Heterogeneous programming

CUDA

Thierry Garcia

CY Tech

Bibliography - Web Resources

- Jason Sanders, "CUDA by Example: An Introduction to General-Purpose GPU Programming", 1st Edition, AddisonWesley, 2010
- D. R. Kaeli, P. Mistry, Dana Schaa, and D. P. Zhang. "Heterogeneous Computing with OpenCL 2.0." Morgan Kaufmann Publishers Inc., 2015.
- Wen-mei W. Hwu, David B. Kirk and Izzat El Hajj, "Programming Massively Parallel Processors, A Hands-on Approach", Fourth Edition, Elsevier, 2023.
- Slides (french) of Frédéric Audra, Glenn Cougoulat
- `https://www.boston.co.uk/info/nvidia-kepler/what-is-gpu-computing.aspx`
- `http://indexation.univ-fcomte.fr/nuxeo/site/esupversions/b8d40adf-f79f-4de`
- `https://www.malekal.com/cpu-vs-gpu-les-differences/`
- `http://www.enseignement.polytechnique.fr/profs/informatique/Eric.Goubault/Cours05html/poly002.html`
- `https://en.wikipedia.org/wiki/Taxonomy_of_Flynn`

Bibliography - Web Resources

- http://www.irisa.fr/alf/downloads/collange/cours/ppar2020_gpu_1.pdf
- https://www.thegeekstuff.com/2012/03/linux-threads-intro/
- https://www.thegeekstuff.com/2012/03/linux-threads-intro/
- https://hal.archives-ouvertes.fr/tel-02977242/document
- https://medium.com/altumea/a-brief-history-of-gpu-47d98d6a0f8a
- https://www.techspot.com/article/650-history-of-the-gpu/
- https://www.pocket-lint.com/gadgets/news/pc-gaming/160954-nvidia-gpu-hist
- https://www.nextplatform.com/2016/04/19/drilling-nvidias-pascal-gpu/
- https://www.techcenturion.com/nvidia-cuda-cores/#nbspnbspnbspnbspCUDA_Cores
- http://www.metz.supelec.fr/metz/personnel/vialle/course/Mineure-HPC/notes-de-cours-specifiques/HPC-05a-GPU-CUDA-archi-bases-2spp.pdf
- https://leria-info.univ-angers.fr/~jeanmichel.richer/cuda_crs2.php
- https://forums.developer.nvidia.com/t/how-to-measure-total-time-for-cpu-a 28234/2

Objectives

- acquire the basic training to analyse heterogeneous architectures with accelerators such as a GPU, as an alternative to multi-core systems of general purpose processors, and be able to compare their performance.
- develop efficient software for these platforms by means of languages.

Course Goals

- learn how to program heterogeneous computing systems.
- use parallel programming API, tools and techniques.

Contents

- Introduction and reminders (autonomous lessons).
- Structure of a heterogeneous system CPU-GPU.
- Introduction to CUDA programming.
- Optimization techniques.
- Programming on heterogeneous systems CPU-GPU.

Data parallelism

Data parallelism

Data parallelism

- When modern software applications run slowly, the problem is usually data-too much data to process.

Data parallelism

- When modern software applications run slowly, the problem is usually data-too much data to process.
- Image-processing applications manipulate images or videos with millions to trillions of pixels.

Data parallelism

- When modern software applications run slowly, the problem is usually data-too much data to process.
- Image-processing applications manipulate images or videos with millions to trillions of pixels.
- Scientific applications model fluid dynamics using billions of grid points.

Data parallelism

- When modern software applications run slowly, the problem is usually data-too much data to process.
- Image-processing applications manipulate images or videos with millions to trillions of pixels.
- Scientific applications model fluid dynamics using billions of grid points.
- Molecular dynamics applications must simulate interactions between thousands to billions of atoms.

Data parallelism

- When modern software applications run slowly, the problem is usually data-too much data to process.
- Image-processing applications manipulate images or videos with millions to trillions of pixels.
- Scientific applications model fluid dynamics using billions of grid points.
- Molecular dynamics applications must simulate interactions between thousands to billions of atoms.
- Airline scheduling deals with thousands of flights, crews, and airport gates.

