ULAKBİM

TRUBA
Turkish Science e-Infrastructure

Parallel Programming with CUDA

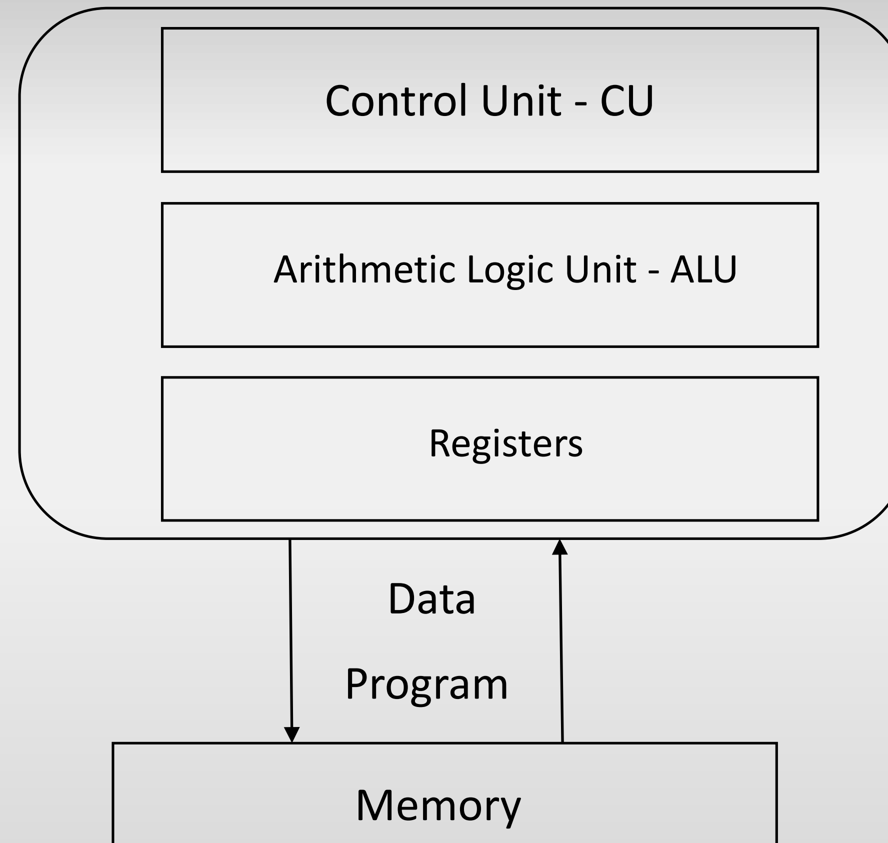
EURO²

Dr. Özcan Dülger
Bilgisayar Mühendisliği, Orta Doğu Teknik Üniversitesi
Bilgisayar Mühendisliği, Artvin Çoruh Üniversitesi

Outline

- **Day1:**
  - **Comparison of the CPU architecture and the GPU architecture**
  - **Differences between the threads of OpenMP and the threads of CUDA**
  - **Introduction to CUDA Programming**
  - **An Example Problem: Vector Addition**
- Day2:
  - Non-Coalesced Access to Global Memory Problem
  - Warp Divergence Problem
  - CUDA Streams and Multi-GPU
  - An Example Problem: Vector Addition with Streams

Von Neumann Architecture



Özcan Dülger, NCC Türkiye

Von Neumann Mimarisi

Central Processing Unit- CPU

Control Unit - CU → Instructions of the program are executed

Arithmetic Logic Unit - ALU → Arithmetical and Logical operations are performed

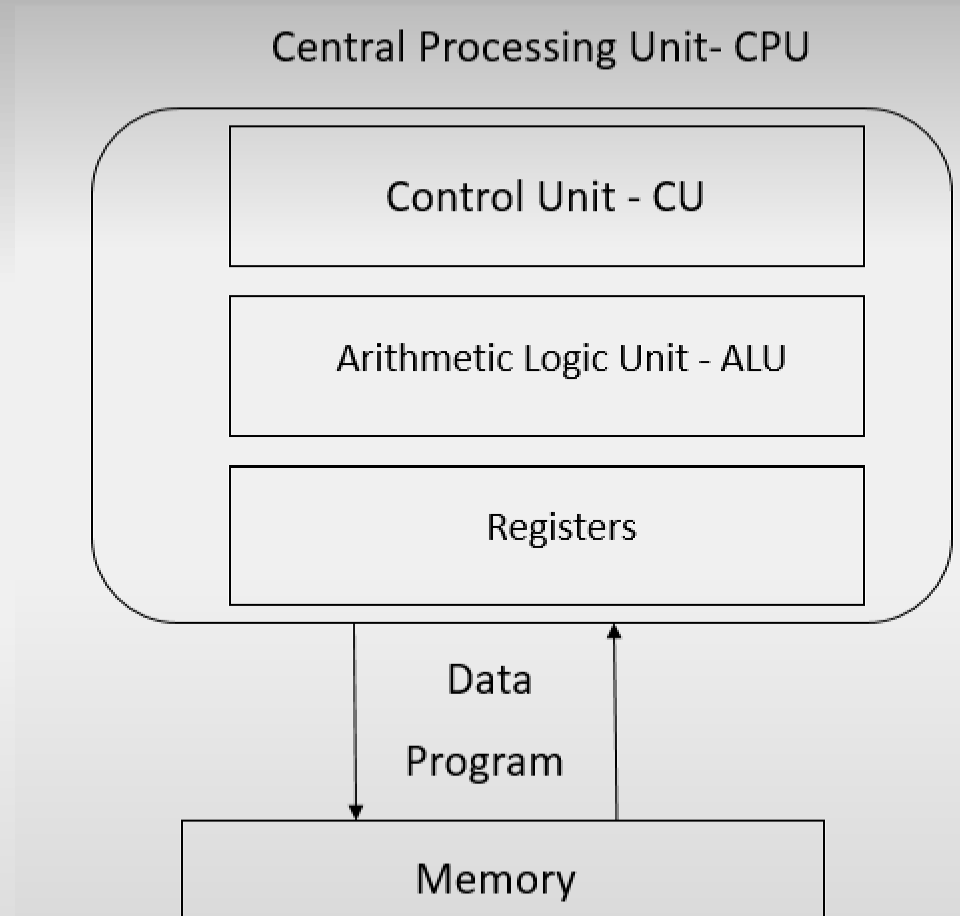
Registers → The data processed during these operations is stored temporarily

Data

Program

Memory → Data and programs are stored

Von Neumann Mimarisi



Program 1

x = 5 → Instruction1
y = x * 3 + 2 → Instruction2
*
*
*

Von Neumann Mimarisi

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Data

Program

Memory

Program 1 – Instruction 1
Program 2 – Instruction 1
Program 3 - Instruction 1
Program 2 – Instruction 2
Program 1 – Instruction 2
Program 4 – Instruction 1
Program 3 – Instruction 2
Program 1 – Instruction 3
*
*
*

Multi Core Processors

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Data

Program

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Data

Program

Memory

Özcan Dülger, NCC Türkiye

Multi Core Processors

Multi Core Processors

Central Processing Unit- CPU

- Control Unit - CU
- Arithmetic Logic Unit - ALU
- Registers

Program 1
Program 2
Program 3
Program 4

Central Processing Unit- CPU

- Control Unit - CU
- Arithmetic Logic Unit - ALU
- Registers

Central Processing Unit- CPU

- Control Unit - CU
- Arithmetic Logic Unit - ALU
- Registers

Central Processing Unit- CPU

- Control Unit - CU
- Arithmetic Logic Unit - ALU
- Registers

Multi Core Processors

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

Program 1
Program 2
Program 3
Program 4

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

Multi Core Processors

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

Program 1
Program 2
Program 3
Program 4

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

Multi Core Processors

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Program 1

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Multi Core Processors

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

Program 1

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

Central Processing Unit- CPU
Control Unit - CU
Arithmetic Logic Unit - ALU
Registers

OpenMP

- is an application programming interface (API) that supports shared-memory multiprocessing programming in C, C++, and Fortran.
- with OpenMP, the instructions of the program can be distributed to the OpenMP threads where all the threads run simultaneously on distinct cores.
- There is a default thread called master thread.

