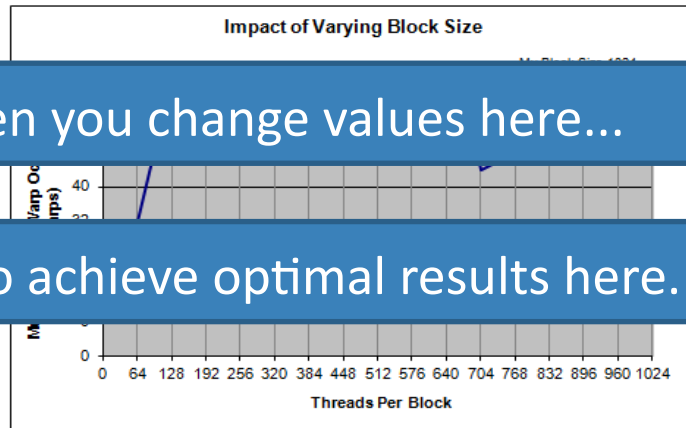
Physical Limits for GPU Compute Capability: 3.5

Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	256
Shared Memory per Multiprocessor (bytes)	49152
Max Shared Memory per Block	49152
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4

Allocated Resources	Per Block	Limit Per SM	Blocks Per SM	= Allocatable
Warps (Threads Per Block / Threads Per Warp)	32	64	2	2
Registers (Warp limit per SM due to per-warp reg count)	32	64	2	2
Shared Memory (Bytes)	0	49152	16	16

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	Warps/Block	Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	2	32	64
Limited by Registers per Multiprocessor	2	32	64
Limited by Shared Memory per Multiprocessor	16		

Physical Max Warps/SM = 64
 Occupancy = 64 / 64 = 100%



CUDA Occupancy Calculator
 Version: 7.5
[Copyright and License](#)



Error handling



Sources of errors

Calls to the CUDA API

E.g., a call to `cudaMalloc()` might fail

Call to execute a computational kernel

Not enough resources

E.g., registers, shared memory, etc.

Issues during execution

E.g., accessing memory outside allocated areas



Error checking

Calls to functions from the CUDA API

All of them return an error code

```
cudaError_t cudaMalloc(void **devPtr, size_t size);
```

== cudaSuccess if the call succeeded

!= cudaSuccess if the call failed

In addition, all of them set an internal variable that can be checked by calling the function

```
cudaGetLastError()
```

Calls to execute computational kernels

No value is returned

An internal variable is set that can be checked by calling the function `cudaGetLastError()`



Macro for error checking

Exploit `cudaGetLastError()` that is common in both cases

Write a macro for error checking

Use the macro after:

Calling a function from the CUDA API

Calling a computational kernel

```
#define cudaCheckError() { \
    cudaError_t e = cudaGetLastError(); \
    if (e != cudaSuccess) { \
        printf("CUDA error %s:%d: %s\n", __FILE__, __LINE__, \
            cudaGetErrorString(e)); \
        exit(1); \
    } \
}
```



```
// Allocate memory on the device for the vectors
cudaMalloc((void **) &A_d, size);
cudaCheckError();
cudaMalloc((void **) &B_d, size);
cudaCheckError();
cudaMalloc((void **) &C_d, size);
cudaCheckError();

// Copy A and B to the device
cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
cudaCheckError();
cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);
cudaCheckError();

// Call the computational kernel
threadsPerBlock = 4;
numOfBlocks = N / 4;
vecAdd<<<numOfBlocks, threadsPerBlock>>>(A_d, B_d, C_d, N);
cudaCheckError();

// Copy result to the host
cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);
cudaCheckError();

// Free memory on the device
cudaFree(A_d);
cudaCheckError();
cudaFree(B_d);
cudaCheckError();
cudaFree(C_d);
cudaCheckError();
```



THANK YOU FOR YOUR ATTENTION

www.prace-ri.eu