Data parallelism

- When modern software applications run slowly, the problem is usually data-too much data to process.
- Image-processing applications manipulate images or videos with millions to trillions of pixels.
- Scientific applications model fluid dynamics using billions of grid points.
- Molecular dynamics applications must simulate interactions between thousands to billions of atoms.
- Airline scheduling deals with thousands of flights, crews, and airport gates.
- For example, in image processing, converting a color pixel to grayscale requires only the data of that pixel.

Data parallelism

- When modern software applications run slowly, the problem is usually data-too much data to process.
- Image-processing applications manipulate images or videos with millions to trillions of pixels.
- Scientific applications model fluid dynamics using billions of grid points.
- Molecular dynamics applications must simulate interactions between thousands to billions of atoms.
- Airline scheduling deals with thousands of flights, crews, and airport gates.
- For example, in image processing, converting a color pixel to grayscale requires only the data of that pixel.
- A global operation, such as finding the average brightness of all pixels in an image, can be broken down into many smaller computations that can be executed independently.

Data parallelism

- Writing data parallel code entails (re)organizing the computation around the data such that we can execute the resulting independent computations in parallel to go much faster.

Data parallelism

- Writing data parallel code entails (re)organizing the computation around the data such that we can execute the resulting independent computations in parallel to go much faster.
- Example of data parallelism with a color-to-grayscale conversion example :

Data parallelism

- Writing data parallel code entails (re)organizing the computation around the data such that we can execute the resulting independent computations in parallel to go much faster.
- Example of data parallelism with a color-to-grayscale conversion example :
  - Fig. 19 shows a color image (left side) consisting of many pixels, each containing a red, green, and blue fractional value (r, g, b) varying from 0 (black) to 1 (full intensity).

Data parallelism

- Writing data parallel code entails (re)organizing the computation around the data such that we can execute the resulting independent computations in parallel to go much faster.
- Example of data parallelism with a color-to-grayscale conversion example :
  - Fig. 19 shows a color image (left side) consisting of many pixels, each containing a red, green, and blue fractional value (r, g, b) varying from 0 (black) to 1 (full intensity).
  - To convert the color image (left side of Fig. 18) to a grayscale image (right side), we compute the luminance value L for each pixel by applying the following weighted sum formula:

$$L = r^*0.21 + g^*0.72 + b^*0.07$$

Data parallelism

- Writing data parallel code entails (re)organizing the computation around the data such that we can execute the resulting independent computations in parallel to go much faster.
- Example of data parallelism with a color-to-grayscale conversion example :
  - Fig. 19 shows a color image (left side) consisting of many pixels, each containing a red, green, and blue fractional value (r, g, b) varying from 0 (black) to 1 (full intensity).
  - To convert the color image (left side of Fig. 19) to a grayscale image (right side), we compute the luminance value L for each pixel by applying the following weighted sum formula:

$$L = r^*0.21 + g^*0.72 + b^*0.07$$

Figure: Conversion of a color image to a grayscale image.

