CUDA: Compute Unified Device Architecture summary

P. Bassia, Prof. Ioannis Pitas
Aristotle University of Thessaloniki
pitas@csd.auth.gr
www.aiia.csd.auth.gr

Version 2.4.1

What is CUDA?

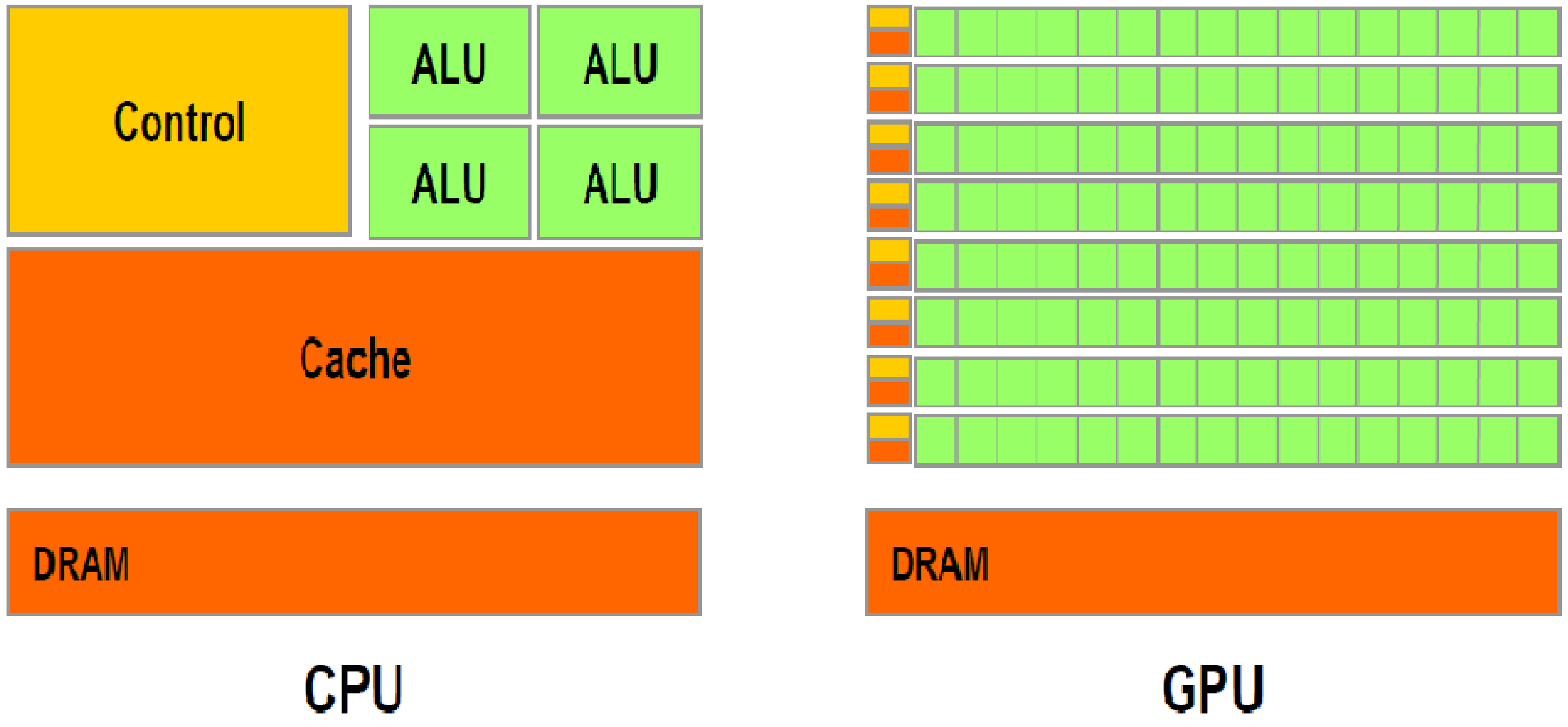
CUDA is a **parallel** computing platform and programming model invented by NVIDIA. It enables **dramatic increases** in computing performance by exploiting the power of the Graphics Processing Unit (**GPU**).

CPU vs GPU

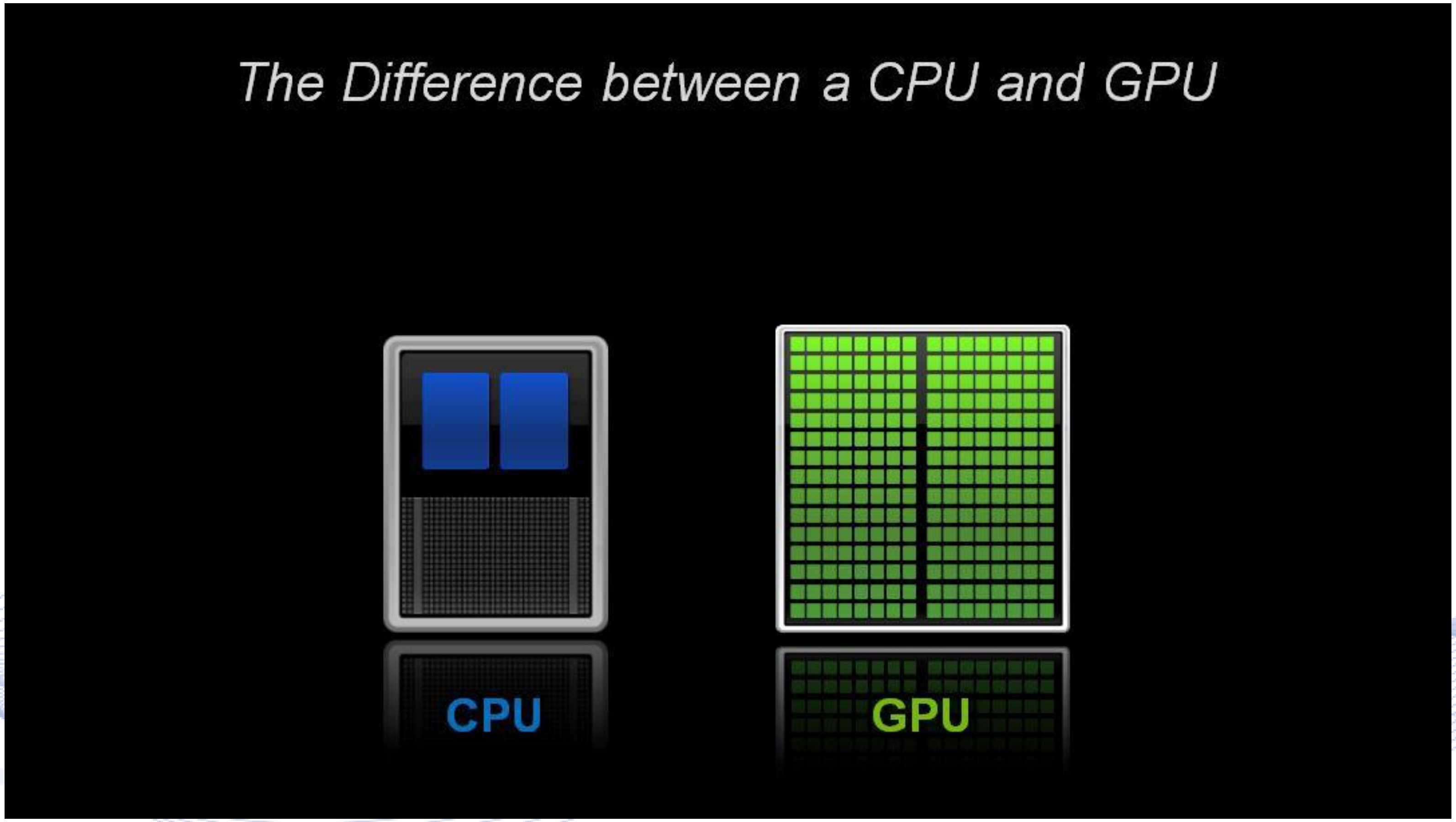
- a CPU consists of **a few** cores optimized for **sequential** serial processing.
- a GPU uses **thousands of smaller** cores which are more efficient for a massively **parallel** architecture aimed at handling multiple functions at the same time (SIMD).
- Each processing unit on a GPU contains **local memory** that improves data manipulation and **reduces fetch time**.