Parallel Programming with CUDA

Özcan Dülger, NCC Türkiye

OpenMP

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Master Thread

Program 1

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

OpenMP

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Master Thread

Thread 2

Thread 3

Thread 4

Program 1

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Parallel Programming with CUDA

Özcan Dülger, NCC Türkiye

OpenMP

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Master Thread

Thread 2

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Program 1

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Thread 3

Thread 4

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

OpenMP

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Master Thread

Thread 2

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Program 1

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

Thread 3

Thread 4

Central Processing Unit- CPU

Control Unit - CU

Arithmetic Logic Unit - ALU

Registers

GOMP_CPU_AFFINITY="0 1 2 3"

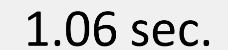
Processor Specifications

Property	Value
Processor	4. Generation Intel Core i7
Processor Number	İ7-4790K
Number of cores	4
Number of threads	8
Processor Frequency	4.00 GHz
Size of Memory	16 GB

Example 1

size is 2^{25}

```
for(counter = 0; counter < size; counter++)
{
		array[counter] = counter * 3 * 3 * 3;
}

for(counter = 0; counter < size; counter++)
{
		array[counter] = array[counter] / 3 / 3 / 3;
		array[counter] = array[counter] * 3 * 3 * 3;
		array[counter] = array[counter] / 3 / 3 / 3;
		array[counter] = array[counter] * 3 * 3 * 3;
		array[counter] = array[counter] / 3 / 3 / 3;
}
```



Example 2

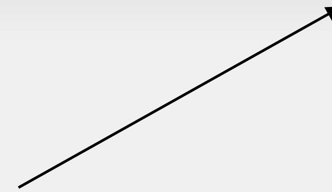
```
omp_set_num_threads(4);//  The number of threads is set
#pragma omp parallel for
        for(counter = 0; counter < size; counter++)
        {
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
        }
```

size is 2^{25}

Example 2

```
omp_set_num_threads(4);//  The number of threads is set
#pragma omp parallel for
        for(counter = 0; counter < size; counter++)
        {
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
        }
```

size is 2^{25}



1.15 sec.

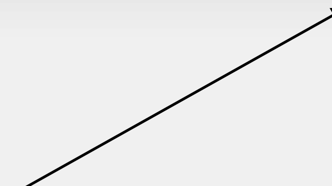
export GOMP_CPU_AFFINITY=0

Example 2

```
omp_set_num_threads(4);//  The number of threads is set
#pragma omp parallel for
        for(counter = 0; counter < size; counter++)
        {
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
        }
```

size is 2^{25}

0.57 sec.

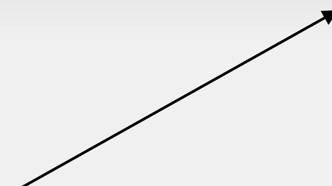


export GOMP_CPU_AFFINITY=0-1

Example 2

```
omp_set_num_threads(4);//  The number of threads is set
#pragma omp parallel for
        for(counter = 0; counter < size; counter++)
        {
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
        }
```

size is 2^{25}



0.29 sec.

export GOMP_CPU_AFFINITY=0-4

Example 3

```
for(int counter = 0;counter < size; counter++)
{
		A[counter] = 1;
		B[counter] = counter+1;
		C[counter] = size-counter;
}

for(int counter = 0; counter < size; counter++)
{
		C[counter] = A[counter] * B[counter];
}

for(int counter = 0; counter < size; counter++)
{
		D[counter] = A[counter] * C[size-counter-1];
}
```

size is 2^{25}

Example 4

```
#pragma omp parallel private(tid)
{
        tid = omp_get_thread_num();// The index of the thread

        #pragma omp for nowait
        for(int counter = 0; counter < size; counter++)
        {
                C[counter] = A[counter] * B[counter];
        }

        #pragma omp for nowait
        for(int counter = 0; counter < size; counter++)
        {
                D[counter] = A[counter] * C[size- counter-1];
        }
}
```

size is 2^{25}

The results are not stable

Example 4 – Solution 1

```
#pragma omp parallel private(tid)
{
        tid = omp_get_thread_num();// The index of the thread

        #pragma omp for nowait
        for(int counter = 0; counter < size; counter++)
        {
                C[counter] = A[counter] * B[counter];
        }

        #pragma omp barrier

        #pragma omp for nowait
        for(int counter = 0; counter < size; counter++)
        {
                D[counter] = A[counter] * C[size- counter-1];
        }
}
```

size is 2^{25}

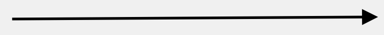
The threads wait here until all the threads reach to this point

EURO^2

Example 4 – Solution 2

```
#pragma omp parallel private(tid)
{
        tid = omp_get_thread_num();// The index of the thread

        #pragma omp for
        for(int counter = 0; counter < size; counter++)
        {
                C[counter] = A[counter] * B[counter];
        }

        #pragma omp for nowait
        for(int counter = 0; counter < size; counter++)
        {
                D[counter] = A[counter] * C[size- counter-1];
        }
}
```

size is 2^{25}



When all the threads complete their jobs, the threads continue with the next instructions

Example 4 - Solution 3

```
int complete_count = 0;// The number of threads that complete its job

#pragma omp parallel private(tid)
{
        tid = omp_get_thread_num();// The index of the thread

        #pragma omp for nowait
        for(int counter = 0;counter < size; counter++)
        {
                C[counter] = A[counter] * B[counter];
        }

        #pragma omp critical
         complete_count = complete_count + 1;

        while(complete_count < the_number_of_threads);

        #pragma omp for nowait
        for(int counter = 0;counter < size; counter++)
        {
                D[counter] = A[counter] * C[size-counter-1];
        }
}
```

size is 2^{25}

When all the threads execute this critical section, the condition of the while loop will not be satisfied

EURO²

CUDA Programming

- is a platform that runs with C, C++ ve Fortran programming language on the graphics processing unit (GPU) and enables the functions running parallel on the GPU
- works on NVIDIA GPUs
- SIMT (Single Instruction Multiple Thread) behavior

GPU Architecture (Tesla K40)

PCI Express 3.0 Host Interface

GigaThread Engine

SMX SMX SMX SMX SMX SMX SMX SMX

Memory Controller

L2 Cache

SMX SMX SMX SMX SMX SMX SMX

Streaming Multiprocessor (SMX) (15)

L2 cache

Memory access point

GPU Architecture (Tesla K40)

SMX

Instruction Cache

Warp Scheduler | Warp Scheduler | Warp Scheduler | Warp Scheduler

Dispatch | Dispatch | Dispatch | Dispatch | Dispatch | Dispatch | Dispatch | Dispatch

Register File (65,536 x 32-bit)

Core | Core | Core | DP Unit | Core | Core | Core | DP Unit | LD/ST | SFU | Core | Core | Core | DP Unit | Core | Core | Core | DP Unit | LD/ST | SFU

Interconnect Network

64 KB Shared Memory / L1 Cache

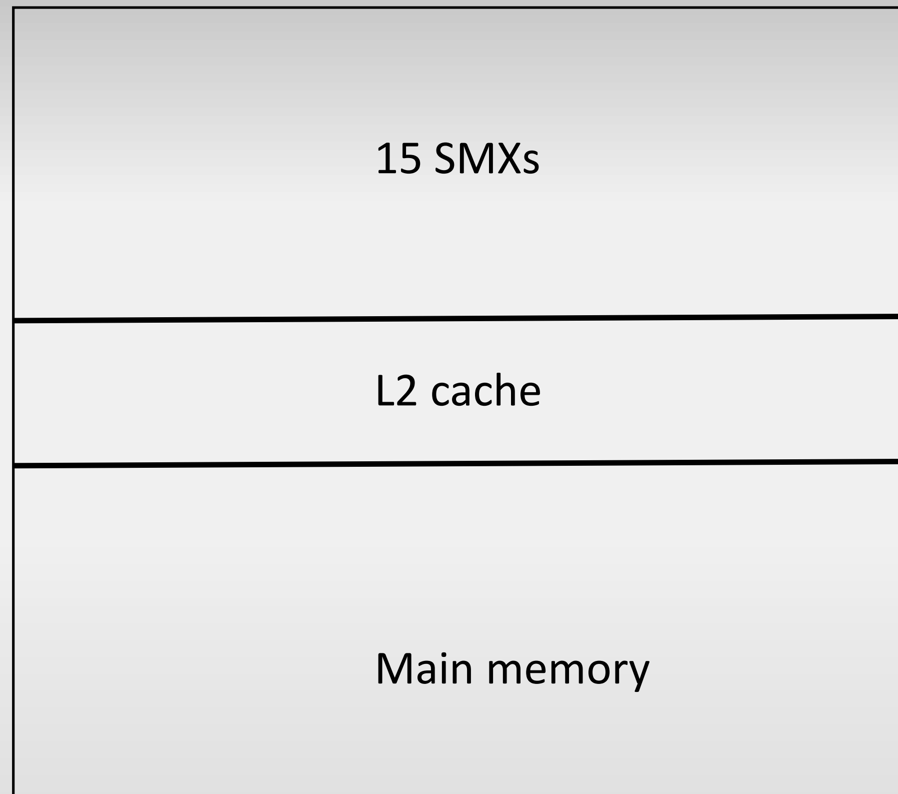
48 KB Read-Only Data Cache

Tex | Tex | Tex | Tex | Tex | Tex | Tex | Tex

Tex | Tex | Tex | Tex | Tex | Tex | Tex | Tex

- 4 Warp Scheduler
- 32-bit Registers
- 192 Cores (SP floating point units)
- 64 DP floating point units
- 32 Special functional units
- 32 Load/Store units
- 64 KB Shared Memory/L1 cache