Data parallelism

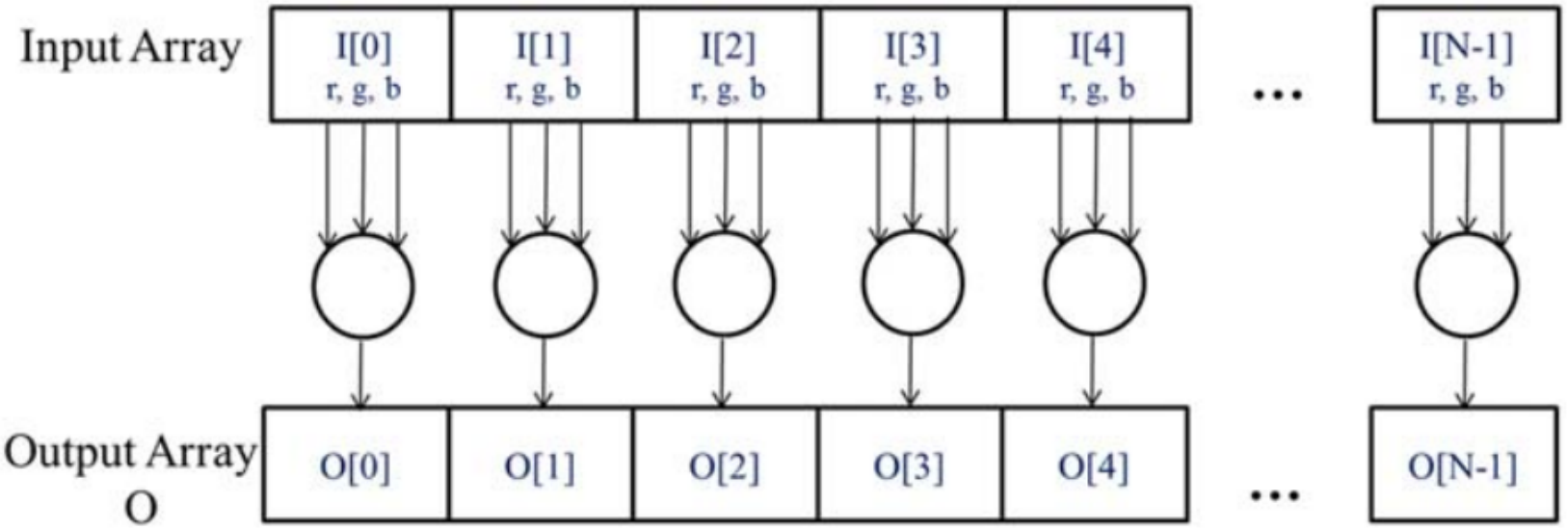


Figure: Data parallelism in image-to-grayscale conversion. Pixels can be calculated independently of each other.

CUDA

What is CUDA?

- CUDA C extends ANSI C programming language with minimal new syntax and library functions to let programmers target heterogeneous computing systems containing both CPU cores and massively parallel GPUs.

CUDA

What is CUDA?

- CUDA C extends ANSI C programming language with minimal new syntax and library functions to let programmers target heterogeneous computing systems containing both CPU cores and massively parallel GPUs.
- As the name implies, CUDA C is built on NVIDIA's CUDA platform.

CUDA

What is CUDA?

- CUDA C extends ANSI C programming language with minimal new syntax and library functions to let programmers target heterogeneous computing systems containing both CPU cores and massively parallel GPUs.
- As the name implies, CUDA C is built on NVIDIA's CUDA platform.
- CUDA is currently the most mature framework for massively parallel computing.

CUDA

What is CUDA?

- CUDA C extends ANSI C programming language with minimal new syntax and library functions to let programmers target heterogeneous computing systems containing both CPU cores and massively parallel GPUs.
- As the name implies, CUDA C is built on NVIDIA's CUDA platform.
- CUDA is currently the most mature framework for massively parallel computing.
- It is broadly used in the high-performance computing industry, with essential tools such as compilers, debuggers, and profilers available on the most common operating systems.

CUDA

What is CUDA?

- CUDA C extends ANSI C programming language with minimal new syntax and library functions to let programmers target heterogeneous computing systems containing both CPU cores and massively parallel GPUs.
- As the name implies, CUDA C is built on NVIDIA's CUDA platform.
- CUDA is currently the most mature framework for massively parallel computing.
- It is broadly used in the high-performance computing industry, with essential tools such as compilers, debuggers, and profilers available on the most common operating systems.
- The structure of a CUDA C program reflects the coexistence of a host (CPU) and one or more devices (GPUs) in the computer.

CUDA

What is CUDA?

