CUDA: Introduction

Christian Trefftz / Greg Wolffe
Grand Valley State University

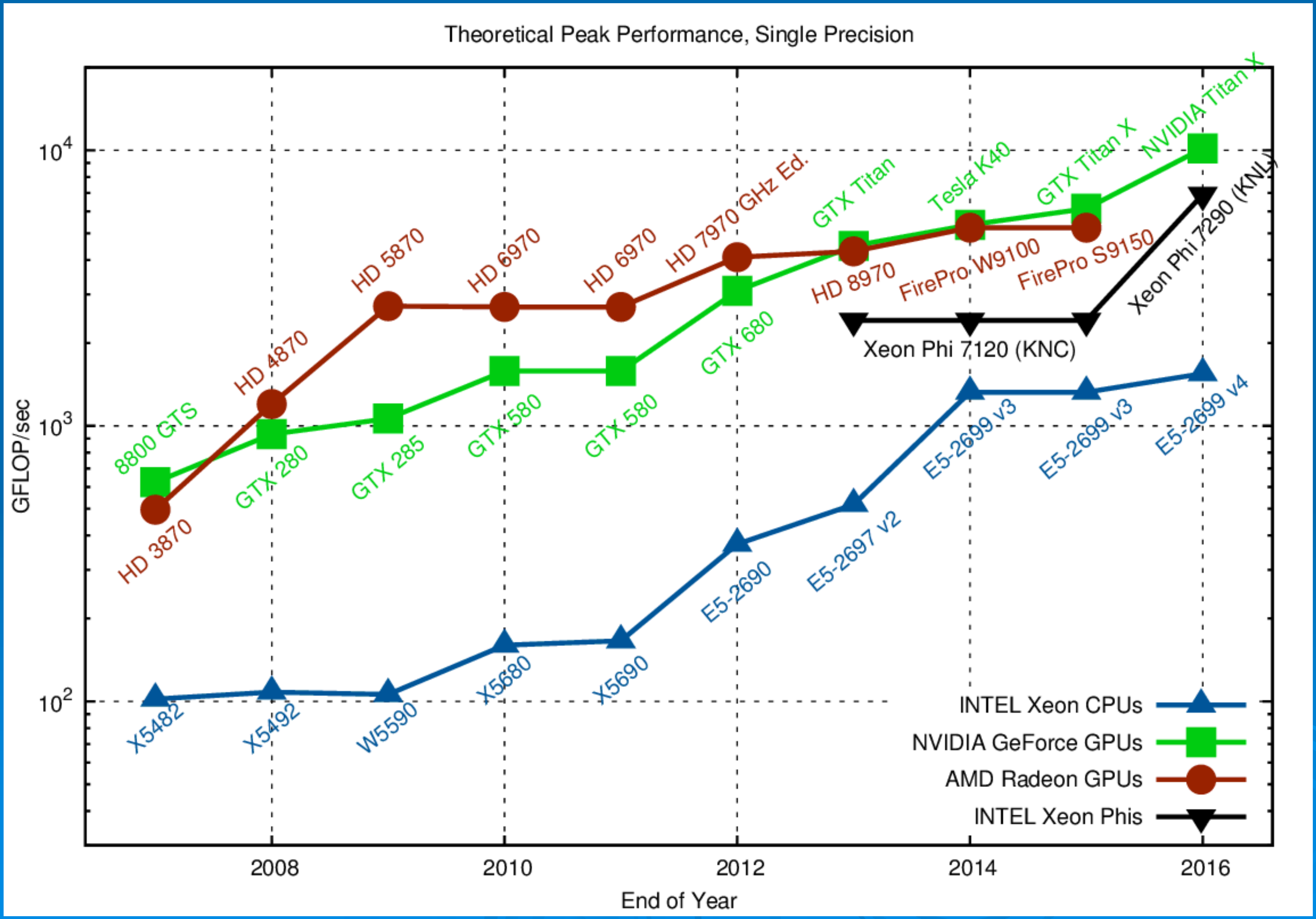
Supercomputing 2008
Education Program

(modifications by Jernej Barbic, 2008-2019)

Terms

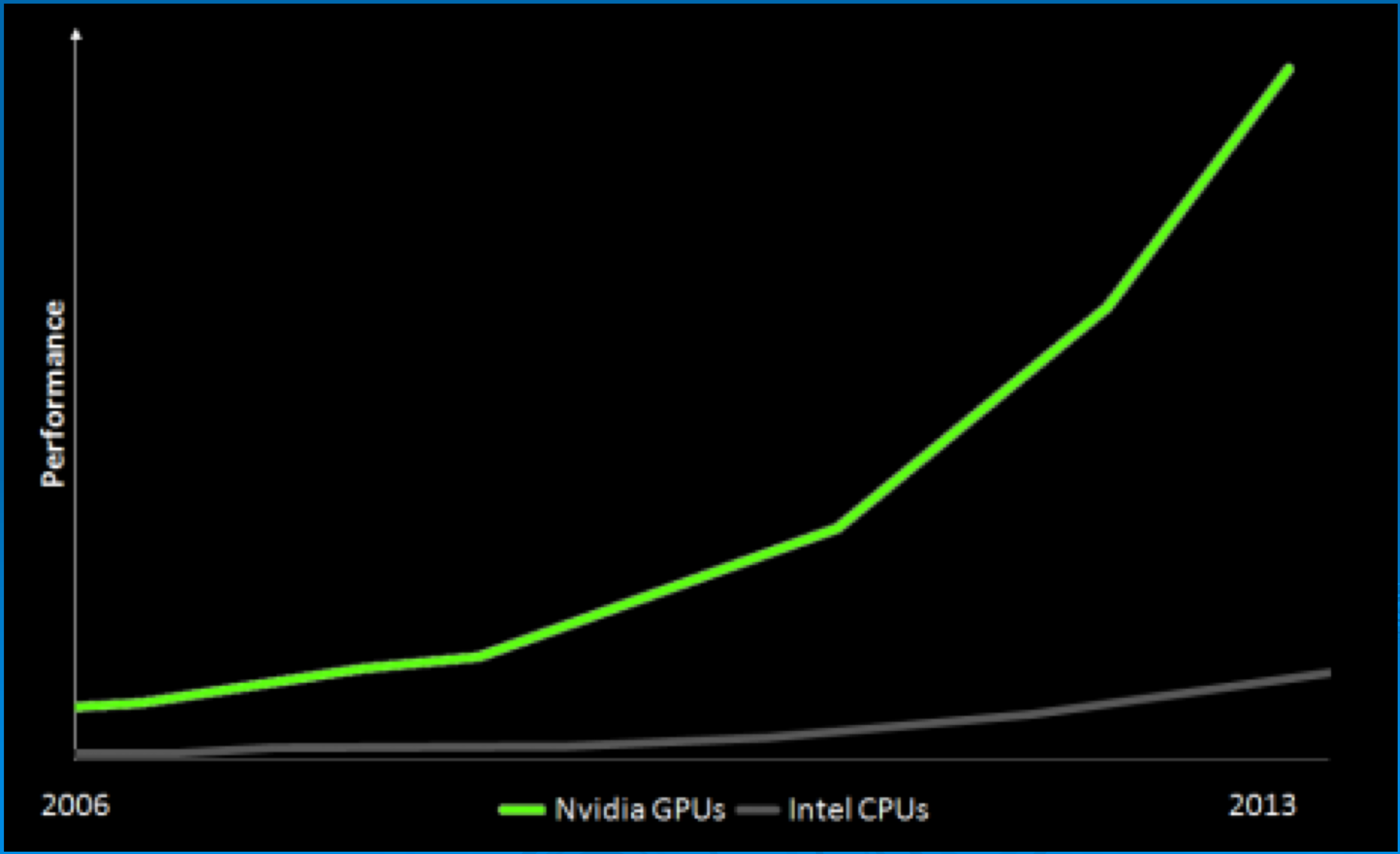
- What is GPGPU?
  - General-Purpose computing on a Graphics Processing Unit
  - Using graphic hardware for non-graphic computations
- What is CUDA?
  - Parallel computing platform and API by Nvidia
  - Compute Unified Device Architecture
  - Software architecture for managing data-parallel programming
  - Introduced in 2007; still actively updated