GPU Architecture (Tesla K40)

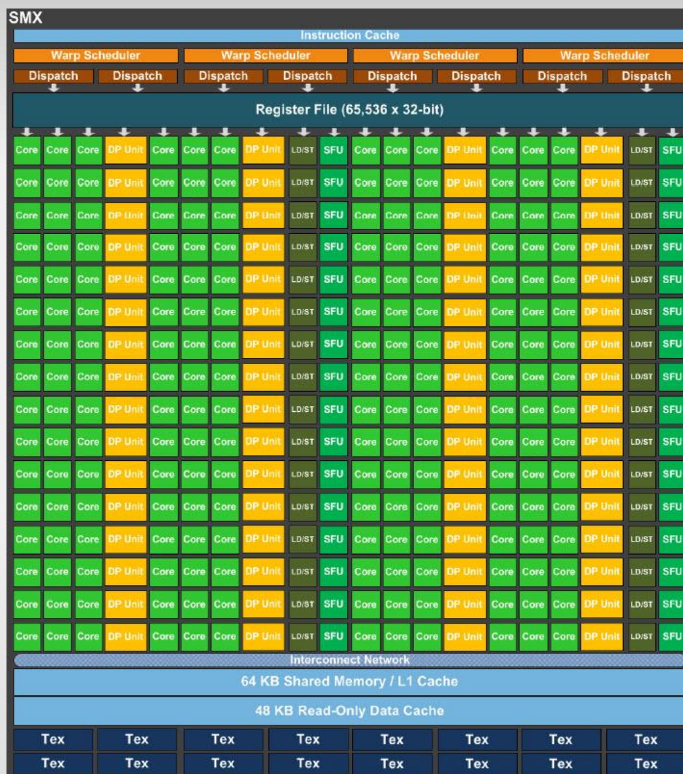


Warp

- Hardware concept
- Consists of 32 threads
- The warps are scheduled to the SMXs instead of threads
- The threads in a warp are physically related to each other
  - For any warp, in order to finish its job, all the threads in the warp must complete their job
  - x = x + 5 + y + sqrtf(x)
    - 3 memory ops, 3 arithmetic ops and 1 special function ops

Warp

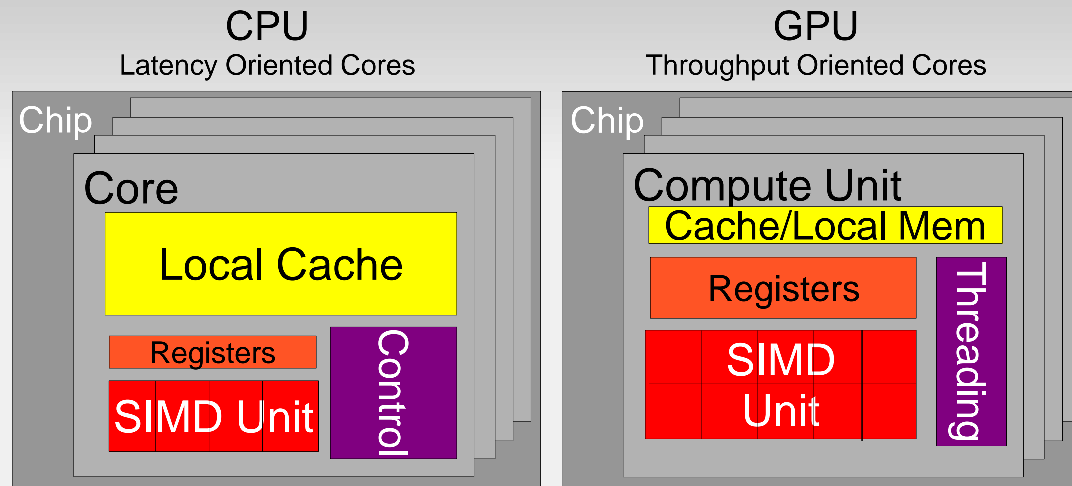
SMX 1



Warp 0 – Instruction 1
Warp 1 – Instruction 1
Warp 2 – Instruction 1
Warp 3 – Instruction 1
Warp 1 – Instruction 2
Warp 0 – Instruction 2
Warp 2 – Instruction 2
Warp 3 – Instruction 2
*
*
*
*
*

Since each SMX has 192 cores, at most 6 warp can run on an SMX at a time

Latency vs Throughput

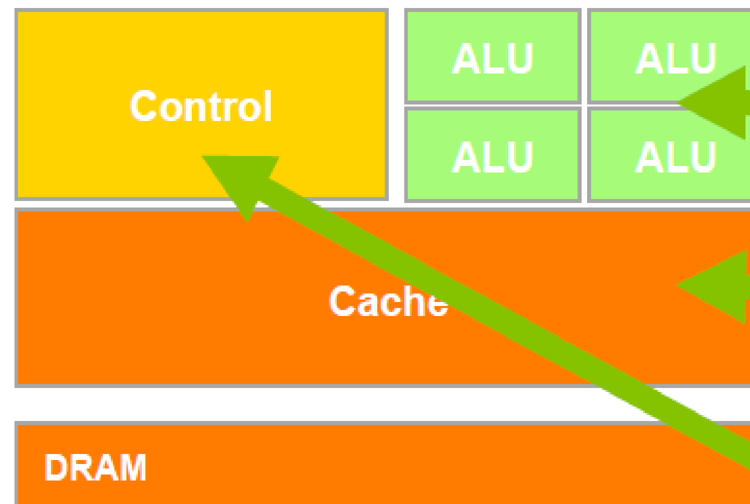


Example:
A car (speed is 300 km) with the capacity of 5 passenger (Latency oriented)
A bus (speed is 100 km) with the capacity of 40 passenger (Throughput oriented)

Ref: GPU Teaching Kit - Accelerated Computing
(licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License)