- CUDA C extends ANSI C programming language with minimal new syntax and library functions to let programmers target heterogeneous computing systems containing both CPU cores and massively parallel GPUs.
- As the name implies, CUDA C is built on NVIDIA's CUDA platform.
- CUDA is currently the most mature framework for massively parallel computing.
- It is broadly used in the high-performance computing industry, with essential tools such as compilers, debuggers, and profilers available on the most common operating systems.
- The structure of a CUDA C program reflects the coexistence of a host (CPU) and one or more devices (GPUs) in the computer.
- Each CUDA C source file can have a mixture of host code and device code.

CUDA

What is CUDA?

- By default, any traditional C program is a CUDA program that contains only host code.

CUDA

What is CUDA?

- By default, any traditional C program is a CUDA program that contains only host code.
- One can add device code into any source file.

CUDA

What is CUDA?

- By default, any traditional C program is a CUDA program that contains only host code.
- One can add device code into any source file.
- The device code is clearly marked with special CUDA C keywords.

CUDA

What is CUDA?

- By default, any traditional C program is a CUDA program that contains only host code.
- One can add device code into any source file.
- The device code is clearly marked with special CUDA C keywords.
- The device code includes functions, or kernels, whose code is executed in a data-parallel manner.

CUDA

What is CUDA?

- By default, any traditional C program is a CUDA program that contains only host code.
- One can add device code into any source file.
- The device code is clearly marked with special CUDA C keywords.
- The device code includes functions, or kernels, whose code is executed in a data-parallel manner.
- The execution of a CUDA program is illustrated in Fig. 3.

CUDA

What is CUDA?

- By default, any traditional C program is a CUDA program that contains only host code.
- One can add device code into any source file.
- The device code is clearly marked with special CUDA C keywords.
- The device code includes functions, or kernels, whose code is executed in a data-parallel manner.
- The execution of a CUDA program is illustrated in Fig. 3.
- The execution starts with host code (CPU serial code).

CUDA

What is CUDA?

- By default, any traditional C program is a CUDA program that contains only host code.
- One can add device code into any source file.
- The device code is clearly marked with special CUDA C keywords.
- The device code includes functions, or kernels, whose code is executed in a data-parallel manner.
- The execution of a CUDA program is illustrated in Fig. 3.
- The execution starts with host code (CPU serial code).
- When a kernel function is called, a large number of threads are launched on a device to execute the kernel.

CUDA

What is CUDA?

- By default, any traditional C program is a CUDA program that contains only host code.
- One can add device code into any source file.
- The device code is clearly marked with special CUDA C keywords.
- The device code includes functions, or kernels, whose code is executed in a data-parallel manner.
- The execution of a CUDA program is illustrated in Fig. 3.
- The execution starts with host code (CPU serial code).
- When a kernel function is called, a large number of threads are launched on a device to execute the kernel.
- All the threads that are launched by a kernel call are collectively called a grid.These threads are the primary vehicle of parallel execution in a CUDA platform.

CUDA

What is CUDA?

- Figure 3 shows the execution of two grids of threads. When all threads of a grid have completed their execution, the grid terminates, and the execution continues on the host until another grid is launched.