Artificial Intelligence & Information Analysis Lab

CPU vs GPU



CPU vs GPU

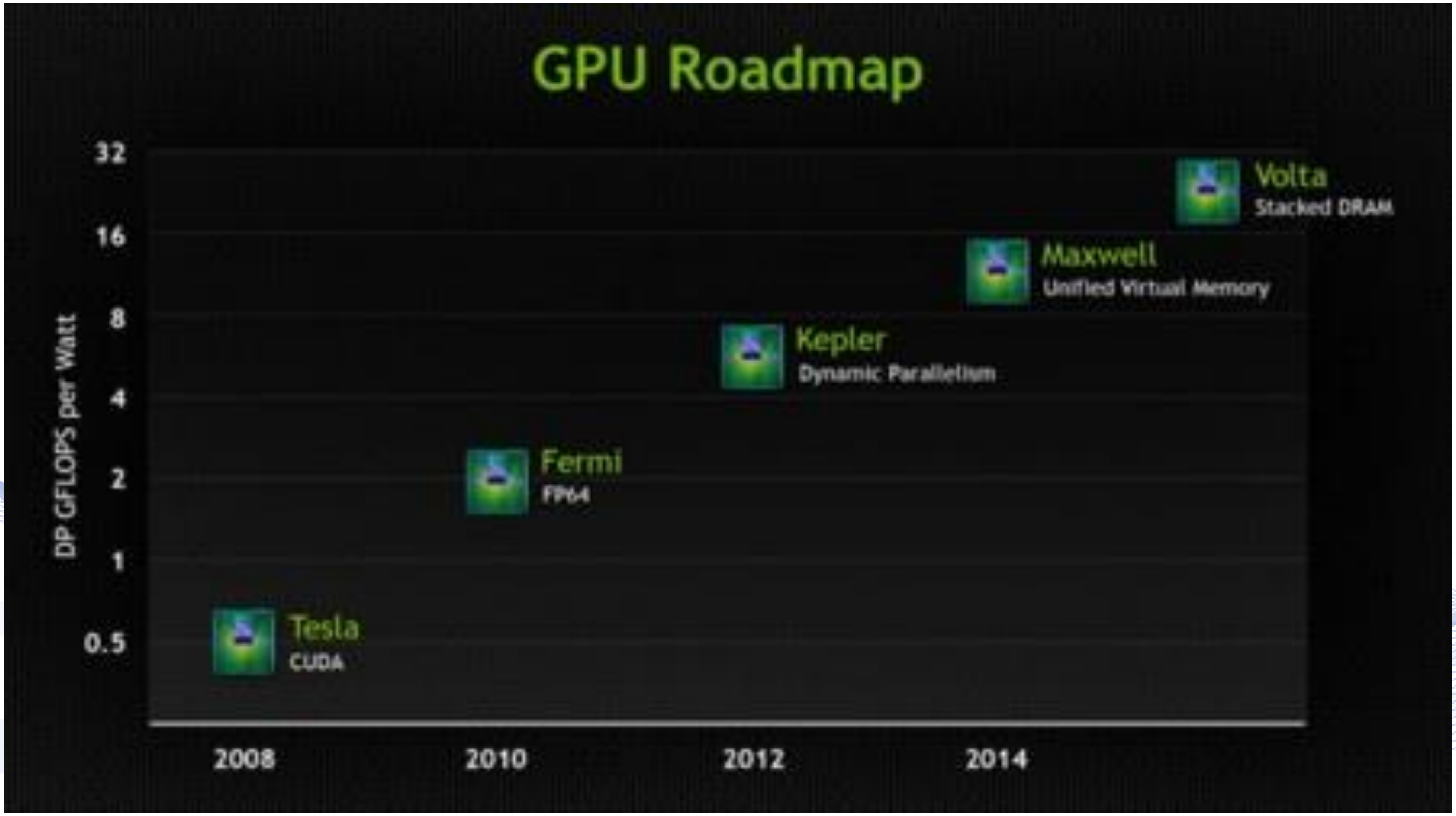


CPU vs GPU

- Multicore **CPU**: **MIMD**
  Focused on latency.
  Best single thread performance.

- Manycore **GPU**: **SIMD**
  Focused on throughput.
  Best for embarrassingly parallel tasks.

GPU microarchitecture history



and nowadays
…Pascal and Turing (2018)

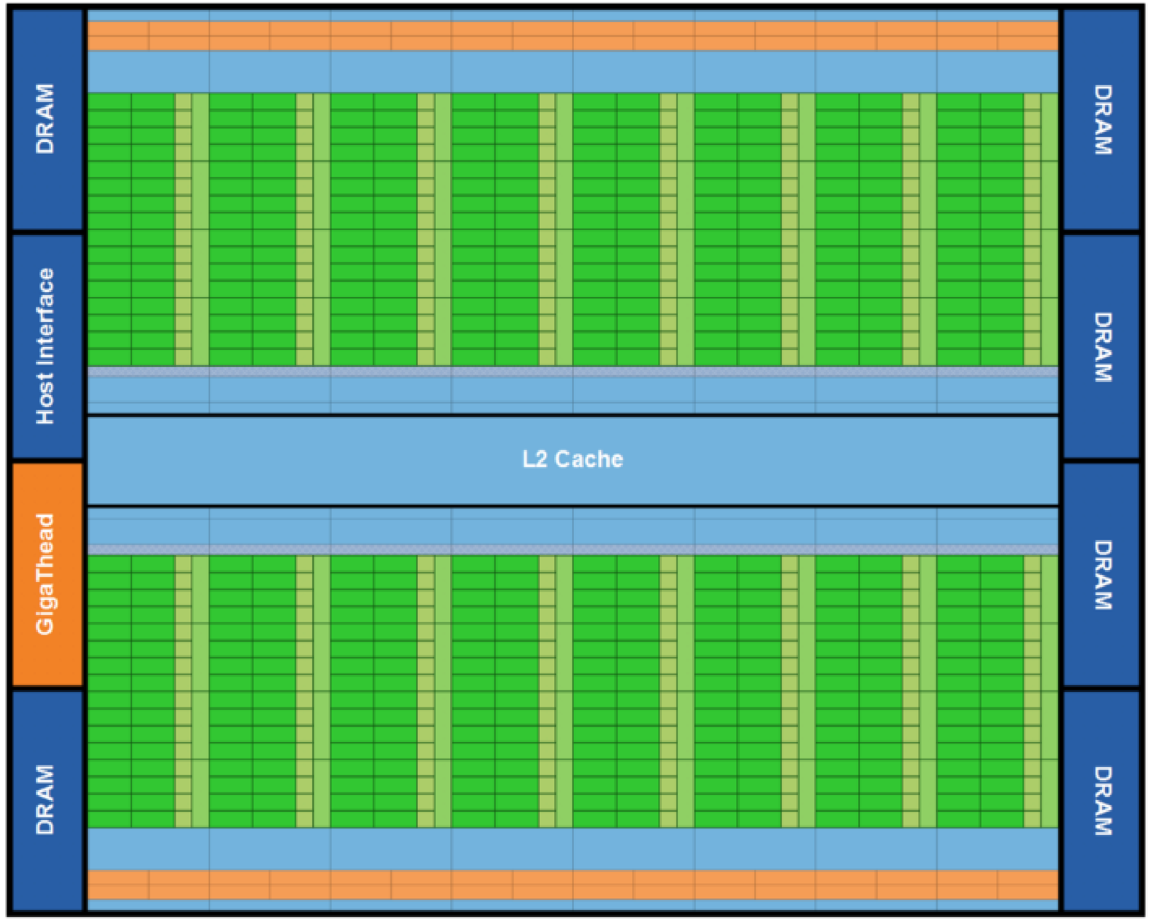
Fermi architecture

16 streaming multiprocessors (SM)