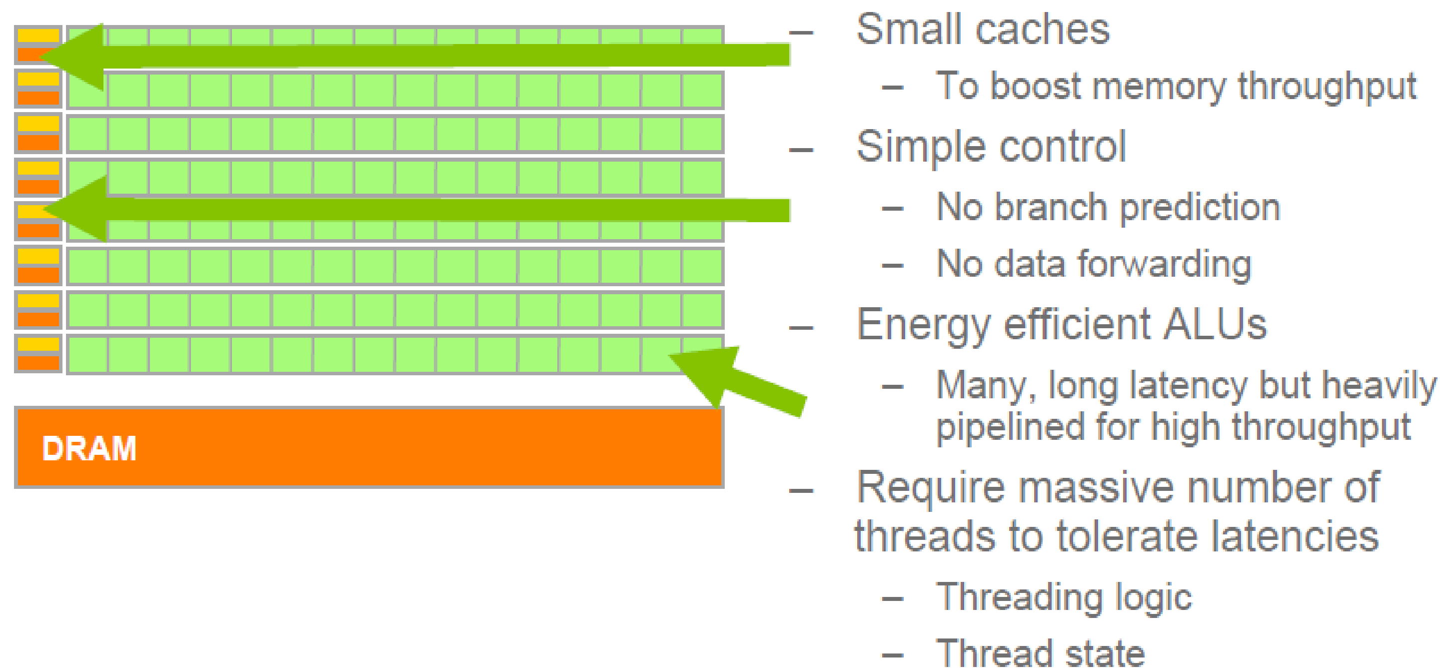
EURO²

CPUs: Latency Oriented Design



- Powerful ALU
  - Reduced operation latency
- Large caches
  - Convert long latency memory accesses to short latency cache accesses
- Sophisticated control
  - Branch prediction for reduced branch latency
  - Data forwarding for reduced data latency

Ref: GPU Teaching Kit - Accelerated Computing
(licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License)

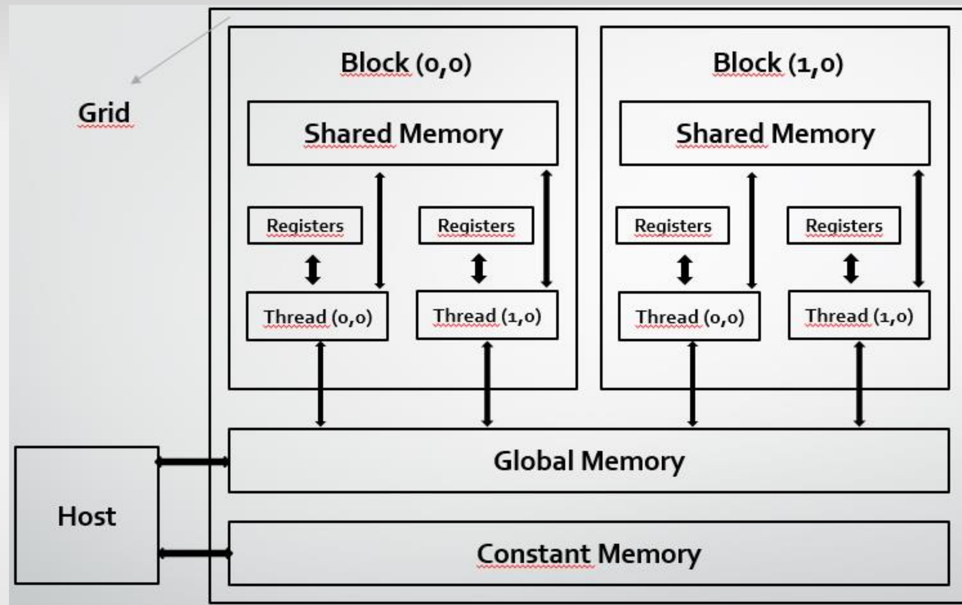


Ref: GPU Teaching Kit - Accelerated Computing

(licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License)

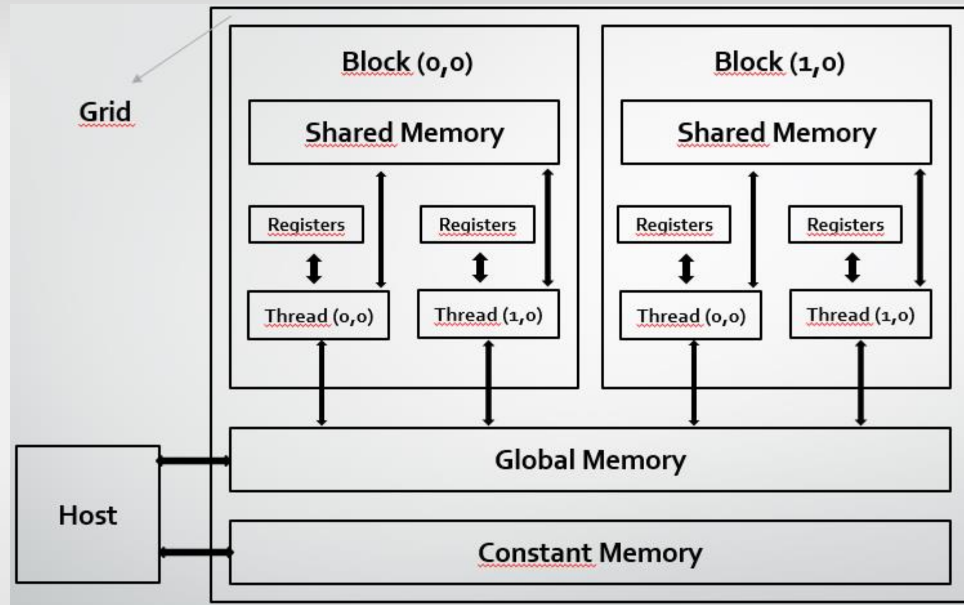
CUDA Programming Concepts

CUDA Programming Concepts



- The blocks are created in a grid.
- The threads are created as blocks.
- Each thread has a unique register and thread id.
- There is shared memory for the threads in a block where all the threads access together.
- All the threads in a block can synchronize each other, but the threads across the blocks can not synchronize
- The threads across the blocks can access L2 cache and global memory together

CUDA Programming Concepts



- Blocks are distributed to SMXs evenly.
- Threads are executed on the cores as warps.
- Warp is a hardware concept and can not be created in a program. But you can control it indirectly in the program.
- “CUDA Kernel” is a function that runs on the GPU.
- The blocks in a grid execute the same kernel.
- When all the threads in all blocks complete execution of this kernel, the execution of this kernel completes.

CUDA Kernel

```
1 #include <stdio.h>
2
3 void my_function()//The function runs on the CPU
4 {
5     printf("Hello World - CPU\n");
6 }
7
8 __global__ void my_kernel()//The kernel runs on the GPU
9 {
10     printf("Hello World - GPU\n");
11 }
12
13 int main()
14 {
15     my_function();//CPU function is called
16
17     my_kernel<<<1,1>>>();//GPU kernel is executed on the GPU
18     cudaDeviceSynchronize();//Waiting until the kernel completes execution
19
20 }
```

__global__ keyword is used

Output:
Hello World - CPU
Hello World - GPU

1 block and 1 thread in each block are created

Compile: nvcc example1.cu -o example1

CUDA Kernel

```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int var_d)//The kernel runs on the GPU
4 {
5     printf("Hello World - GPU - value = %d\n",var_d);
6 }
7
8 int main()
9 {
10     int var = 5;
11     my_kernel<<<2,32>>>(var);//GPU kernel is executed on the GPU
12     cudaDeviceSynchronize();//Waiting until the kernel completes execution
13
14 }
```

Register of the thread on the GPU

Output:
Hello World – GPU – value = 5

Stack region of the memory on the CPU

CUDA Kernel

```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int var_d)//The kernel runs on the GPU
4 {
5     printf("Hello World - GPU - value = %d\n",var_d);
6 }
7
8 int main()
9 {
10     int var = 5;
11     my_kernel<<<2,32>>>(var);//GPU kernel is executed on the GPU
12     cudaDeviceSynchronize();//Waiting until the kernel completes execution
13
14 }
```

Output:
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
*
*
*

2 blocks and 32 threads on each block are created. 64 threads in total.