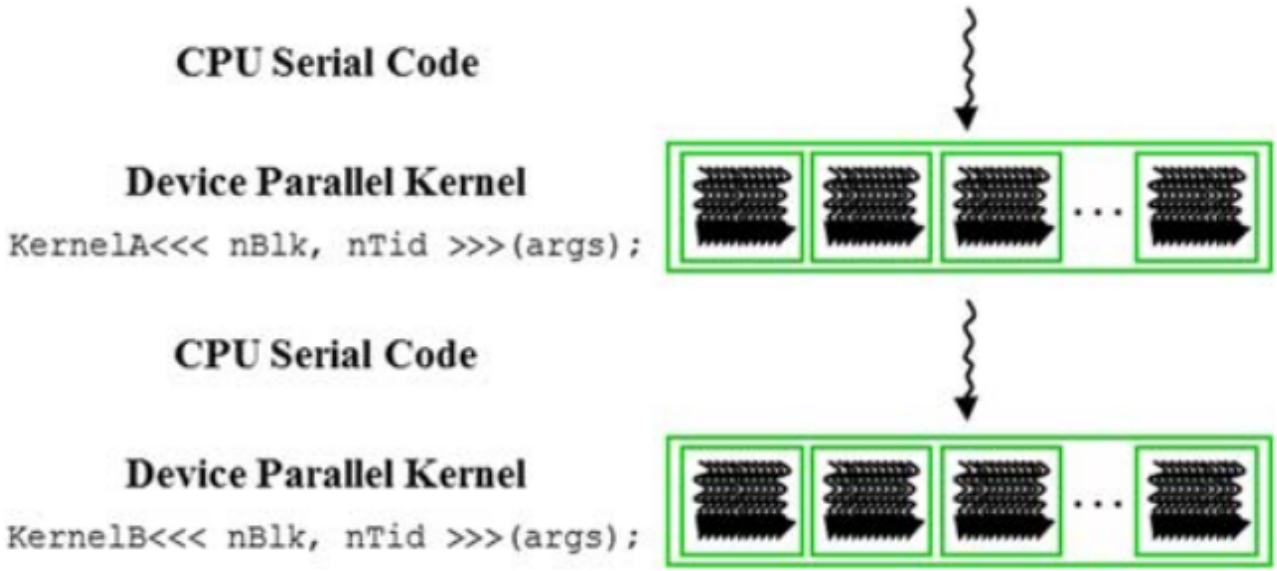


Figure: Execution of a CUDA program.

CUDA

What is CUDA?

- CUDA Architecture

CUDA

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing

CUDA

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance

CUDA

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++

CUDA

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++

CUDA

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming

CUDA

What is CUDA?

- CUDA Architecture
  - Expose GPU parallelism for general-purpose computing
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Direct APIs to manage devices, memory etc.

CUDA

- Until 2006, GPUs were difficult to program because it was necessary to use the APIs (Application Programming Interface) of the graphics functions (OpenGL and Direct3D), which limited performance and the type of application that could be parallelized.

CUDA

- Until 2006, GPUs were difficult to program because it was necessary to use the APIs (Application Programming Interface) of the graphics functions (OpenGL and Direct3D), which limited performance and the type of application that could be parallelized.
- From 2007, NVidia made the CUDA (Compute Unified Device Architecture) available to programmers.

CUDA

- Until 2006, GPUs were difficult to program because it was necessary to use the APIs (Application Programming Interface) of the graphics functions (OpenGL and Direct3D), which limited performance and the type of application that could be parallelized.
- From 2007, NVidia made the CUDA (Compute Unified Device Architecture) available to programmers.
- Nvidia said : "CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU)."

CUDA

CUDA programming is based on two models:

CUDA

CUDA programming is based on two models:

- the logic model which allows to express :

CUDA

CUDA programming is based on two models:

- the logic model which allows to express :
  - the calculation to be performed (C/C++ code)

CUDA

CUDA programming is based on two models:

- the logic model which allows to express :
  - the calculation to be performed (C/C++ code)
  - the organisation of threads in the form of a grid of thread blocks

CUDA

CUDA programming is based on two models:

- the logic model which allows to express :
  - the calculation to be performed (C/C++ code)
  - the organisation of threads in the form of a grid of thread blocks
- the physical model which is responsible for distributing and executing the threads on the GPU's execution cores

CUDA

- The major problem in CUDA is that the logical and physical models are strongly coupled and in particular the physical model has a strong influence on the organisation of threads.

CUDA

- The major problem in CUDA is that the logical and physical models are strongly coupled and in particular the physical model has a strong influence on the organisation of threads.
- Finding the most efficient solution to a given problem is not straightforward as we will see. One must take into account :

CUDA

- The major problem in CUDA is that the logical and physical models are strongly coupled and in particular the physical model has a strong influence on the organisation of threads.
- Finding the most efficient solution to a given problem is not straightforward as we will see. One must take into account :
  - choice of grid and block size

CUDA

- The major problem in CUDA is that the logical and physical models are strongly coupled and in particular the physical model has a strong influence on the organisation of threads.
- Finding the most efficient solution to a given problem is not straightforward as we will see. One must take into account :
  - choice of grid and block size
  - use of shared memory, constant memory, texture memory