Motivation

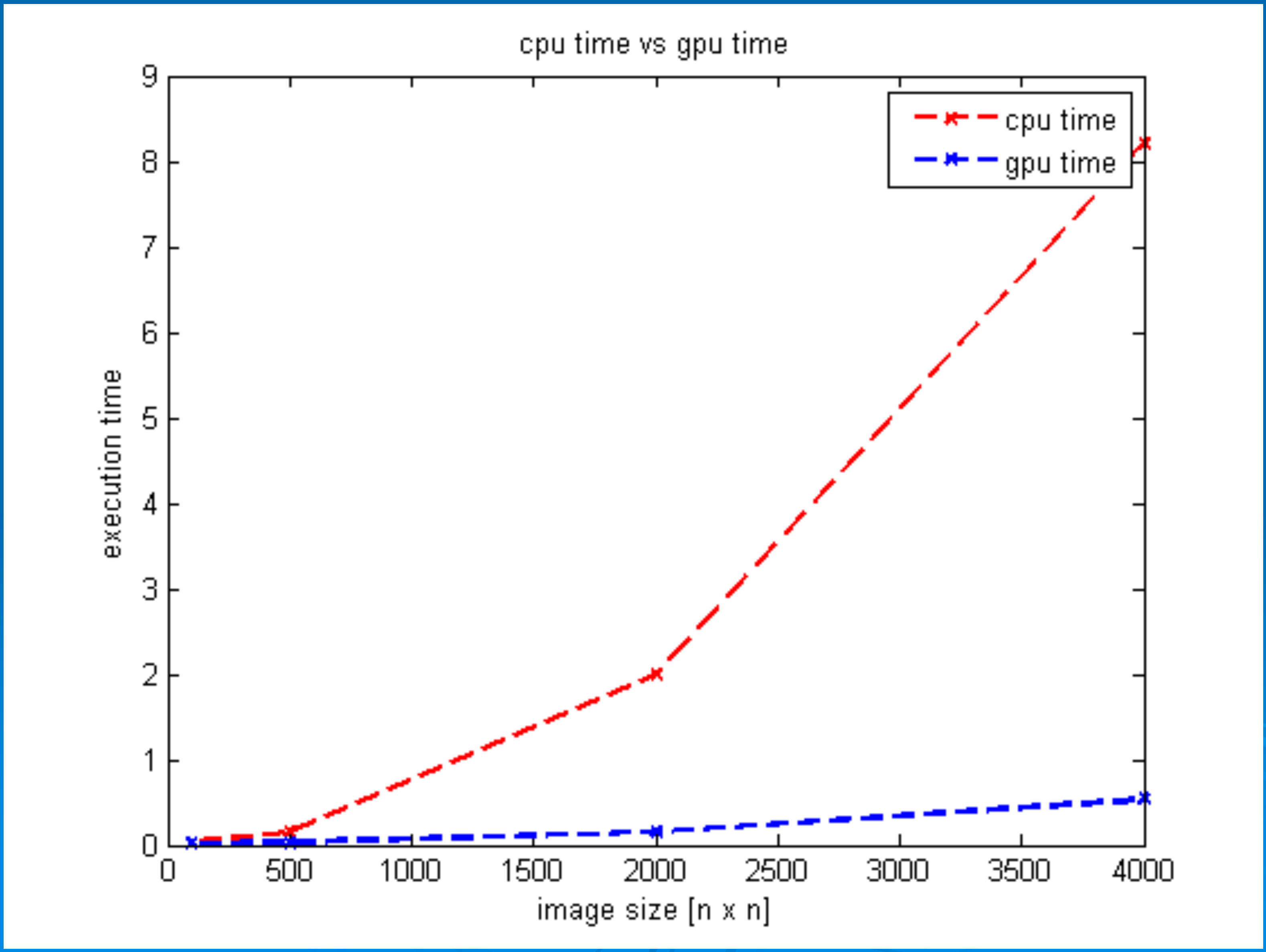


Theoretical Peak Performance, Single Precision
GFLOP/sec
End of Year
2008
2010
2012
2014
2016
10^4
10^3
10^2
8800 GTS
GTX 280
GTX 285
GTX 580
GTX 580
GTX 680
GTX Titan
Tesla K40
GTX Titan X
NVIDIA Titan X
HD 3870
HD 4870
HD 5870
HD 6970
HD 6970
HD 7970 GHz Ed.
HD 8970
FirePro W9100
FirePro S9150
Xeon Phi 7120 (KNC)
Xeon Phi 7290 (KNL)
X5482
X5492
W5590
X5680
X5690
E5-2690
E5-2697 v2
E5-2699 v3
E5-2699 v3
E5-2699 v4
INTEL Xeon CPUs
NVIDIA GeForce GPUs
AMD Radeon GPUs
INTEL Xeon Phis

Motivation



Motivation



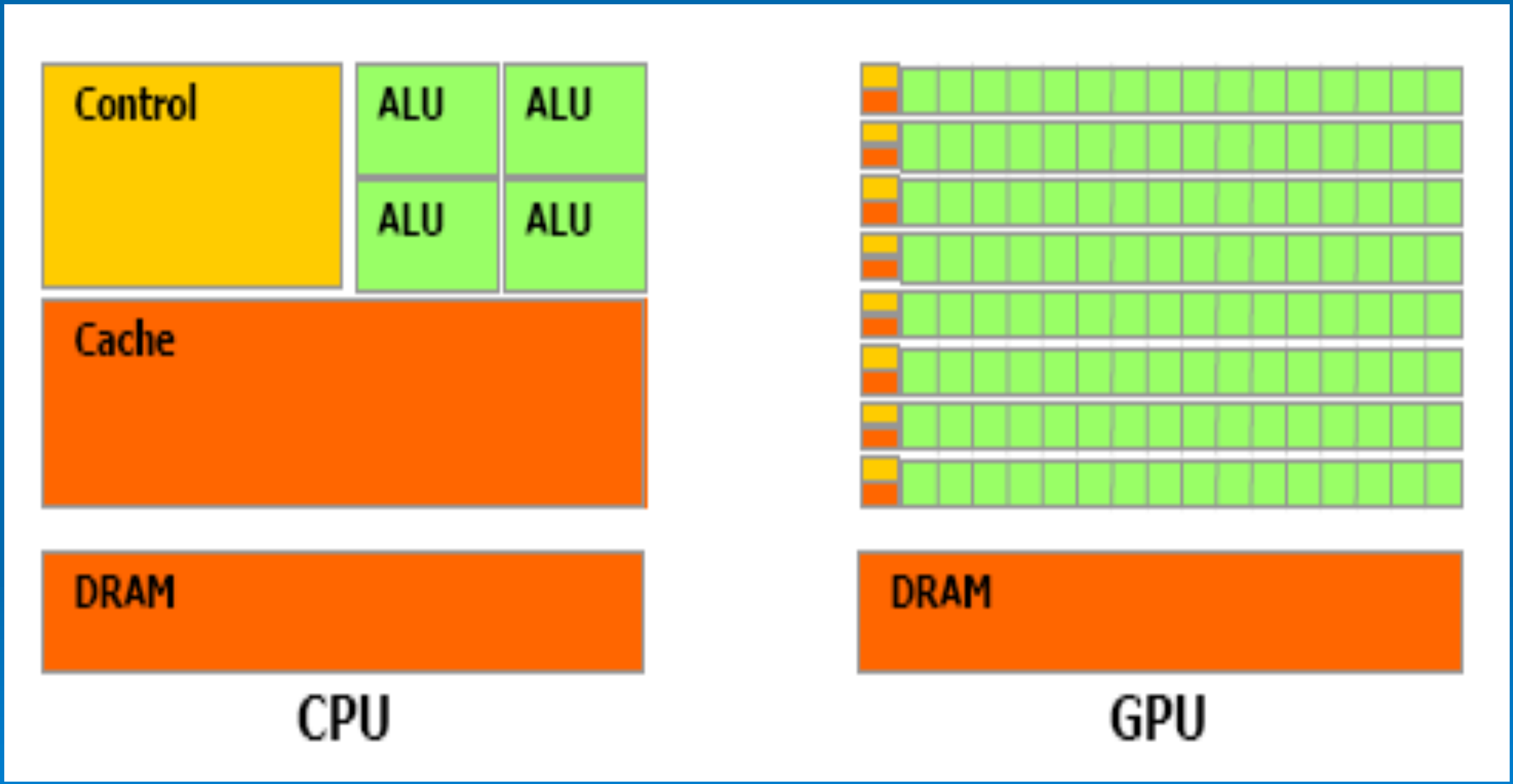
cpu time vs gpu time
cpu time
gpu time
execution time
image size [n x n]
0
1
2
3
4
5
6
7
8
9
0
500
1000
1500
2000
2500
3000
3500
4000