CUDA Kernel

```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int var_d)//The kernel runs on the GPU
4 {
5     printf("Hello World - GPU - value = %d\n",var_d);
6 }
7
8 int main()
9 {
10     int var = 5;
11     dim3 no_of_blocks(2),no_of_threads(32);
12     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kernel is executed on the GPU
13     cudaDeviceSynchronize();//Waiting until the kernel completes execution
14 }
```

Output:
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
Hello World – GPU – value = 5
*
*
*

With “dim3” type variables, we can define the number of blocks and the number of threads in each block

CUDA Kernel

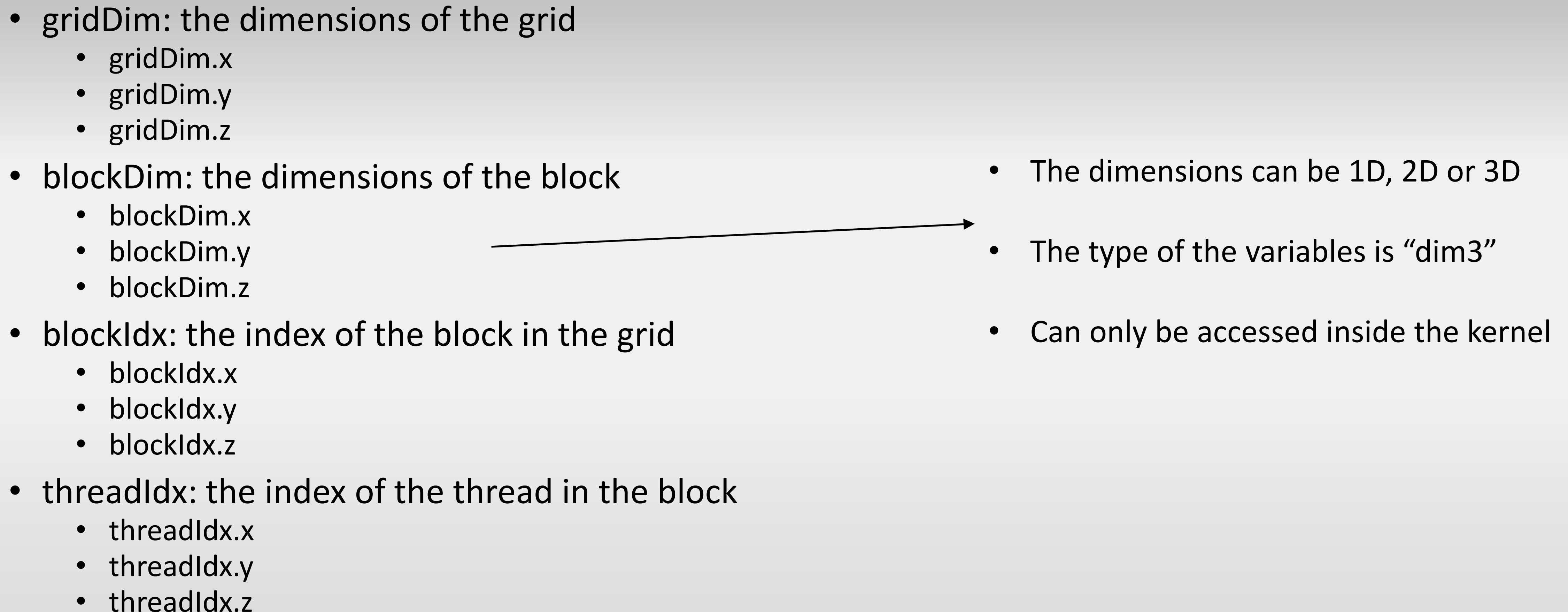
```
dim3 no_of_blocks1D(2),no_of_threads1D(32);//1D
my_kernel<<<no_of_blocks1D,no_of_threads1D>>>(var);//GPU kernel is executed on the GPU

dim3 no_of_blocks2D(2,2),no_of_threads2D(4,4);//2D
my_kernel<<<no_of_blocks2D,no_of_threads2D>>>(var);//GPU kernel is executed on the GPU

dim3 no_of_blocks3D(2,2,2),no_of_threads3D(2,2,2);//3D
my_kernel<<<no_of_blocks3D,no_of_threads3D>>>(var);//GPU kernel is executed on the GPU
```

- The number of blocks and the number of threads can be defined as 1D, 2D or 3D
- In the example, the blocks and the threads are created in three different dimensions
    - The total number of threads are same in all scenarios
    - However, the number of blocks are not same
- In the following examples, the dimension will be 1D

Parallel Programming with CUDA

Özcan Dülger, NCC Türkiye

EURO[2]



Built-in Variables

```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int var_d)//The kernel runs on the GPU
4 {
5     printf("Block No = %d - Thread No: %d\n",blockIdx.x,threadIdx.x);
6 }
7
8 int main()
9 {
10     int var = 5;
11     dim3 no_of_blocks(2),no_of_threads(4);
12     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kernel is executed on the GPU
13     cudaDeviceSynchronize();//Waiting until the kernel completes execution
14 }
```

Output:
Block No = 1 - Thread No: 0
Block No = 1 - Thread No: 1
Block No = 1 - Thread No: 2
Block No = 1 - Thread No: 3
Block No = 0 - Thread No: 0
Block No = 0 - Thread No: 1
Block No = 0 - Thread No: 2
Block No = 0 - Thread No: 3

Built-in Variables

```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int var_d)//The kernel runs on the GPU
4 {
5     printf("Block No = %d - Thread No: %d\n",blockIdx.x,threadIdx.x);
6 }
7
8 int main()
9 {
10     int var = 5;
11     dim3 no_of_blocks(2),no_of_threads(4);
12     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kernel is executed on the GPU
13     cudaDeviceSynchronize();//Waiting until the kernel completes execution
14 }
```

?

Output:
Block No = 1 - Thread No: 4
Block No = 1 - Thread No: 5
Block No = 1 - Thread No: 6
Block No = 1 - Thread No: 7
Block No = 0 - Thread No: 0
Block No = 0 - Thread No: 1
Block No = 0 - Thread No: 2
Block No = 0 - Thread No: 3

?

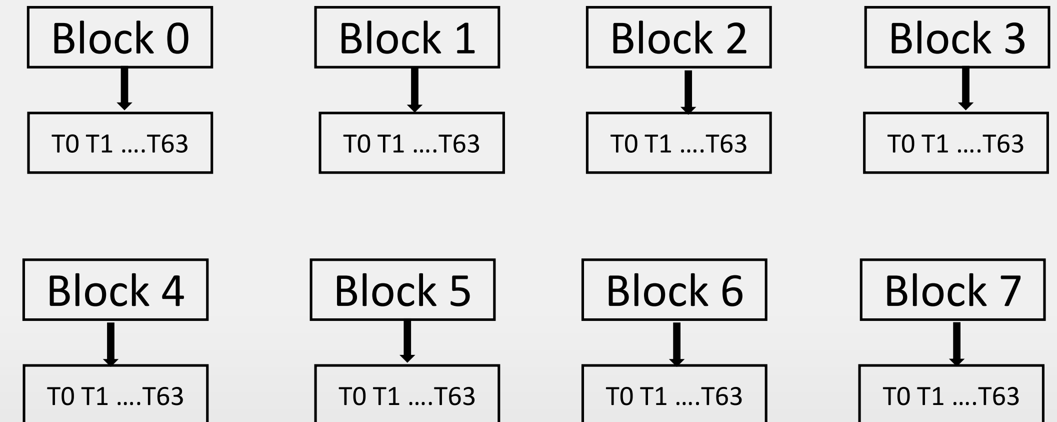
- How we obtain global (unique) thread id?