CUDA

- The major problem in CUDA is that the logical and physical models are strongly coupled and in particular the physical model has a strong influence on the organisation of threads.
- Finding the most efficient solution to a given problem is not straightforward as we will see. One must take into account :
  - choice of grid and block size
  - use of shared memory, constant memory, texture memory
  - avoidance of branches

CUDA

- The major problem in CUDA is that the logical and physical models are strongly coupled and in particular the physical model has a strong influence on the organisation of threads.
- Finding the most efficient solution to a given problem is not straightforward as we will see. One must take into account :
  - choice of grid and block size
  - use of shared memory, constant memory, texture memory
  - avoidance of branches
  - ...

CUDA

Terminology:

- Host : The CPU and its memory (host memory).

CUDA

Terminology:

- Host : The CPU and its memory (host memory).
- Device : The GPU and its memory (device memory)

CUDA

Terminology:

- Host : The CPU and its memory (host memory).
- Device : The GPU and its memory (device memory)

Introduction

Remark: When programming in CUDA we try to differentiate between data in main memory and data in GPU memory. In general, a prefix is used for variable names:

Introduction

Remark: When programming in CUDA we try to differentiate between data in main memory and data in GPU memory. In general, a prefix is used for variable names:

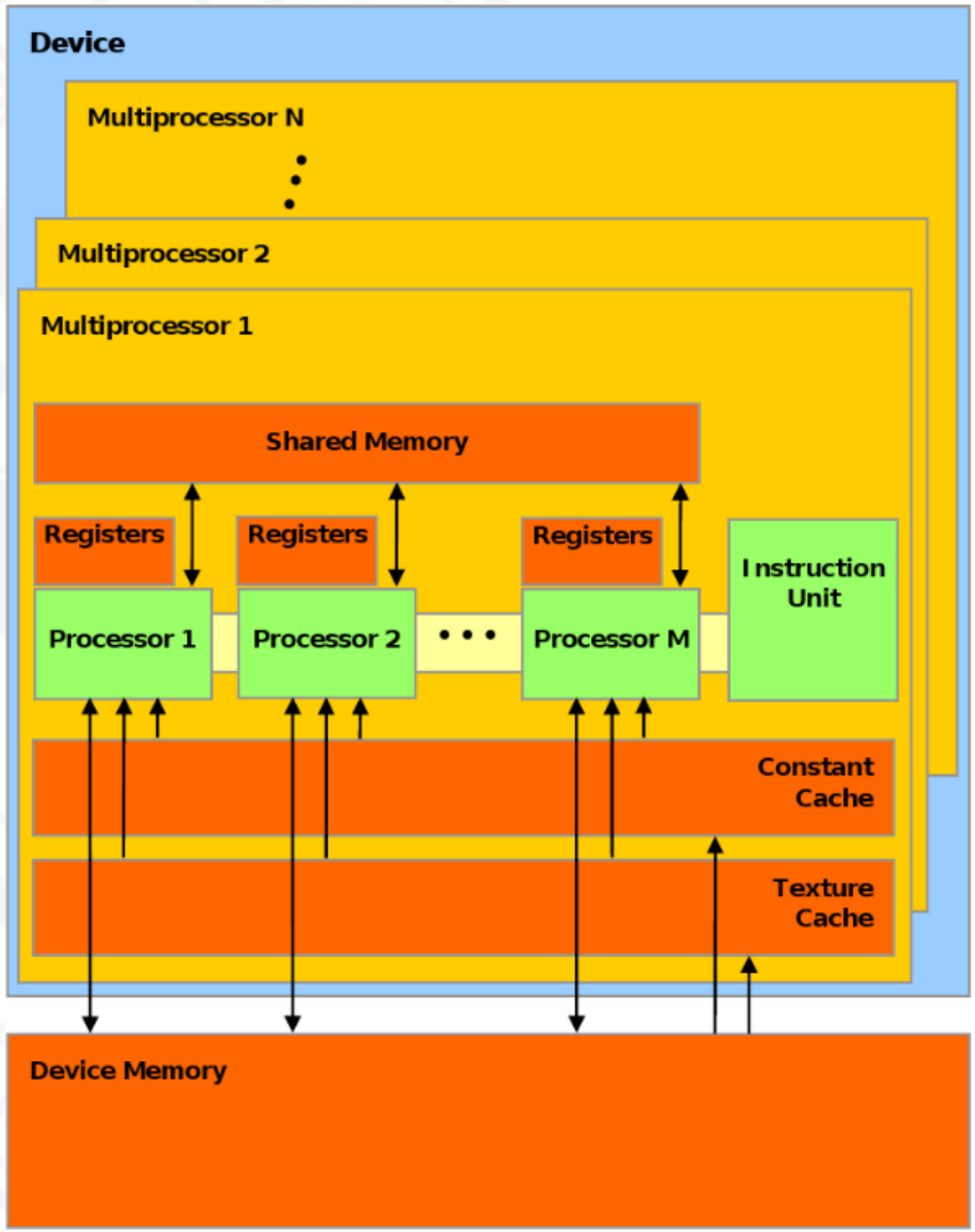
- host = CPU: h_mem, hMem, c_mem, CMem, ...