CPU vs. GPU

- CPU
  - Fast caches
  - Branching adaptability
  - High performance
- GPU
  - Multiple ALUs
  - Fast onboard memory
  - High throughput on parallel tasks
    - Executes program on each fragment/vertex

- CPUs are great for ***task*** parallelism
- GPUs are great for ***data*** parallelism

CPU vs. GPU - Hardware



- More transistors devoted to data processing

Traditional Graphics Pipeline

Vertex processing

⇩

Rasterizer

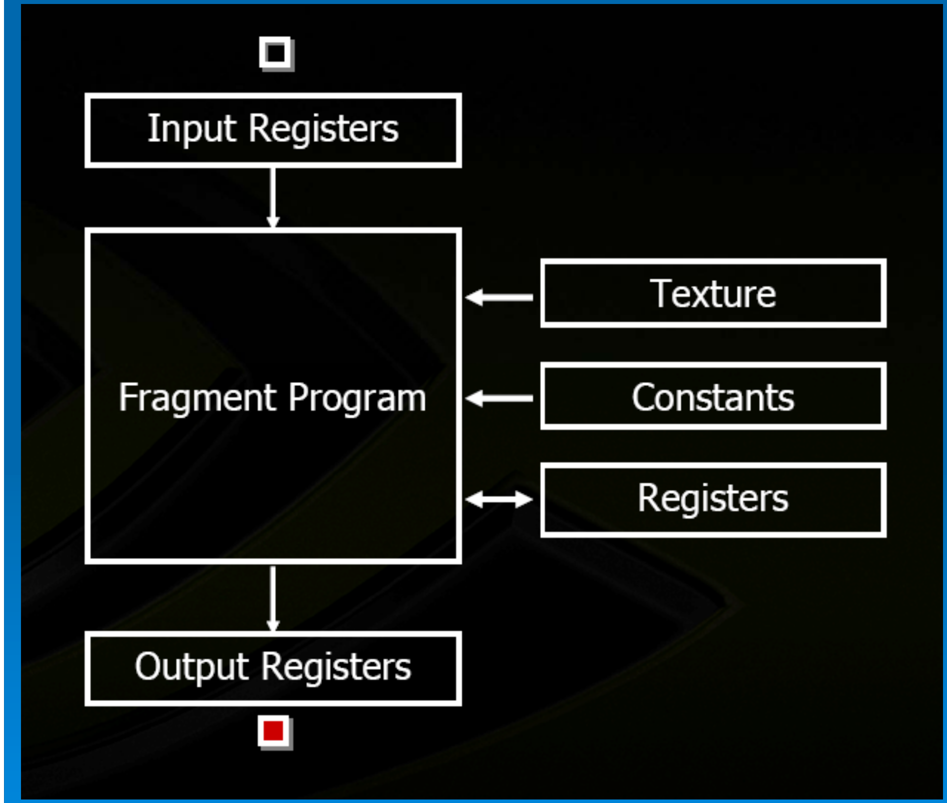
⇩

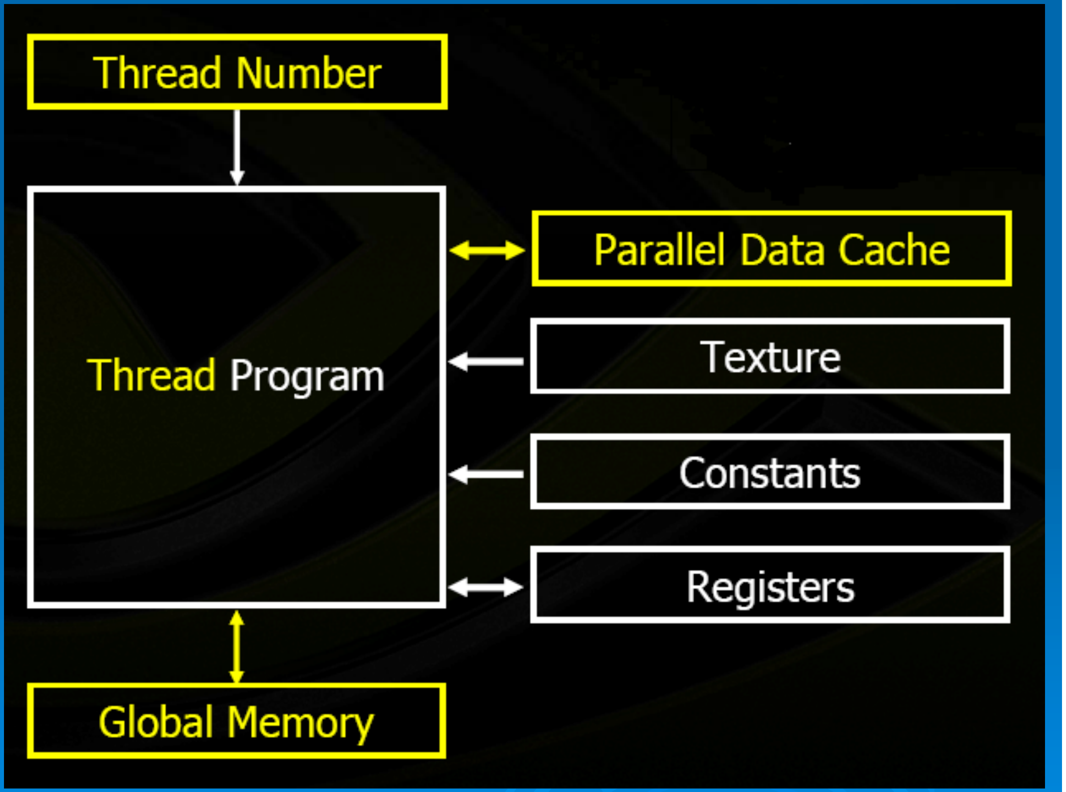
Fragment processing

⇩

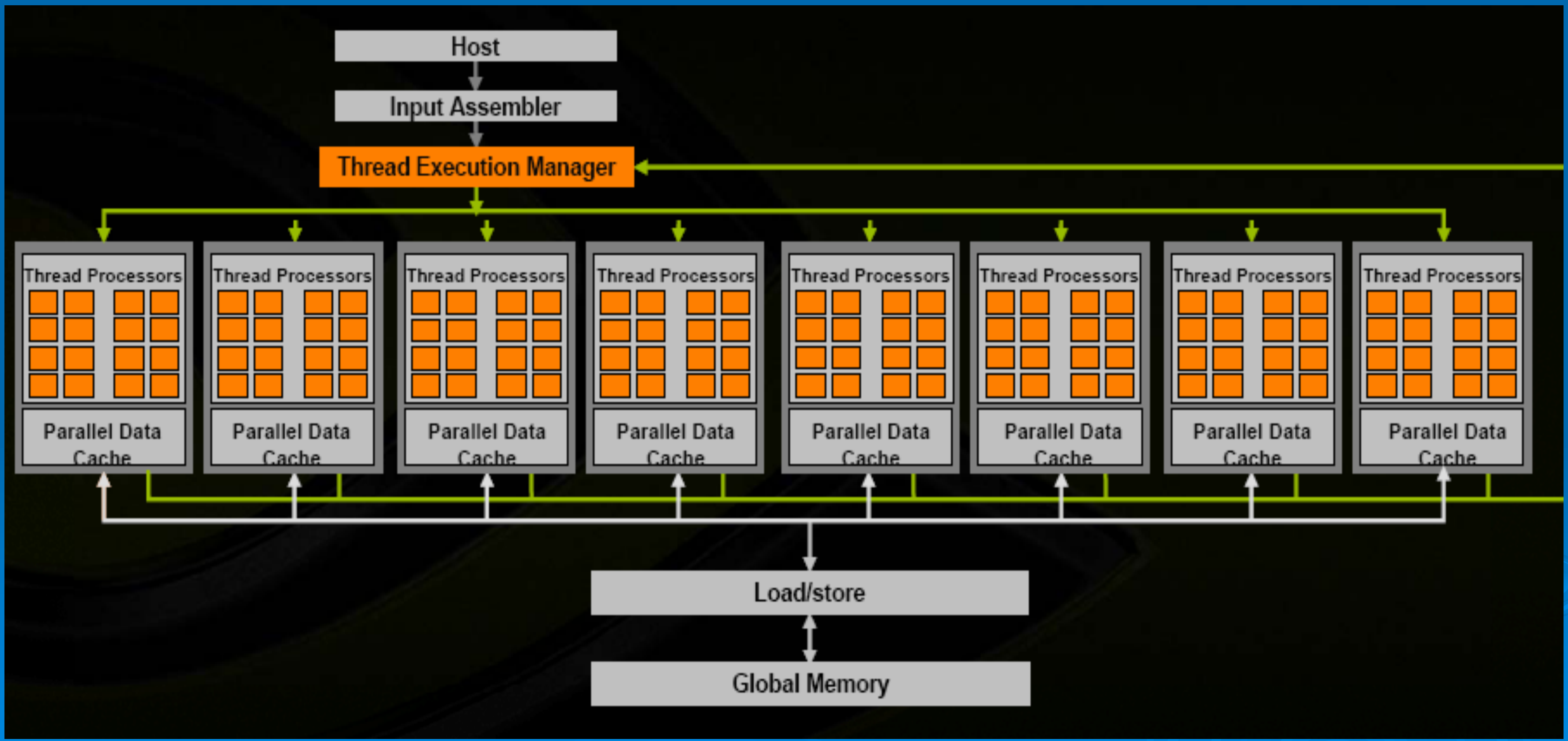
Renderer (textures)