Özcan Dülger, NCC Türkiye

Built-in Variables

- The number of blocks = 8
- The number of threads in a block = 64
- Total number of threads = 512

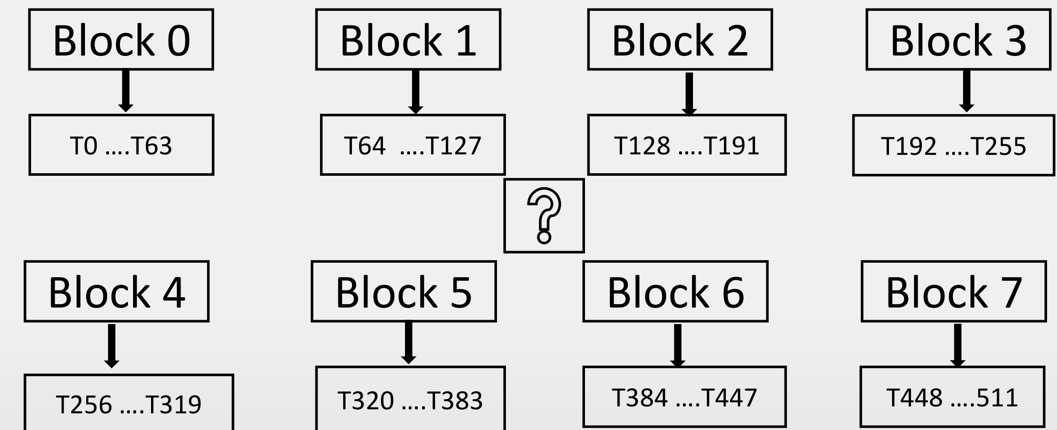
```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int var_d)//The kernel runs on the GPU
4 {
5     printf("Block No = %d - Thread No: %d\n",blockIdx.x,threadIdx.x);
6 }
7
8 int main()
9 {
10     int var = 5;
11     int gridSize = 8,blockSize = 64;
12     dim3 no_of_blocks(gridSize),no_of_threads(blockSize);
13     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kernel is executed on the GPU
14     cudaDeviceSynchronize();//Waiting until the kernel completes execution
15 }
```



Built-in Variables

- The number of blocks = 8
- The number of threads in a block = 64
- Total number of threads = 512

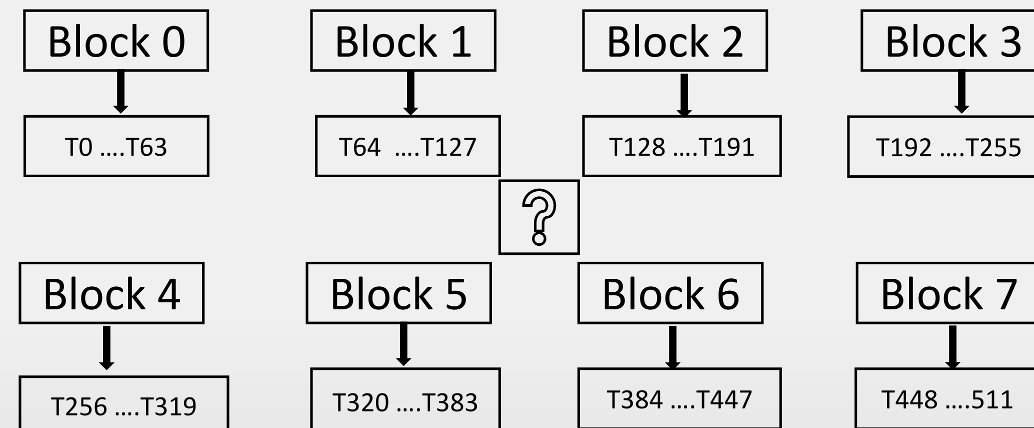
```
 1 #include <stdio.h>
 2
 3 __global__ void my_kernel(int var_d)//The kernel runs on the GPU
 4 {
 5     printf("Block No = %d - Thread No: %d\n",blockIdx.x,threadIdx.x);
 6 }
 7
 8 int main()
 9 {
10     int var = 5;
11     int gridSize = 8,blockSize = 64;
12     dim3 no_of_blocks(gridSize),no_of_threads(blockSize);
13     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kernel is executed on the GPU
14     cudaDeviceSynchronize();//Waiting until the kernel completes execution
15 }
```



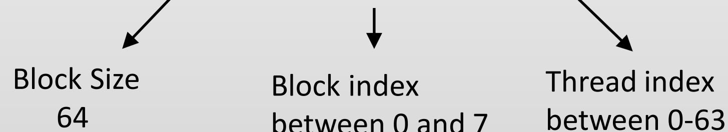
Built-in Variables

- The number of blocks = 8
- The number of threads in a block = 64
- Total number of threads = 512

```c
 1 #include <stdio.h>
 2
 3 __global__ void my_kernel(int var_d)//The kernel runs on the GPU
 4 {
 5     printf("Block No = %d - Thread No: %d\n",blockIdx.x,threadIdx.x);
 6 }
 7
 8 int main()
 9 {
10     int var = 5;
11     int gridSize = 8,blockSize = 64;
12     dim3 no_of_blocks(gridSize),no_of_threads(blockSize);
13     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kernel is executed on the GPU
14     cudaDeviceSynchronize();//Waiting until the kernel completes execution
15 }
```



- tid = blockDim.x*blockIdx.x+threadIdx.x;



EURO2

Built-in Variables

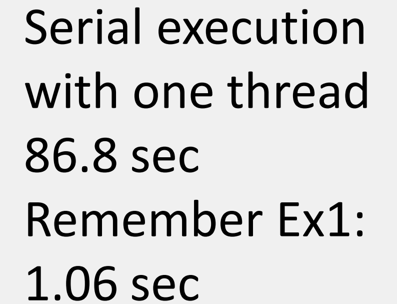
```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int var_d)//The kernel runs on the GPU
4 {
5     unsigned int tid = blockDim.x*blockIdx.x+threadIdx.x;
6     printf("Block No = %d - Thread No: %d - Global Thread No: %d\n",blockIdx.x,threadIdx.x,tid);
7 }
8
9 int main()
10 {
11     int var = 5;
12     dim3 no_of_blocks(2),no_of_threads(4);
13     my_kernel<<<no_of_blocks,no_of_threads>>>(var);//GPU kernel is executed on the GPU
14     cudaDeviceSynchronize();//Waiting until the kernel completes execution
15 }
```

Output:
Block No = 1 - Thread No: 0 - Global Thread No: 4
Block No = 1 - Thread No: 1 - Global Thread No: 5
Block No = 1 - Thread No: 2 - Global Thread No: 6
Block No = 1 - Thread No: 3 - Global Thread No: 7
Block No = 0 - Thread No: 0 - Global Thread No: 0
Block No = 0 - Thread No: 1 - Global Thread No: 1
Block No = 0 - Thread No: 2 - Global Thread No: 2
Block No = 0 - Thread No: 3 - Global Thread No: 3

Example 5

```
__global__ void Compute(float *array, int size)
{
        for(int counter = 0; counter < size; counter++)
        {
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
        }
}
Compute<<<1,1>>>(array _GPU, size);
```

size is 2^{25}



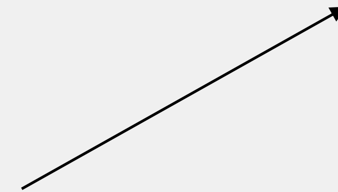
Serial execution with one thread 86.8 sec

Remember Ex1: 1.06 sec

Example 6

```
__global__ void Compute(float *array, int size)
{
        int tid_local = threadIdx.x;
        for(int counter = tid_local*(size/4); counter < (tid_local+1)*(size/4); counter ++)
        {
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
                array[counter] = array[counter] * 3 * 3 * 3;
                array[counter] = array[counter] / 3 / 3 / 3;
        }
}
Compute<<<1,4>>>(array_GPU, size);
```

size is 2^{25}



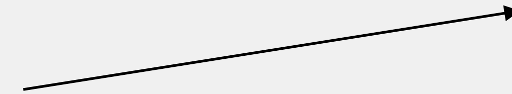
4 threads as in Example 2.
23.1 sec

Remember Ex2:
0.29 sec

Example 7

```
__global__ void Compute(float *array)
{
                int tid = blockDim.x*blockIdx.x+threadIdx.x;

                array[tid] = array[tid] / 3 / 3 / 3;
                array[tid] = array[tid] * 3 * 3 * 3;
                array[tid] = array[tid] / 3 / 3 / 3;
                array[tid] = array[tid] * 3 * 3 * 3;
                array[tid] = array[tid] / 3 / 3 / 3;
}
dim3 threadsPerBlock(1024);
dim3 numBlocks(size/1024);
Compute<<< numBlocks,threadsPerBlock >>>(array_GPU);
```

size is 2^{25}



2^{25} threads
0.005 sec.