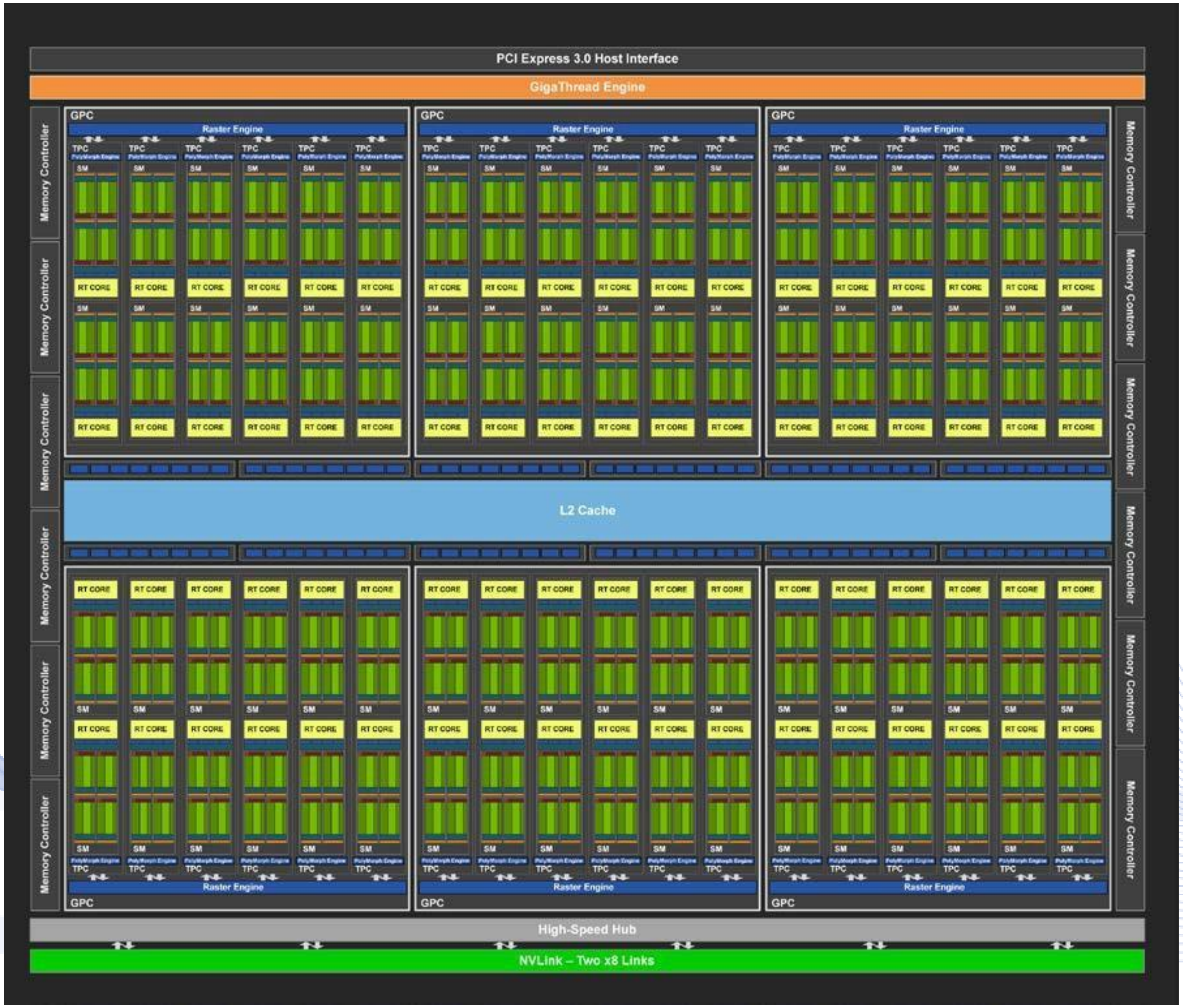
Each SM contains:

- 32 cuda cores
- 2 Warp Schedulers and dispatch units
- a bunch of registers,
- 64 KB configurable shared memory
- L1 cache

1 Warp = 32 threads

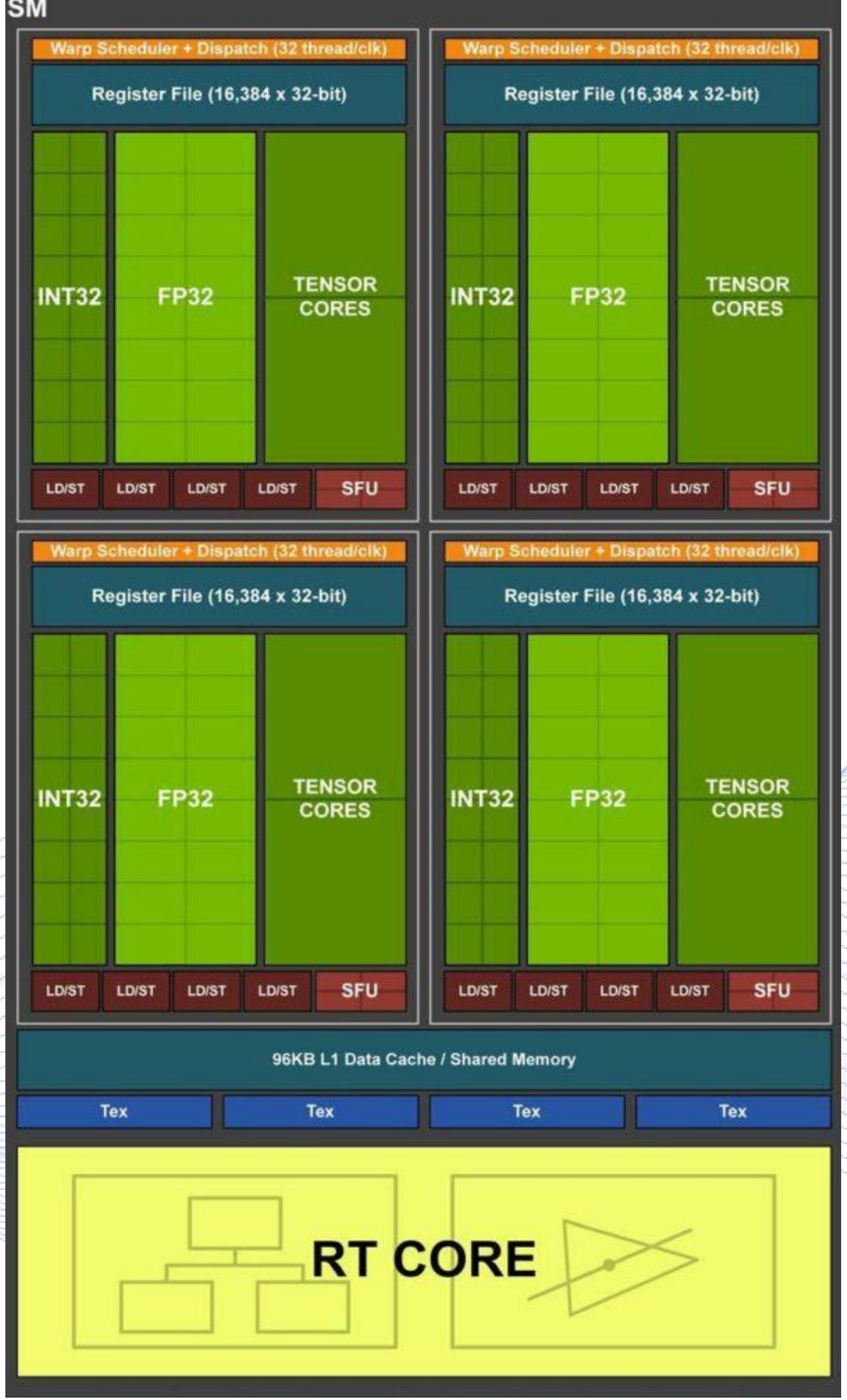


Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).