Pixel / Thread Processing

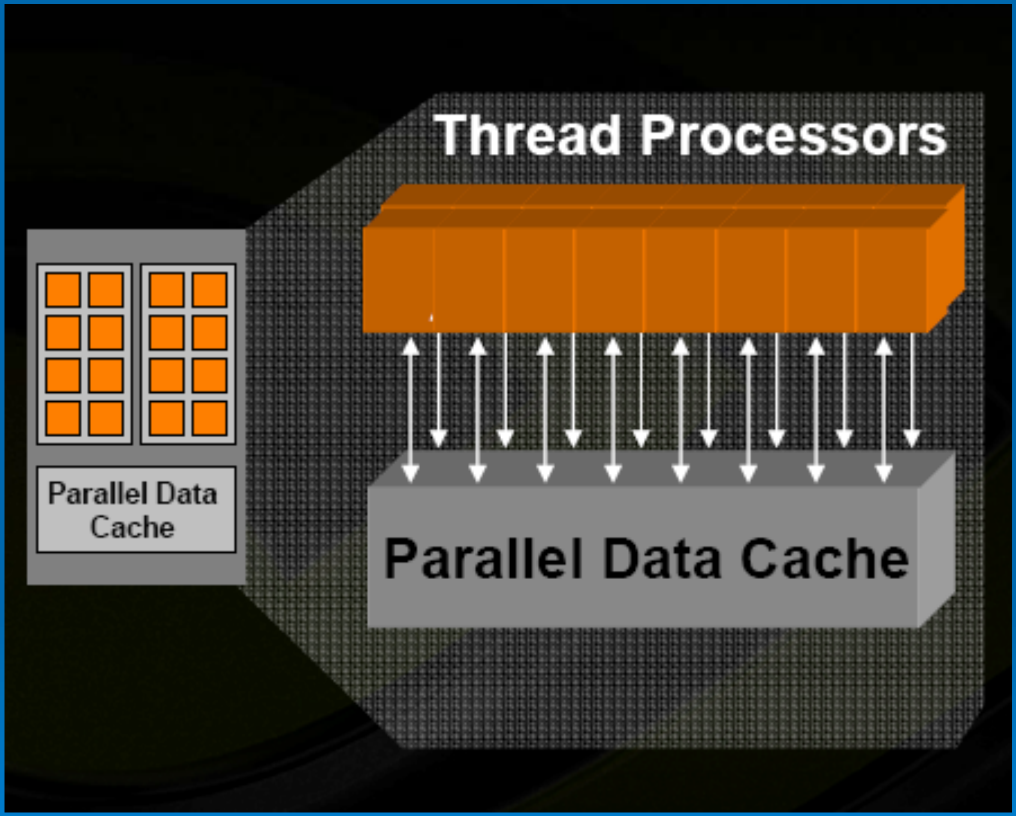




GPU Architecture



Processing Element

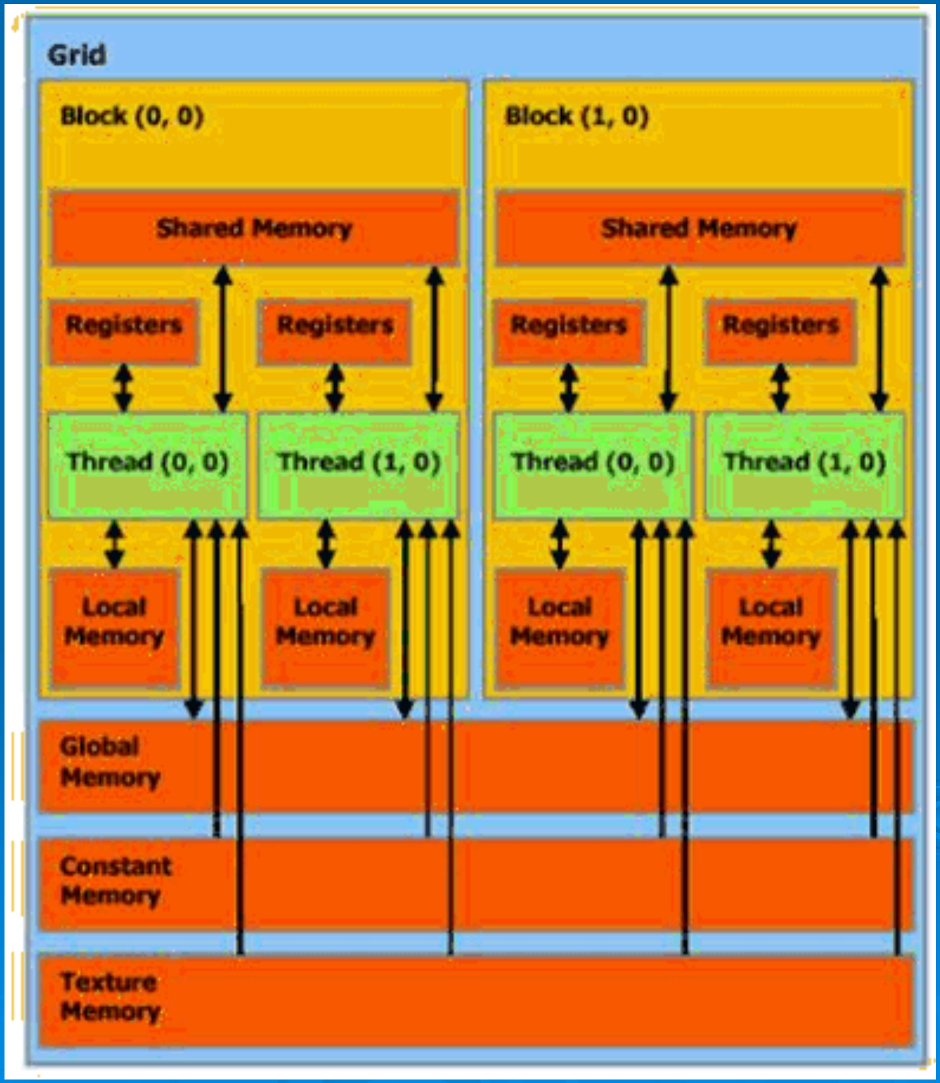


- Processing element = thread processor

GPU Memory Architecture

Uncached:
- Registers
- Shared Memory
- Local Memory
- Global Memory

Cached:
- Constant Memory
- Texture Memory



Data-parallel Programming

- Think of the GPU as a massively-threaded co-processor
- Write “kernel” functions that execute on the device -- processing multiple data elements in parallel