EURO²

Example 8

size is 2^{25}

```
__global__ void Compute(float *A,float *B,float *C,float *D,int size)
{
        int tid = blockDim.x*blockIdx.x+threadIdx.x;
        C[tid] = A[tid] * B[tid];
        D[tid] = A[tid] * C[size-tid-1];
}
Compute<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,D_GPU,size);
```

→ The results are not stable!
Remember Example 4

Example 8

size is 2^{25}

```
__global__ void Compute(float *A,float *B,float *C,float *D,int size)
{
        int tid = blockDim.x*blockIdx.x+threadIdx.x;
        C[tid] = A[tid] * B[tid];
        __syncthreads();
        D[tid] = A[tid] * C[size-tid-1];
}
Compute<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,D_GPU,size);
```

Still not stable!
__syncthreads() is only valid for the threads in the same block

Example 8

size is 2^{25}

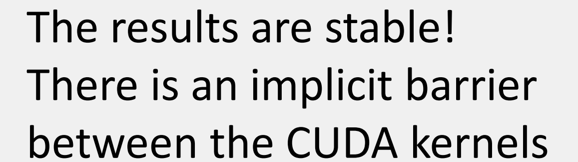
```
__global__ void Compute(float *A,float *B,float *C,float *D,int size,int *complete_count)
{
		int tid = blockDim.x*blockIdx.x+threadIdx.x;
		C[tid] = A[tid] * B[tid];
		atomicAdd(complete_count,1);
		while(complete_count[0] < size);
		D[tid] = A[tid] * C[size-tid-1];
}
Compute<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,D_GPU,size, complete_count);
```

Infinite loop!
The threads can not escape from ‘while’ loop so the waiting blocks can not be assigned to an SMX.

Example 8 Solution

size is 2^{25}

```
__global__ void Compute1(float *A,float *B,float *C,float *D)
{
        int tid = blockDim.x*blockIdx.x+threadIdx.x;
        C[tid] = A[tid] * B[tid];
}
__global__ void Compute2(float *A,float *B,float *C,float *D,int size)
{
        int tid = blockDim.x*blockIdx.x+threadIdx.x;
        D[tid] = A[tid] * C[size-tid-1];
}

Compute1<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,D_GPU);
Compute2<<<numBlocks,threadsPerBlock>>>(A_GPU,B_GPU,C_GPU,D_GPU,size);
```



CUDA Programming

- CUDA is very suitable when the number of data is very large and the same instruction applies to whole data. For an efficient application with CUDA, we must avoid:
  - Nested if-else structures
  - Large number of dependencies between the instructions
  - Nested for or while loops
  - Too many synchronizations across the blocks
  - Non coalesced access patterns on the global memory
  - Assigning different types of instructions to different threads

Özcan Dülger, NCC Türkiye

EURO²

CUDA Memory Operations

```
1 #include <stdio.h>
2
3 void my_function(int *A,int size)
4 {
5     for(int i=1;i<=size;i++)
6         printf("A[i] = %d\n",A[i-1]);
7 }
8
9 int main()
10 {
11     int size = 1000;//The size of the array
12     int *A_Host;
13     A_Host = new int[size];//Allocated on the heap region of the memory on the CPU
14
15     for(int i=1;i<=size;i++)//Assigning values to the array
16         A_Host[i-1] = i;
17
18     my_function(A_Host,size);//CPU function is called
19
20     delete[] A_Host;//Freeing memory location of the array
21 }
```

CUDA Memory Operations

```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int *A,int size)//CUDA kernel
4 {
5     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
6     if(tid<size)
7         printf("A[tid] = %d\n",A[tid]);
8 }
9
10 int main()
11 {
12     int size = 1000;//The size of the array
13     int ThreadPerBlock = 64;//The size of the block. It is better to be multiples of 32
14     int BlockPerGrid = (size-1)/ThreadPerBlock+1;//The number of blocks
15     int *A_Host;
16     A_Host = new int[size];//Allocated on the heap region of the memory on the CPU
17
18     for(int i=1;i<=size;i++)//Assigning values to array
19         A_Host[i-1] = i;
20
21     int *A_GPU;
22     cudaMalloc(&A_GPU,sizeof(int)*size);//Allocated on the global memory of the GPU
23     cudaMemcpy(A_GPU,A_Host,sizeof(int)*size,cudaMemcpyHostToDevice);//Copying data from CPU to GPU
24
25     dim3 DimBlock(ThreadPerBlock);//The number of threads in a block
26     dim3 DimGrid(BlockPerGrid);//The number of blocks in a grid
27
28     my_kernel<<<DimGrid,DimBlock>>>(A_GPU,size);//CUDA kernel is executed
29     cudaMemcpy(A_Host,A_GPU,sizeof(int)*size,cudaMemcpyDeviceToHost);//Copying data from GPU to CPU
30
31     delete[] A_Host;//Freeing memory location of the array on the CPU
32     cudaFree(A_GPU);//Freeing memory location of the array on the GPU
33 }
```

- cudaMalloc(void** address, size_t size)
  - Par.1: The address of the memory location to be allocated
  - Par.2: The size of the memory location in terms of bytes
- cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)
  - Par.1: the address of the memory location where the original data will be copied
  - Par. 2: the address of the original data on the memory
  - Par. 3: the size of data in terms of bytes
  - Par. 4: transfer direction
    - cudaMemcpyHostToDevice
    - cudaMemcpyDeviceToHost
    - cudaMemcpyDeviceToDevice
- cudaFree(void* address)
  - Freeing memory location in the given address on the GPU

CUDA Memory Operations

```
#include <stdio.h>

__global__ void my_kernel(int *A,int size)//CUDA kernel
{
    int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
    if(tid<size)
        printf("A[tid] = %d\n",A[tid]);
}

int main()
{
    int size = 1000;//The size of the array
    int ThreadPerBlock = 64;//The size of the block. It is better to be multiples of 32
    int BlockPerGrid = (size-1)/ThreadPerBlock+1;//The number of blocks
    int *A_Host;
    A_Host = new int[size];//Allocated on the heap region of the memory on the CPU

    for(int i=1;i<=size;i++)//Assigning values to array
        A_Host[i-1] = i;

    int *A_GPU;
    cudaMalloc(&A_GPU,sizeof(int)*size);//Allocated on the global memory of the GPU
    cudaMemcpy(A_GPU,A_Host,sizeof(int)*size,cudaMemcpyHostToDevice);//Copying data from CPU to GPU

    dim3 DimBlock(ThreadPerBlock);//The number of threads in a block
    dim3 DimGrid(BlockPerGrid);//The number of blocks in a grid

    my_kernel<<<DimGrid,DimBlock>>>(A_GPU,size);//CUDA kernel is executed
    cudaMemcpy(A_Host,A_GPU,sizeof(int)*size,cudaMemcpyDeviceToHost);//Copying data from GPU to CPU

    delete[] A_Host;//Freeing memory location of the array on the CPU
    cudaFree(A_GPU);//Freeing memory location of the array on the GPU
}
```

- Since the number of threads in a warp is 32 and warps are scheduled to the cores, it is important to set the number of threads in a block to the multiples of 32 in order to use the resources efficiently.
- If we create 15 blocks, the number of threads is not sufficient for each element of the array. So we must create 16 blocks.
- We need a formula in order to obtain proper number of blocks.
  - (size-1)/ThreadPerBlock+1

EURO²

CUDA Memory Operations

```
1 #include <stdio.h>
2
3 __global__ void my_kernel(int *A,int size)//CUDA kernel
4 {
5     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
6     if(tid<size)
7         printf("A[tid] = %d\n",A[tid]);
8 }
9
10 int main()
11 {
12     int size = 1000;//The size of the array
13     int ThreadPerBlock = 64;//The size of the block. It is better to be multiples of 32
14     int BlockPerGrid = (size-1)/ThreadPerBlock+1;//The number of blocks
15     int *A_Host;
16     A_Host = new int[size];//Allocated on the heap region of the memory on the CPU
17
18     for(int i=1;i<=size;i++)//Assigning values to array
19         A_Host[i-1] = i;
20
21     int *A_GPU;
22     cudaMalloc(&A_GPU,sizeof(int)*size);//Allocated on the global memory of the GPU
23     cudaMemcpy(A_GPU,A_Host,sizeof(int)*size,cudaMemcpyHostToDevice);//Copying data from CPU to GPU
24
25     dim3 DimBlock(ThreadPerBlock);//The number of threads in a block
26     dim3 DimGrid(BlockPerGrid);//The number of blocks in a grid
27
28     my_kernel<<<DimGrid,DimBlock>>>(A_GPU,size);//CUDA kernel is executed
29     cudaMemcpy(A_Host,A_GPU,sizeof(int)*size,cudaMemcpyDeviceToHost);//Copying data from GPU to CPU
30
31     delete[] A_Host;//Freeing memory location of the array on the CPU
32     cudaFree(A_GPU);//Freeing memory location of the array on the GPU
33 }
```

- When 16 blocks are created, 1024 threads will be active. Since the size of the array is 1000, 24 threads face with “out of bound” error.
- By using “If” statement, we can overcome this problem.