Turing TU102 Full GPU with 72 SM Units

Artificial Intelligence & Information Analysis Lab


SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
Tex
Tex
Tex
RT CORE

TURING STREAMING MULTIPROCESSOR (SM) ARCHITECTURE

Artificial Intelligence & Information Analysis Lab

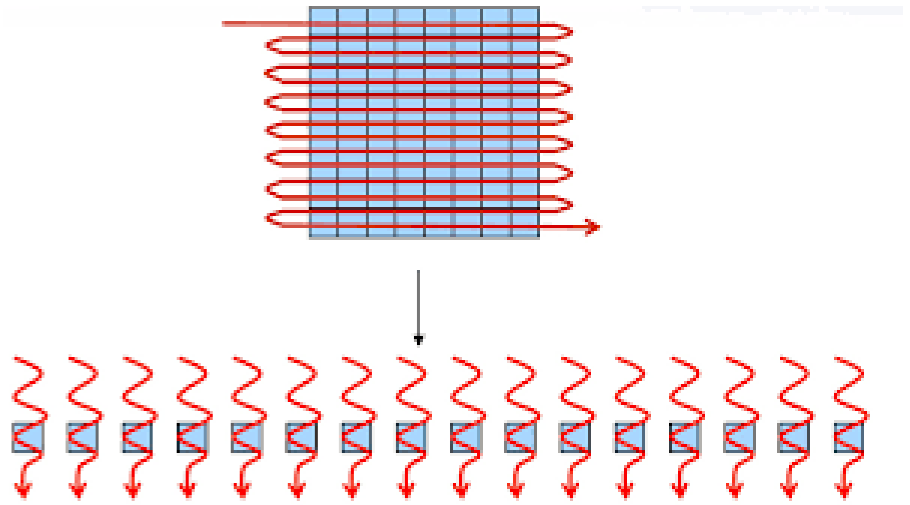
More about CUDA…

- CUDA is a set of developing tools to create applications that will be executed on a GPU (Graphics Processing Unit).
- CUDA was developed by NVidia and can only run on NVidia GPUs of G8x series and up.
- CUDA was released on February 15, 2007 for PC and Beta version for MacOS X on August 19, 2008.

Heterogeneous Computing

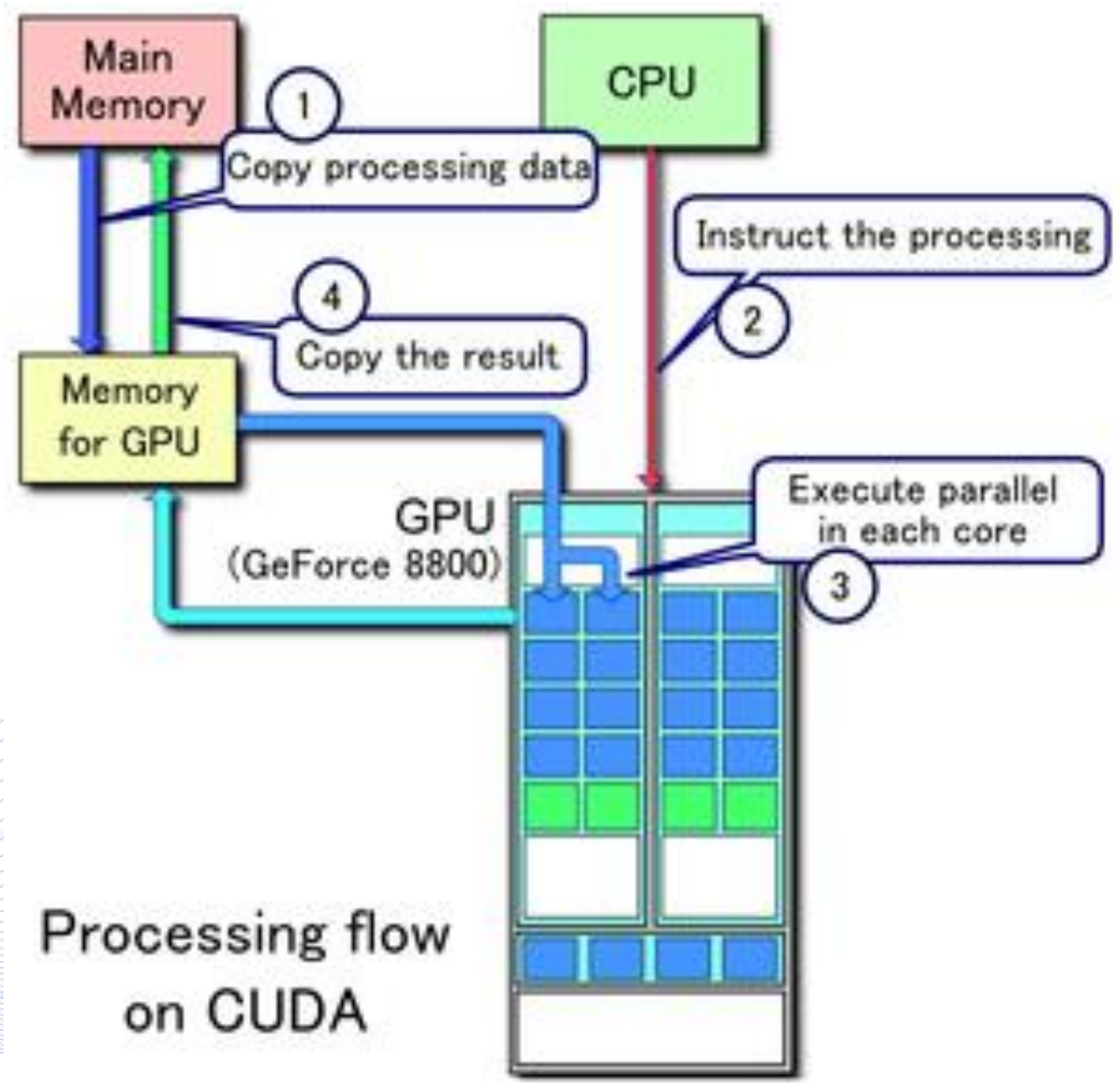
- Heterogeneous computing refers to systems which use more than one kind of processor or cores to maximize performance.
- In CUDA terminology:
  - **Host** is the **CPU**
  - **Device** is the **GPU**

Stream Computing



Serial processing of the data set using **for loop**

Data set decomposed into a *stream of elements.* ***Thread*** *is the execution of the same function (****kernel****) on each data element.*


Main Memory
CPU
1
Copy processing data
Instruct the processing
2
4
Copy the result
Memory for GPU
GPU (GeForce 8800)
Execute parallel in each core
3
Processing flow on CUDA

Program Structure of CUDA

- A typical CUDA program has code intended both for the GPU and the CPU.
- A traditional C program is a CUDA program with only the host code.
- The **host code** can be compiled by a traditional C compiler as the **GCC**
- The **device code** needs a special compiler to understand the api functions that are used. For Nvidia GPUs, the compiler is called the **NVCC** (Nvidia C Compiler).

Program Structure of CUDA

- The **device** code runs on the **GPU** while the **host** code runs on the **CPU**.
- The **NVCC** processes a CUDA program, and separates the host code from the device code (special CUDA keywords are looked for)
- The code intended to run on the GPU is marked with **special CUDA keywords** for labeling **data-parallel functions**, called ‘**Kernels**’. Kernels are decomposed to run in parallel on the multiple **GPU cores**.
- The device code is further compiled by the NVCC and executed on the GPU.