- Keep it busy! ⇨ massive threading
- Keep your data close! ⇨ local memory

Hardware Requirements

- CUDA-capable video card
- Power supply
- Cooling
- PCI-Express

A Gentle Introduction to CUDA Programming

Credits

- The code used in this presentation is based on code available in:
  - the Tutorial on CUDA in Dr. Dobbs Journal
  - Andrew Bellenir's code for matrix multiplication
  - Igor Majdandzic's code for Voronoi diagrams
  - NVIDIA's CUDA programming guide

Software Requirements/Tools

- CUDA device driver
- CUDA Toolkit (compiler, CUBLAS, CUFFT)
- CUDA Software Development Kit
  - Emulator

Profiling:

- Occupancy calculator
- Visual profiler

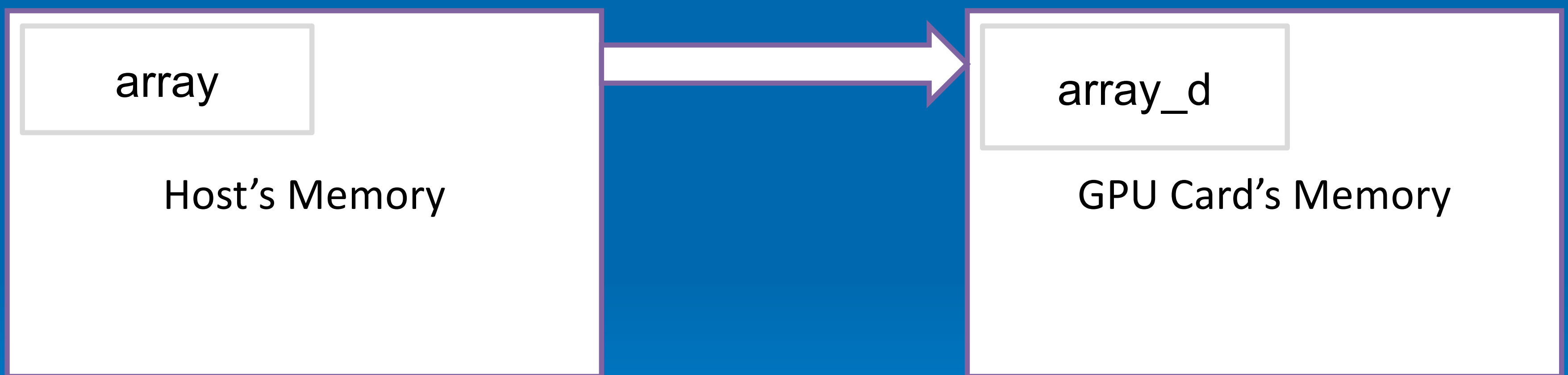
To compute, we need to:

- Allocate memory for the computation on the GPU (incl. variables)
- Provide input data
- Specify the computation to be performed
- Read the results from the GPU (output)

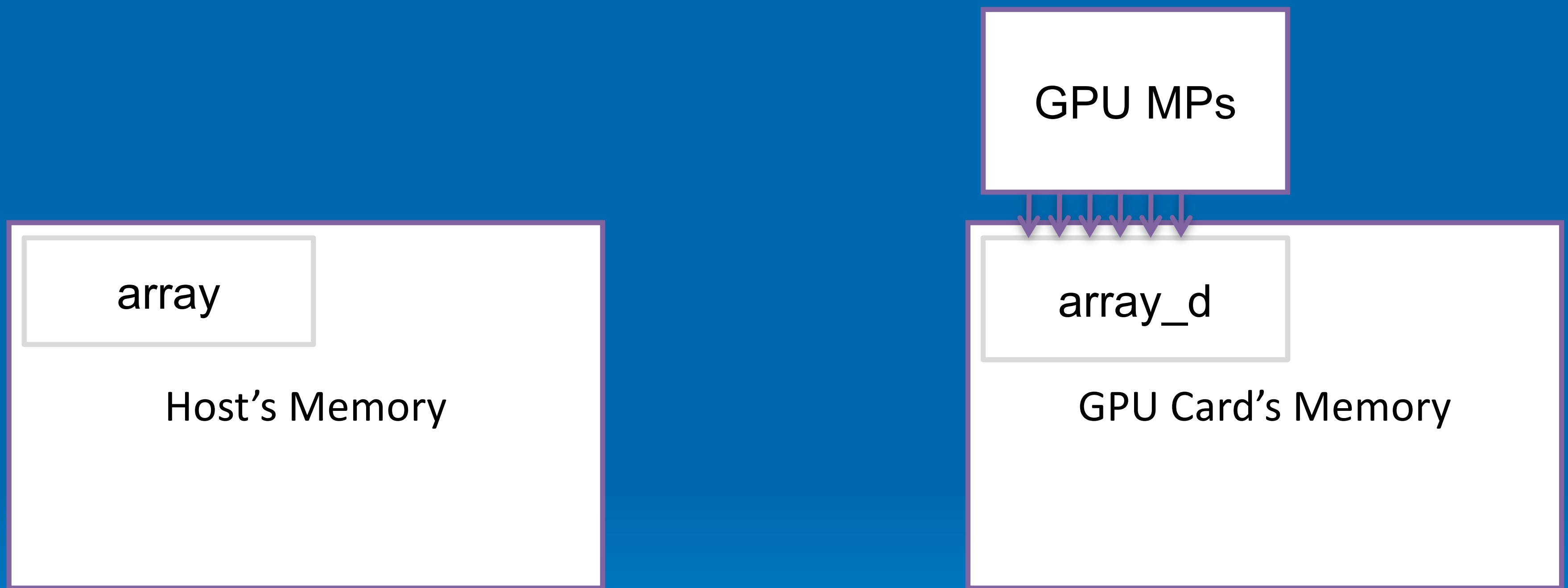
Initially:

Allocate Memory in the GPU card

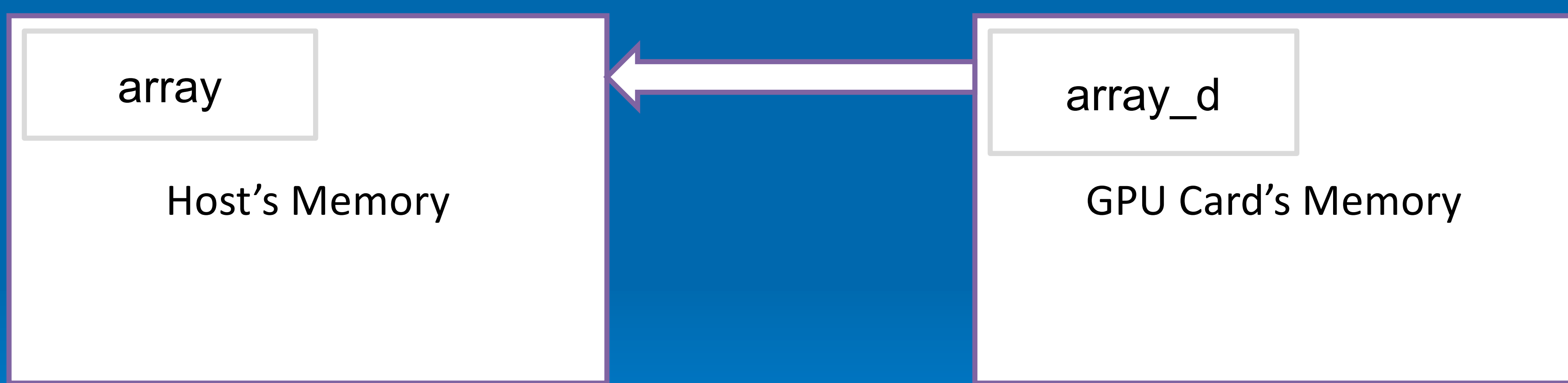
Copy content from the host's memory to the GPU card memory



Execute code on the GPU



Copy results back to the host memory



The Kernel

- The code to be executed in the stream processors on the GPU
- Simultaneous execution in several (perhaps all) stream processors on the GPU
- How is every instance of the kernel going to know which piece of data it is working on?