EURO²

CUDA Variable Types

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

Note: The arrays created by the threads are allocated on the global memory.

Özcan Dülger, NCC Türkiye

CUDA Memory Operations

```
1 #include <stdio.h>
2
3 const int ThreadPerBlock = 64;//The size of the block. It is better to be the multiples of 32
4
5 __global__ void my_kernel(int *A,int size)//CUDA kernel
6 {
7     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
8
9     __shared__ int shared_A[ThreadPerBlock];//Shared array. Its size is equal to the size of the blocks
10    shared_A[threadIdx.x] = A[tid];//Each thread copies its data on the global memory to the shared memory
11
12    __syncthreads();//When all the threads in the block reach to this point, they continue with the next instruction
13
14    //All the threads in the block can access and manipulate this shared array together
15 }
16
17 int main()
18 {
19    int size = 1000;//The size of the arrays
20    int BlockPerGrid = (size-1)/ThreadPerBlock+1;//The number of blocks
21    int *A_Host;
22    A_Host = new int[size];//Allocated on the heap region of the memory on the CPU
23
24    for(int i=1;i<=size;i++)//Assigning values to array
25        A_Host[i-1] = i;
26
27    int *A_GPU;
28    cudaMalloc(&A_GPU,sizeof(int)*size);//Allocated on the global memory of the GPU
29    cudaMemcpy(A_GPU,A_Host,sizeof(int)*size,cudaMemcpyHostToDevice);//Copying data from CPU to GPU
30
31    dim3 DimBlock(ThreadPerBlock);//The number of threads in a block
32    dim3 DimGrid(BlockPerGrid);//The number of blocks in a grid
33
34    my_kernel<<<DimGrid,DimBlock>>>(A_GPU,size);//CUDA kernel is executed
35    cudaMemcpy(A_Host,A_GPU,sizeof(int)*size,cudaMemcpyDeviceToHost);//Copying data from GPU to CPU
36
37    delete[] A_Host;//Freeing memory location of the array on the CPU
38    cudaFree(A_GPU);//Freeing memory location of the array on the GPU
39 }
```

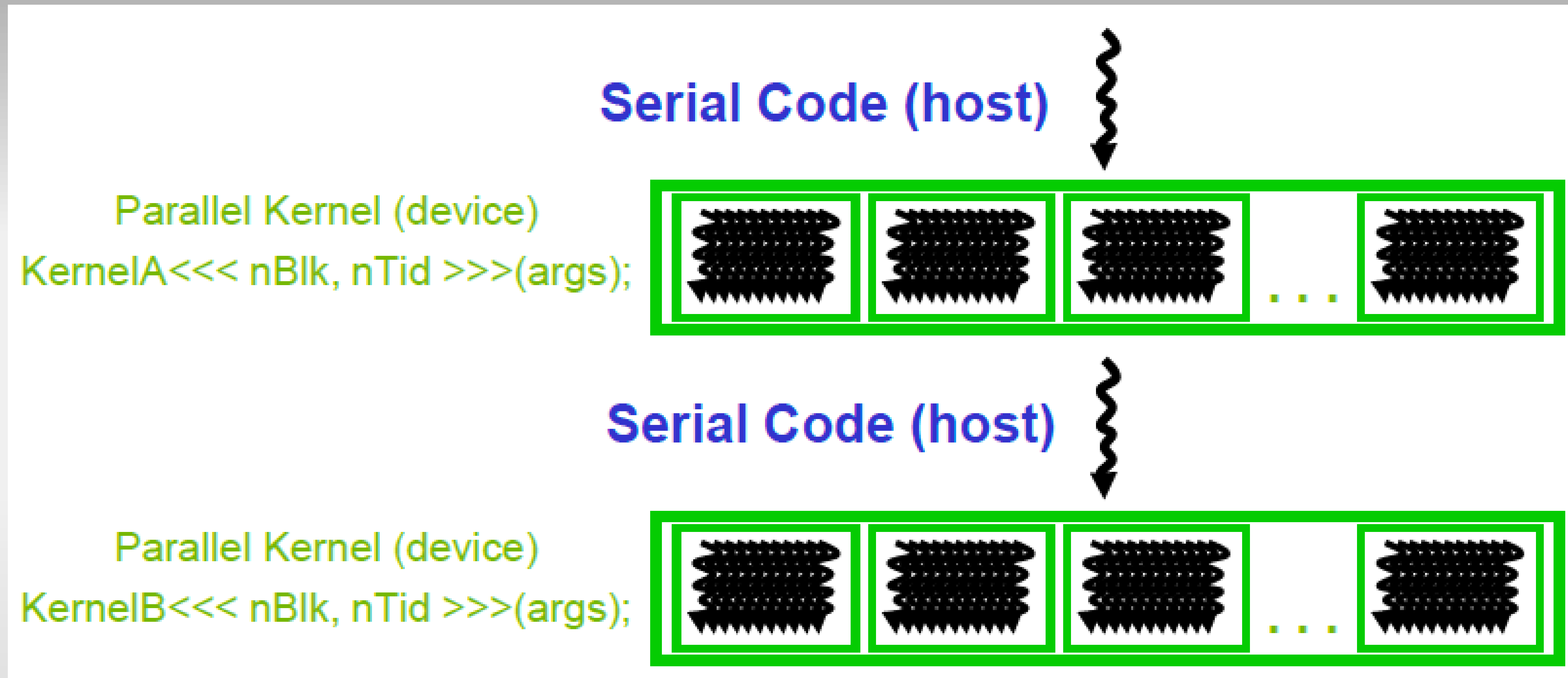
EURO2

Thread Synchronization

```
 5 __global__ void my_kernel(int *A,int size)//CUDA kernel
 6 {
 7     int tid = blockDim.x*blockIdx.x+threadIdx.x;//Global thread id
 8 
 9     __shared__ int shared_A[ThreadPerBlock];//Shared array. Its size is equal to the size of the blocks
10     shared_A[threadIdx.x] = A[tid];//Each thread copies its data on the global memory to the shared memory
11 
12     __syncthreads();//When all the threads in the block reach to this point, they continue with the next instruction
13 
14     //All the threads in the block can access and manipulate this shared array together
15 }
```

EURO²

CUDA Execution Model

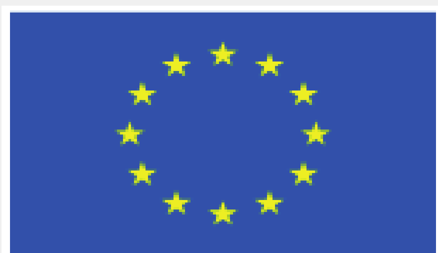


Ref: GPU Teaching Kit - Accelerated Computing
(licensed by NVIDIA and the University of Illinois under the Creative Commons Attribution-NonCommercial 4.0 International License)

Blocking-Non-Blocking Functions

- Blocking (Synchronize) Functions: The program waits these functions to complete their execution before executing the next instruction on the CPU:
    - cudaMalloc
    - cudaMemcpy
    - cudaDeviceSynchronize
    - cudaMallocHost
    - cudaFree
- Non-Blocking (Asynchronize) Functions: The program executes the next instruction on the CPU immediately:
    - CUDA Kernel
    - cudaMallocAsync
    - cudaMemcpyAsync
    - cudaMemset
    - cudaEventRecord

Regards!



This event was supported by the EuroCC 2 project. This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101101903. The JU receives support from the Digital Europe Programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Türkiye, Republic of North Macedonia, Iceland, Montenegro, Serbia.

An Example Problem

- Vector Addition
- Can be accessed: https://indico.truba.gov.tr/event/131/