Program Structure of CUDA

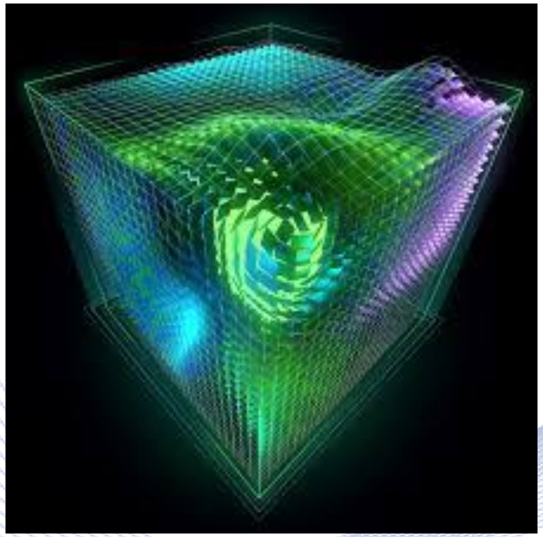
- Source files must have extension **.cu**
- CUDA is using kernel functions. These functions are executed on the GPU **simultaneously** by many **threads** in parallel.
- CUDA provides several extensions to the C-language. **__global__** declares a kernel function that will be executed on CUDA device. Return type for all these functions is **void**. These functions are user defined.
- When a **kernel** function is **called**, configuration values are provided for that function. Those values are included within "**<<<**" and "**>>>**" (triple angle or chevron brackets)

Why threads and blocks

Threads have mechanisms to:

- **Communicate**
- **Synchronize**

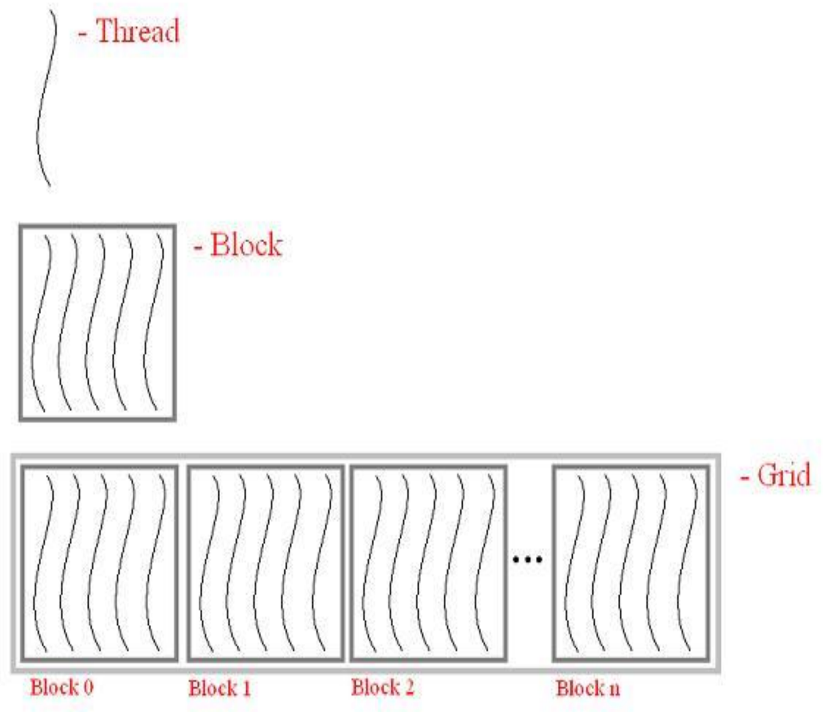
Blocks do not have the same mechanisms



Artificial Intelligence & Information Analysis Lab

Thread Hierarchy

- **Thread** – Distributed by the CUDA runtime (identified by threadIdx)
- **Warp** – A scheduling unit of up to 32 threads
- **Block** – A user defined group of 1 to 512 threads (identiffied by blockIdx)
- **Grid** – A group of one or more blocks. A grid is created for each CUDA kernel function



Artificial Intelligence & Information Analysis Lab

CUDA Warp

- CUDA utilizes **SIMT** (Single Instruction Multiple Thread)
- **Warps** are groups of **32** threads. Each warp receives a single instruction and “broadcasts” it to all of its threads.
- Because a warp receives a single instruction, it will diverge and converge as each thread branches independently

Artificial Intelligence & Information Analysis Lab

CUDA Built-In Variables

- **dim3 gridDim;**
  - –Dimensions of the grid in blocks (**gridDim.z unused)**
    Number of blocks in grid = gridDim.x * gridDim.y
- **dim3 blockDim;**
  - –Dimensions of the block in threads
    Number of threads in a block = blockDim.x * blockDim.y * blockDim.z
- **dim3 blockIdx;**
  - –Block index within the grid
- **dim3 threadIdx;**
  - –Thread index within the block

Artificial Intelligence & Information Analysis Lab

Example Initializing Values

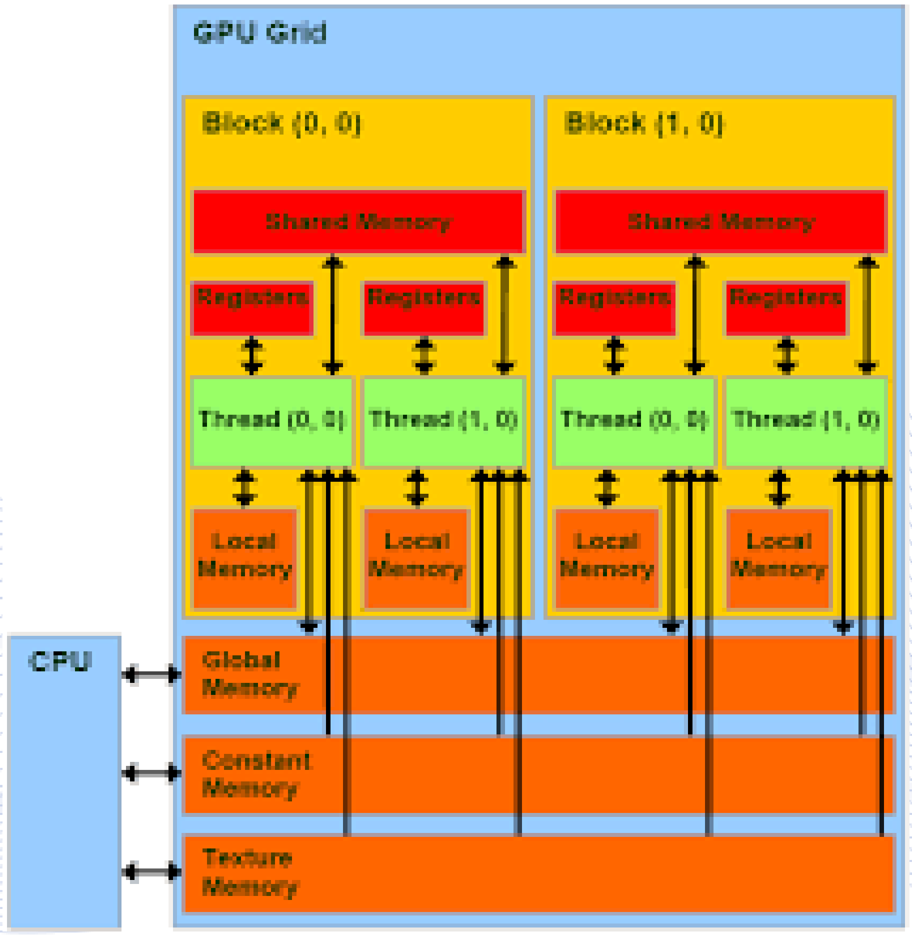
- To set dimensions:

```
dim3 grid(16,16); // grid = 16 x 16 blocks
dim3 block(32,32); // block = 32 x 32 threads
myKernel<<<grid, block>>>(...);
```

- which sets:
  - grid.x = 16;
  - grid.y = 16;
  - block.x = 32;
  - block.y = 32;
  - block.z = 1;

Artificial Intelligence & Information Analysis Lab

Cuda Memories



Device Global Memory and Data Transfer

- A typical **GPU** comes with its own **global** memory (**DRAM**). This is called the **device** memory.
- To execute a kernel on the GPU, the programmer needs to allocate separate memory on the GPU by writing code.
- After **allocating memory** on the **device**, data has to be **transferred** from the **host** memory to the **device** memory.
- After the **kernel** is **executed** on the device, the **result** has to be **transferred back** from the **device** memory to the **host** memory.
- The allocated memory on the device has to be **freed-up**.