Grid and Block Size

- Grid size: The number of blocks
  - Can be 1 or 2-dimensional array of blocks

- Each block is divided into threads
  - Can be 1, 2, or 3-dimensional array of threads

Let's look at a very simple example

- The code has been divided into two files:
  - simple.c
  - simple.cu
- simple.c is ordinary code in C
- It allocates an array of integers, initializes it to values corresponding to the indices in the array and prints the array.
- It calls a function that modifies the array
- The array is printed again.

simple.c

```
#include <stdio.h>
#define SIZEOFARRAY 64
extern void fillArray(int *a,int size);

/* The main program */
int main(int argc,char *argv[])
{
/* Declare the array that will be modified by the GPU */
 int a[SIZEOFARRAY];
 int i;
/* Initialize the array to 0s */
 for(i=0;i < SIZEOFARRAY;i++) {
   a[i]=0;
 }
 /* Print the initial array */
 printf("Initial state of the array:\n");
for(i = 0;i < SIZEOFARRAY;i++) {
   printf("%d ",a[i]);
 }
 printf("\n");
/* Call the function that will in turn call the function in the GPU that will fill
the array */
 fillArray(a,SIZEOFARRAY);
 /* Now print the array after calling fillArray */
 printf("Final state of the array:\n");
 for(i = 0;i < SIZEOFARRAY;i++) {
   printf("%d ",a[i]);
 }
 printf("\n");
 return 0;
}
```

simple.cu

- simple.cu contains two functions
  - fillArray(): A function that will be executed on the host and which takes care of:
    - Allocating variables in the global GPU memory
    - Copying the array from the host to the GPU memory
    - Setting the grid and block sizes
    - Invoking the kernel that is executed on the GPU
    - Copying the values back to the host memory
    - Freeing the GPU memory

fillArray (part 1)

```
#define BLOCK_SIZE 32
extern "C" void fillArray(int *array, int arraySize)
{
   int * array_d;
   cudaError_t result;

   /* cudaMalloc allocates space in GPU memory */
   result =
   cudaMalloc((void**)&array_d,sizeof(int)*arraySize);

   /* copy the CPU array into the GPU array_d */
   result = cudaMemcpy(array_d,array,sizeof(int)*arraySize,
                              cudaMemcpyHostToDevice);
```

fillArray (part 2)

```
    /* Indicate block size */
    dim3 dimblock(BLOCK_SIZE);
    /* Indicate grid size */
    dim3 dimgrid(arraySize / BLOCK_SIZE);

    /* Call the kernel */
    cu_fillArray<<<dimgrid,dimblock>>>(array_d);

    /* Copy the results from GPU back to CPU memory */
    result =
    cudaMemcpy(array,array_d,sizeof(int)*arraySize,cudaMemcpyDevice
    ToHost);

    /* Release the GPU memory */
    cudaFree(array_d);
}
```

simple.cu (cont.)

- The other function in simple.cu is cu_fillArray():
  - This is the GPU kernel
  - Identified by the keyword: __global__
  - Built-in variables:
    - blockIdx.x : block index within the grid
    - threadIdx.x: thread index within the block

cu_fillArray

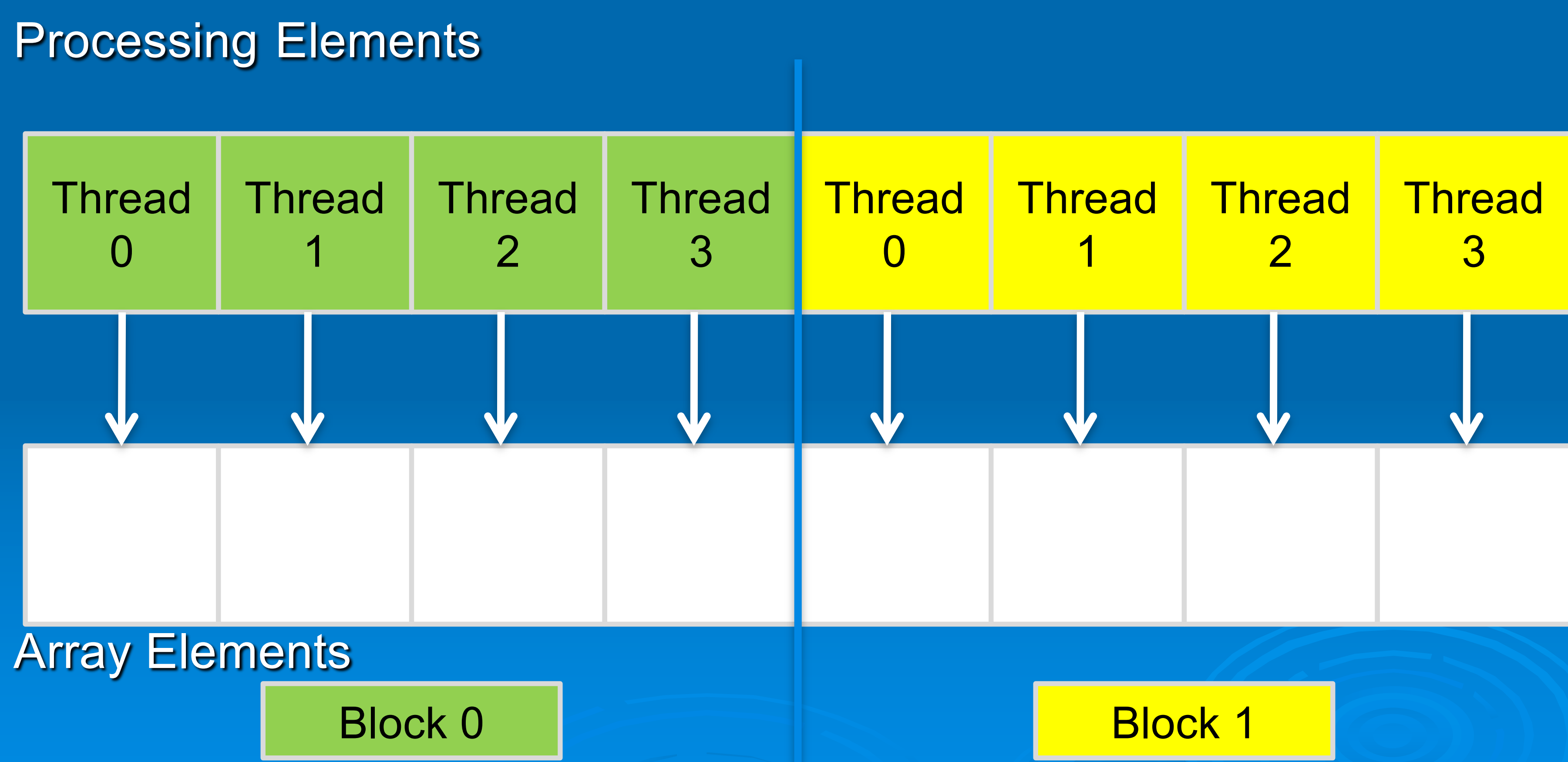
```
__global__ void cu_fillArray(int * array_d)
{
   int x;
   x = blockIdx.x * BLOCK_SIZE + threadIdx.x;
   array_d[x] = x;
}

__global__ void cu_addIntegers(int * array_d1, int * array_d2)
{
   int x;
   x = blockIdx.x * BLOCK_SIZE + threadIdx.x;
   array_d1[x] += array_d2[x];
}
```

To compile:

- nvcc simple.c simple.cu –o simple
- The compiler generates the code for both the host and the GPU
- Demo on cuda.littlefe.net …

In the GPU:



Processing Elements
Thread 0
Thread 1
Thread 2
Thread 3
Thread 0
Thread 1
Thread 2
Thread 3
Array Elements
Block 0
Block 1

Another Example: **saxpy**

- SAXPY (Scalar Alpha X Plus Y)
  - A common operation in linear algebra
- CUDA: loop iteration ⇨ thread

Traditional Sequential Code

```
void saxpy_serial(int n,
                  float alpha,
                  float *x,
                  float *y)
{
  for(int i = 0;i < n;i++)
     y[i] = alpha*x[i] + y[i];
}
```