CUDA Example Program - Addition on the Device

A simple kernel to add two integer tables:

```
__global__ void addKernel(int *c, int *a, int *b)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}
```

__global__ is a CUDA C/C++ keyword meaning
addKernel() will execute on the device
addKernel() will be called from the host

GPU memory allocation

// Allocate GPU buffers for three vectors (two input, one output) .

```
cudaStatus = cudaMalloc((void**)&dev_c, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }
    cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }
    cudaStatus = cudaMalloc((void**)&dev_b, size * sizeof(int));
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        goto Error;
    }
```

Freeing device memory

```
cudaFree(dev_c);
cudaFree(dev_a);
cudaFree(dev_b);
```

Parallel vector addition (1)

- We can perform parallel vector addition using:
  - Many blocks with one thread each

```
addKernel <<<size, 1 >>> (dev_c, dev_a, dev_b);
```

  However, this is NOT a good idea at all.

  - One block with many threads

```
addKernel <<<1, size >>> (dev_c, dev_a, dev_b);
```

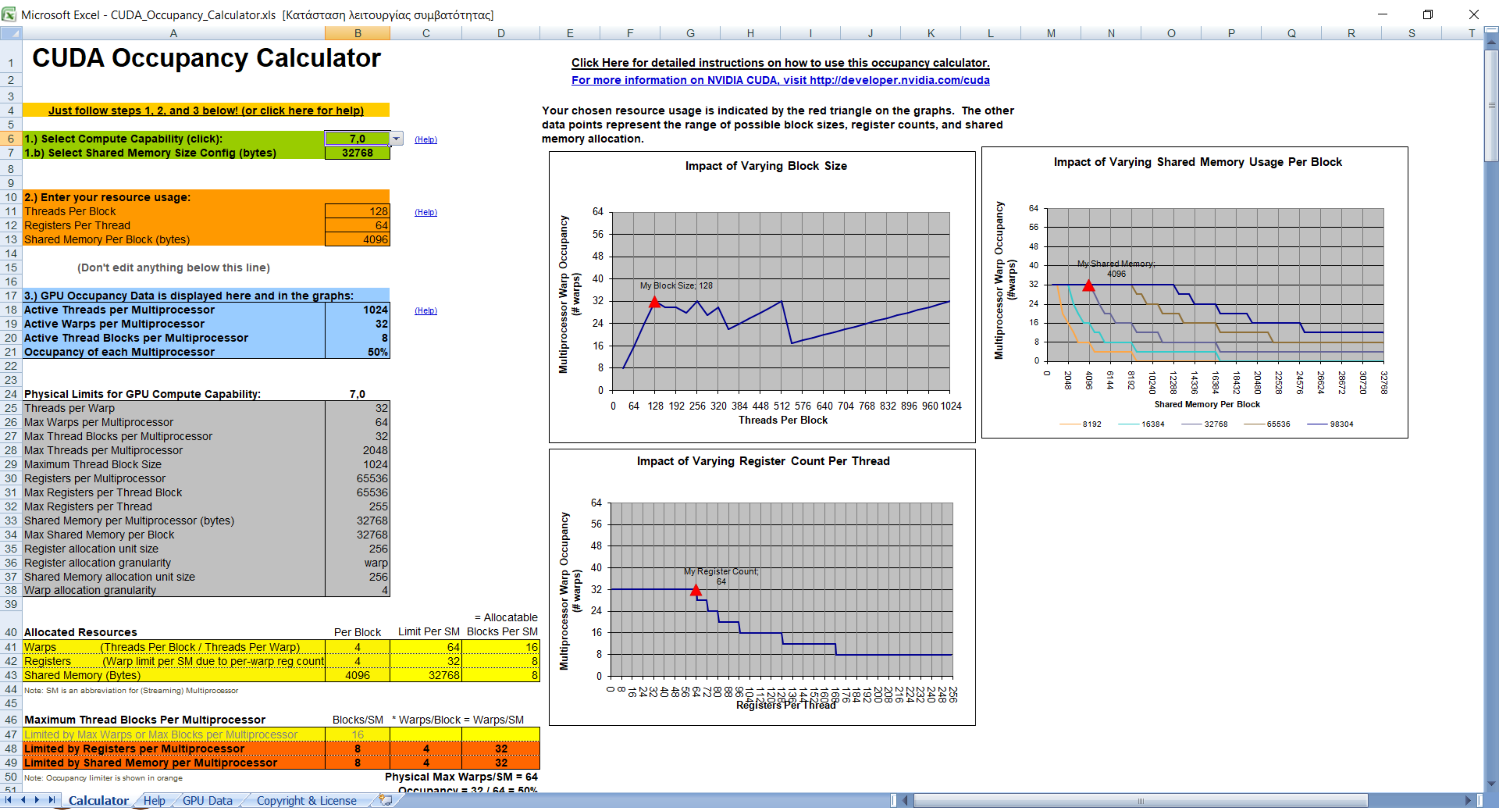
  Use of only 1 block, i.e. only 1 SM on the GPU results to very poor performance. In practice, we need to use multiple blocks to utilize all SMs.

Parallel vector addition (2)

- We can perform parallel vector addition using:

```
dim3 blocksPerGrid(N/256,1,1); //assuming 256 divides N exactly
dim3 threadsPerBlock(256,1,1);
addKernel<<<blocksPerGrid, threadsPerBlock>>>(dev_c, dev_a, dev_b);
```

- We have chosen to use 256 threads per block, which is typically a good number (multiple of 32)



Microsoft Excel - CUDA_Occupancy_Calculator.xls [Κατάσταση λειτουργίας συμβατότητας]
Impact of Varying Block Size
Multiprocessor Warp Occupancy (# warps)
Threads Per Block
My Block Size; 128
Impact of Varying Shared Memory Usage Per Block
Multiprocessor Warp Occupancy (#warps)
Shared Memory Per Block
My Shared Memory; 4096
8192
16384
32768
65536
98304
Impact of Varying Register Count Per Thread
Multiprocessor Warp Occupancy (# warps)
Registers Per Thread
My Register Count; 64
Calculator
Help
GPU Data
Copyright & License

CUDA Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Just follow steps 1, 2, and 3 below! (or click here for help)

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

1.) Select Compute Capability (click):	7,0	(Help)
1.b) Select Shared Memory Size Config (bytes)	32768	

2.) Enter your resource usage:		
Threads Per Block	128	(Help)
Registers Per Thread	64	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1024	(Help)
Active Warps per Multiprocessor	32	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	50%	

Physical Limits for GPU Compute Capability:	7,0
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	32768
Max Shared Memory per Block	32768
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	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	4	64	16
Registers (Warp limit per SM due to per-warp reg count)	4	32	8
Shared Memory (Bytes)	4096	32768	8

Note: SM is an abbreviation for (Streaming) Multiprocessor

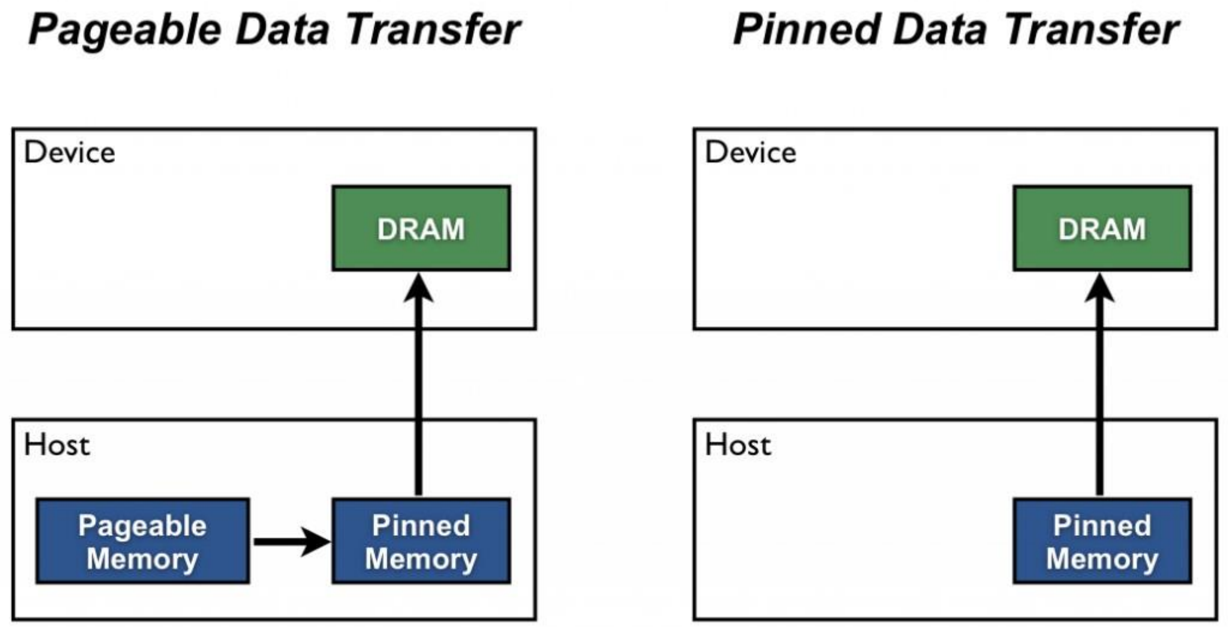
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	16		
Limited by Registers per Multiprocessor	8	4	32
Limited by Shared Memory per Multiprocessor	8	4	32

Note: Occupancy limiter is shown in orange

Physical Max Warps/SM = 64
Occupancy = 32 / 64 = 50%

GPU optimization guide (3)

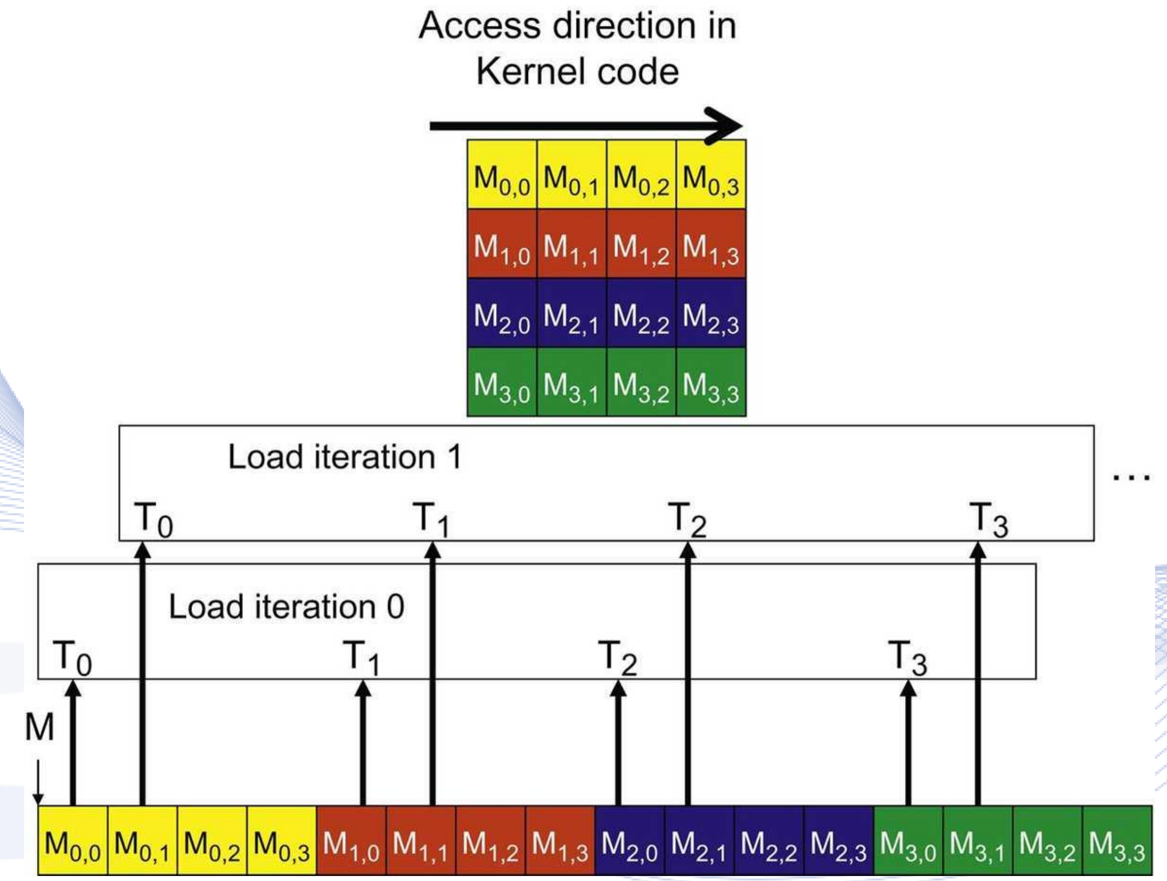
- Host (CPU) data allocations are **pageable** by default. The GPU cannot access data directly from pageable host memory, so when a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or "**pinned**", host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory, as illustrated below.
- Allocate pinned host memory in CUDA C/C++ using cudaMallocHost() or cudaHostAlloc(), and deallocate it with cudaFreeHost()



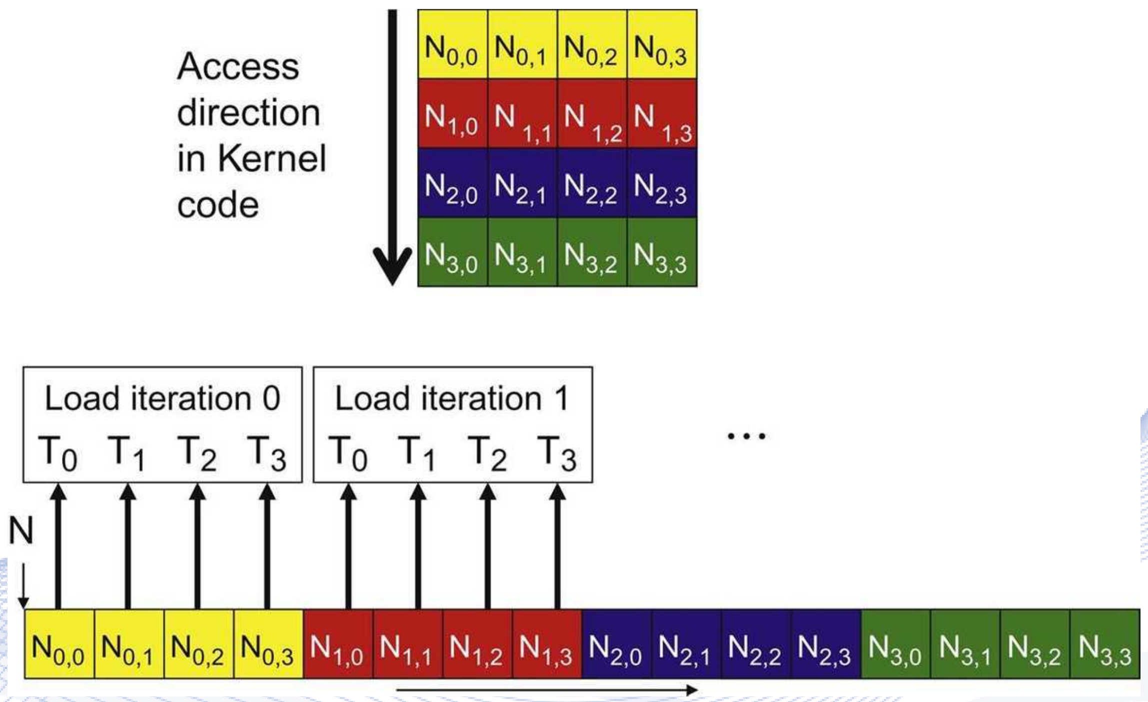
https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/