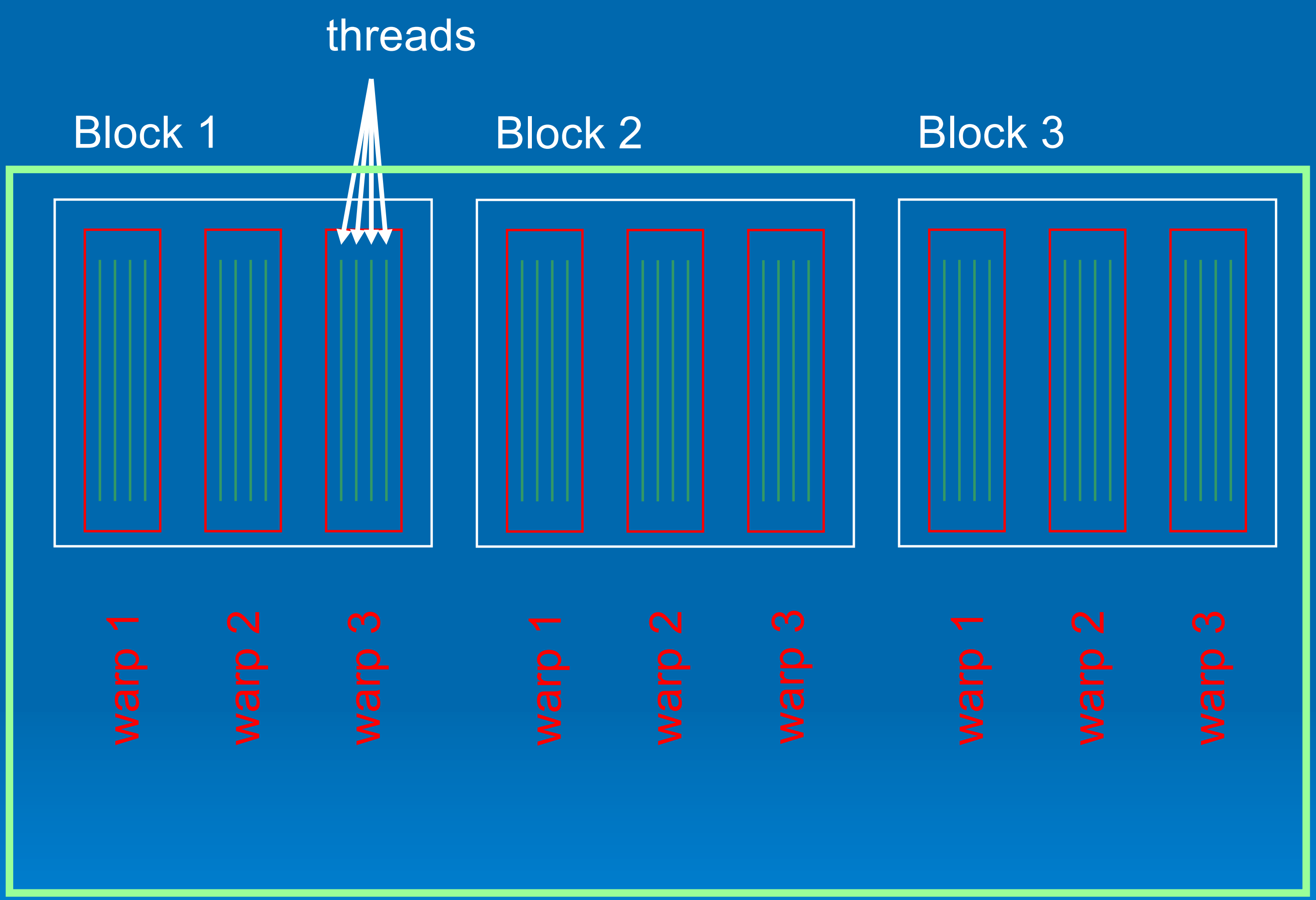
CUDA Code

```
__global__ void saxpy_parallel(int n,
                               float alpha,
                               float *x,
                               float *y) {
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  if (i<n)
     y[i] = alpha*x[i] + y[i];
}
```

“Warps”

- Each block is split into SIMD groups of threads called "warps".

- Each warp contains the same number of threads, called the "warp size”


threads
Block 1
Block 2
Block 3
Block 4
warp 1
warp 2
warp 3
warp 1
warp 2
warp 3
warp 1
warp 2
warp 3
warp 1
warp 2
warp 3
Multi-processor 1

Keeping multiprocessors in mind…

- Each multiprocessor can process multiple blocks at a time.
- How many depends on the number of registers per thread and how much shared memory per block is required by a given kernel.
- If a block is too large, it will not fit into the resources of an MP.

Performance Tip: Block Size

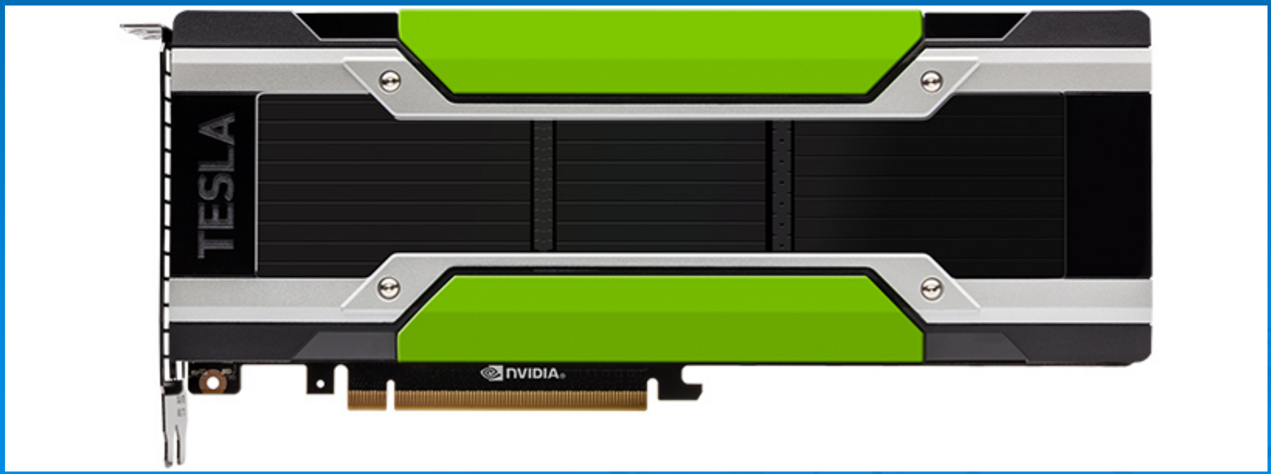
- Critical for performance
- Recommended value is 192 or 256
- Maximum value is 512
- Should be a multiple of 32 since this is the warp size for Series 8 GPUs and thus the native execution size for multiprocessors
- Limited by number of registers on the MP
- Series 8 GPU MPs have 8192 registers which are shared between all the threads on an MP

Performance Tip: Grid Size (number of blocks)

- Recommended value is at least 100, but 1000 would scale for many generations of hardware
- Actual value depends on problem size
- It should be a multiple of the number of MPs for an even distribution of work (not a requirement though)
- Example: 24 blocks
  - Grid will work efficiently on Series 8 (12 MPs), but it will waste resources on new GPUs with 32MPs

Example: Tesla P100

- Launched in 2016
- "Pascal" architecture (successors: Volta, Turing)
- Double-precision performance: 4.7 TeraFLOPS
- Single-precision performance: 9.3 TeraFLOPS
- GPU Memory: 16 GB



Example: Tesla P100

- Number of Multiprocessors (MPs): 56
- Number of Cuda Cores per MP: 64
- Total number of Cuda Cores: 3584
- #Cuda Cores = #number of floating point instructions that can be processed per cycle
- MPs can run multiple threads per core simultaneously (similar to hyperthreading on CPU)
- Hence, #threads can be larger than #cores