GPU optimization guide (4)

Write the code so that **consecutive threads** access **consecutive memory locations** (memory **coalescing**)



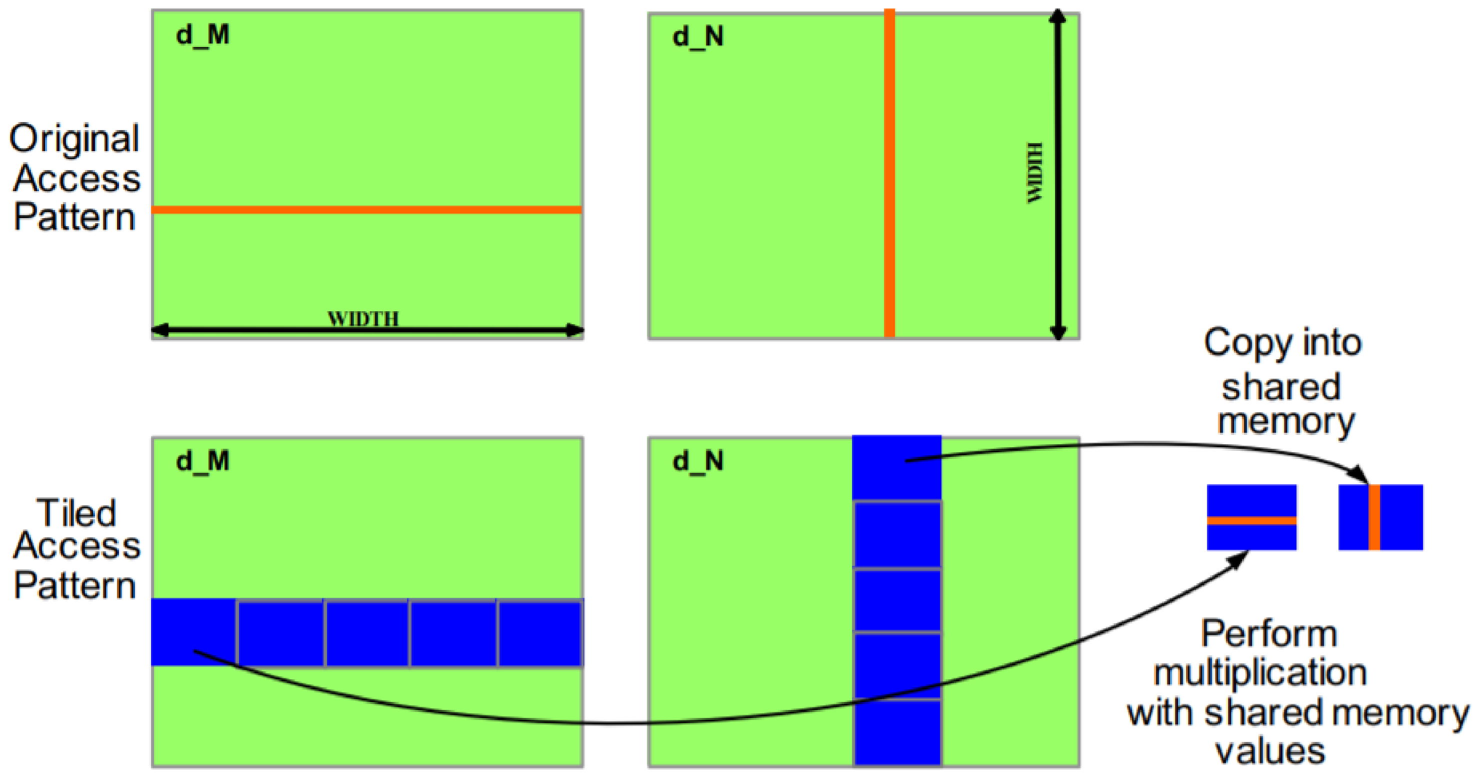
Access direction in Kernel code
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$
Load iteration 1
T_0 T_1 T_2 T_3
...
Load iteration 0
T_0 T_1 T_2 T_3
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$ $M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$



Access direction in Kernel code
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$
$N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$
$N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$
$N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$
Load iteration 0
T_0 T_1 T_2 T_3
Load iteration 1
T_0 T_1 T_2 T_3
...
N
$N_{0,0}$ $N_{0,1}$ $N_{0,2}$ $N_{0,3}$ $N_{1,0}$ $N_{1,1}$ $N_{1,2}$ $N_{1,3}$ $N_{2,0}$ $N_{2,1}$ $N_{2,2}$ $N_{2,3}$ $N_{3,0}$ $N_{3,1}$ $N_{3,2}$ $N_{3,3}$

GPU optimization guide (5.2)

Corner Turning: from global to shared memory



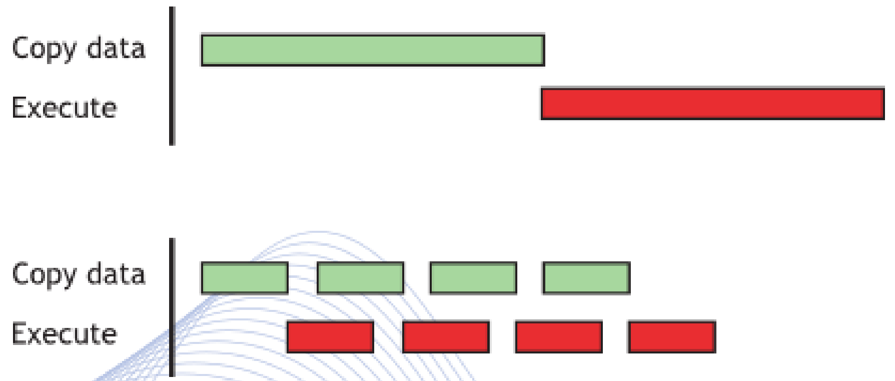
GPU optimization guide (6)

- **Sequential copy and execute**

```
cudaMemcpy(a_d, a_h, N*sizeof(float), dir);
kernel<<<N/nThreads, nThreads>>>(a_d);
```

- **Staged concurrent copy and execute**

```
size=N*sizeof(float)/nStreams;
for (i=0; i<nStreams; i++) {
    offset = i*N/nStreams;
    cudaMemcpyAsync(a_d+offset, a_h+offset, size, dir, stream[i]);
    kernel<<<N/(nThreads*nStreams), nThreads, 0, stream[i]>>>(a_d+offset); }
```



***Stream** is a sequence of operations that execute in issue-order on the GPU.*

*CUDA operations in **different** streams may run **concurrently**.*

*CUDA operations from **different** streams may be **interleaved***

CUDA Convolution implemented samples

- Convolution FFT2D
- Convolution Separable

Implemented samples installed with CUDA in ProgramData\NVIDIA Corporation\CUDA Samples\

Bibliography

- nvidia.com
- Supercomputing for the Masses by Peter Zalutski
- Basic Concepts in GPU Computing by Mao Gao
- Parallel Programming with CUDA Matthew Guidry Charles McClendon
- https://www.tutorialspoint.com/cuda/
- https://www3.nd.edu/
- http://users.wfu.edu
- http://www.bsc.es/
- https://devblogs.nvidia.com/nvidia-turing-architecture-in-depth/
- https://www.epcc.edu
- https://www.micc.unifi.it/
- https://jhui.github.io/

Q & A

Thank you very much for your attention!

More material in
http://icarus.csd.auth.gr/cvml-web-lecture-series/

Contact: Prof. I. Pitas
pitas@csd.auth.gr