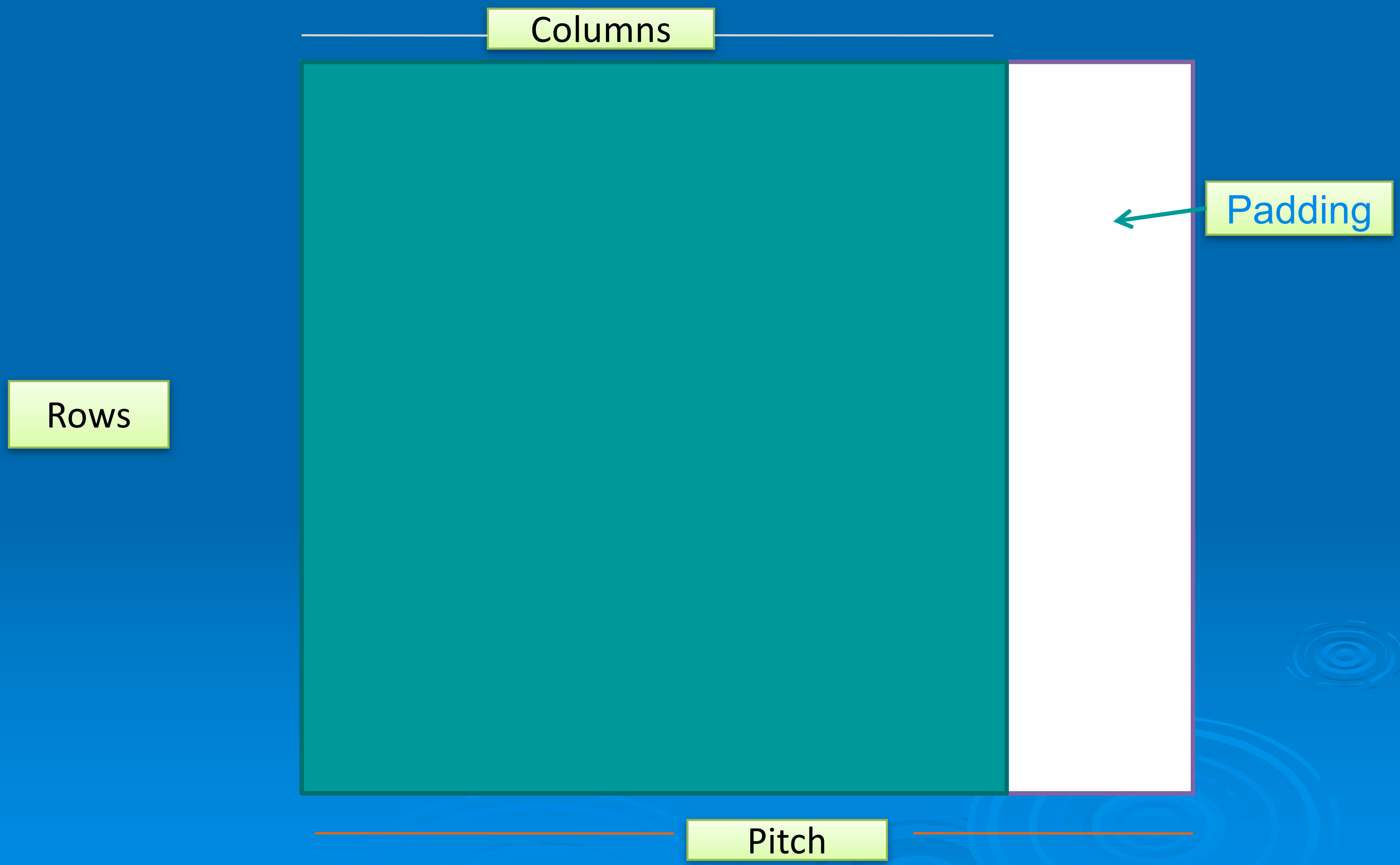
Memory Alignment

- Memory access faster if data aligned at 64 byte boundaries
- Hence, allocate 2D arrays so that every row starts at a 64-byte boundary
- Tedious for a programmer

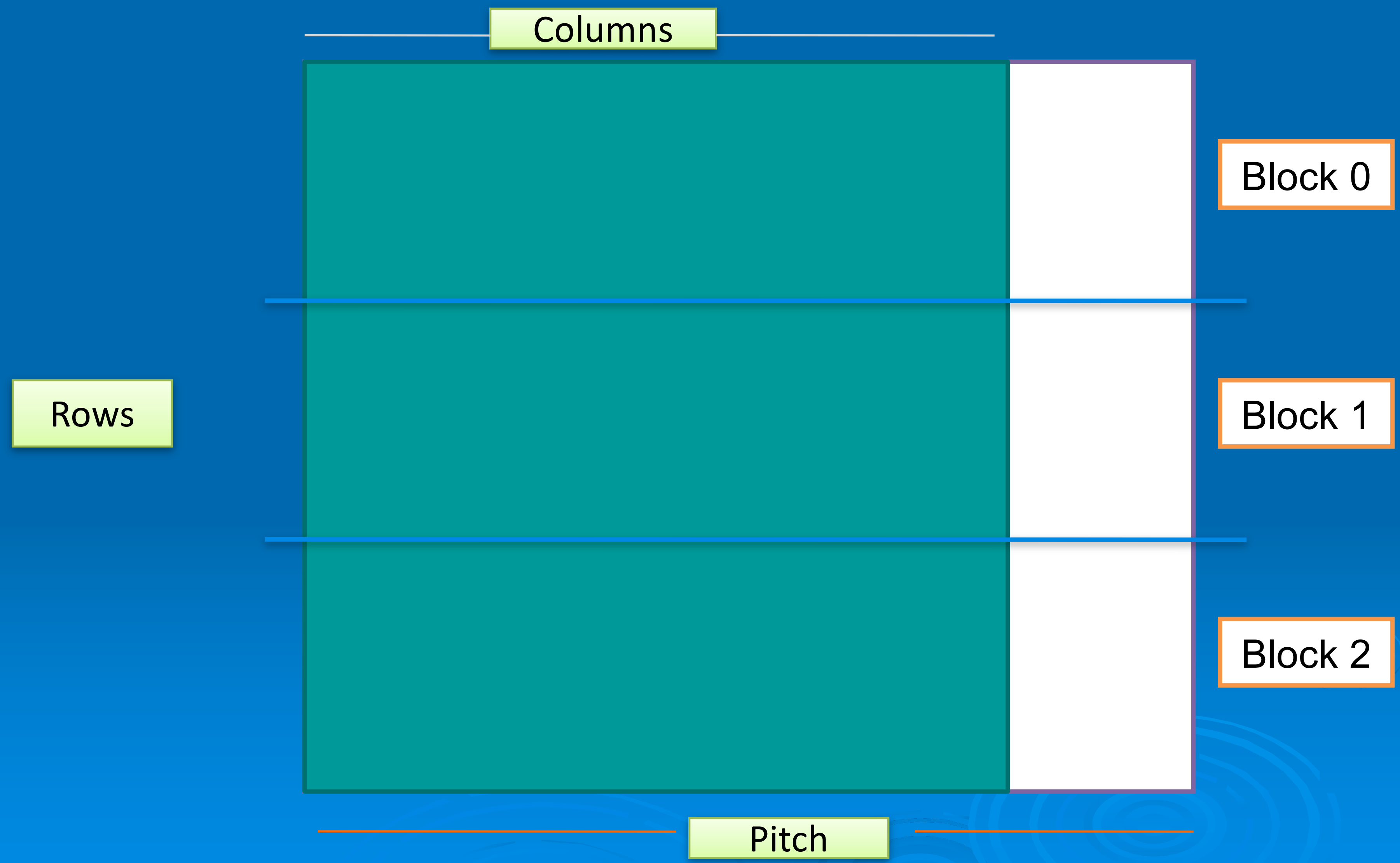
Allocating 2D arrays with "pitch"

- CUDA offers special versions of:
  - Memory allocation of 2D arrays so that every row is padded (if necessary): cudaMallocPitch()
  - Memory copy operations that take into account the pitch: cudaMemcpy2D()

Pitch



Dividing the work by blocks:



Watchdog timer

- OS may force programs using the GPU to time out if running too long
- Exceeding the limit can cause CUDA program failure.
- Possible solution: run CUDA on a GPU that is NOT attached to a display.

Resources on line

- http://www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=532
- http://www.ddj.com/hpc-high-performance-computing/207200659
- http://www.nvidia.com/object/cuda_home.html#
- http://www.nvidia.com/object/cuda_learn.html
- "Computation of Voronoi diagrams using a graphics processing unit" by Igor Majdandzic et al. available through IEEE Digital Library, DOI: 10.1109/EIT.2008.4554342