Introduction

Remark: When programming in CUDA we try to differentiate between data in main memory and data in GPU memory. In general, a prefix is used for variable names:

- host = CPU: h_mem, hMem, c_mem, CMem, ...
- device = GPU: d_mem, dMem, g_mem, gMem, ...

Introduction



CUDA Runtime Model: General principles

CUDA

CUDA Runtime Model: General principles

- Memory hierarchy

CUDA

CUDA Runtime Model: General principles

- Memory hierarchy
  - Registers,

CUDA

CUDA Runtime Model: General principles

- Memory hierarchy
  - Registers,
  - Shared memory,

CUDA

CUDA Runtime Model: General principles

- Memory hierarchy
  - Registers,
  - Shared memory,
  - Global memory.

CUDA

CUDA Runtime Model: General principles

- Memory hierarchy
  - Registers,
  - Shared memory,
  - Global memory.
- Many threads in parallel

CUDA

CUDA Runtime Model: General principles

- Memory hierarchy
  - Registers,
  - Shared memory,
  - Global memory.
- Many threads in parallel
  - Kernel: the same program executed by several threads at the same time

CUDA

CUDA Runtime Model: General principles

- Memory hierarchy
  - Registers,
  - Shared memory,
  - Global memory.
- Many threads in parallel
  - Kernel: the same program executed by several threads at the same time
  - Very light threads (very fast context creation and context)

CUDA

CUDA Runtime Model: General principles

- Memory hierarchy
  - Registers,
  - Shared memory,
  - Global memory.
- Many threads in parallel
  - Kernel: the same program executed by several threads at the same time
  - Very light threads (very fast context creation and context)
  - Thousands of threads on hundreds of processors for optimal operation.

CUDA

Kernels and threads

CUDA

Kernels and threads

- Kernel: code running in parallel

CUDA

Kernels and threads

- Kernel: code running in parallel
- One kernel executed by all threads

CUDA

Kernels and threads

- Kernel: code running in parallel
- One kernel executed by all threads
- Threads all execute the same code

CUDA

Cooperation between threads

CUDA

Cooperation between threads

- Sharing of intermediate results, factoring of memory accesses

CUDA

Cooperation between threads

- Sharing of intermediate results, factoring of memory accesses
  - Beware, very inefficient global cooperation.

CUDA

Cooperation between threads

- Sharing of intermediate results, factoring of memory accesses
  - Beware, very inefficient global cooperation.
- A kernel launches a grid of thread blocks

CUDA

Cooperation between threads

- Sharing of intermediate results, factoring of memory accesses
  - Beware, very inefficient global cooperation.
- A kernel launches a grid of thread blocks
- Communication intra-block by shared memory

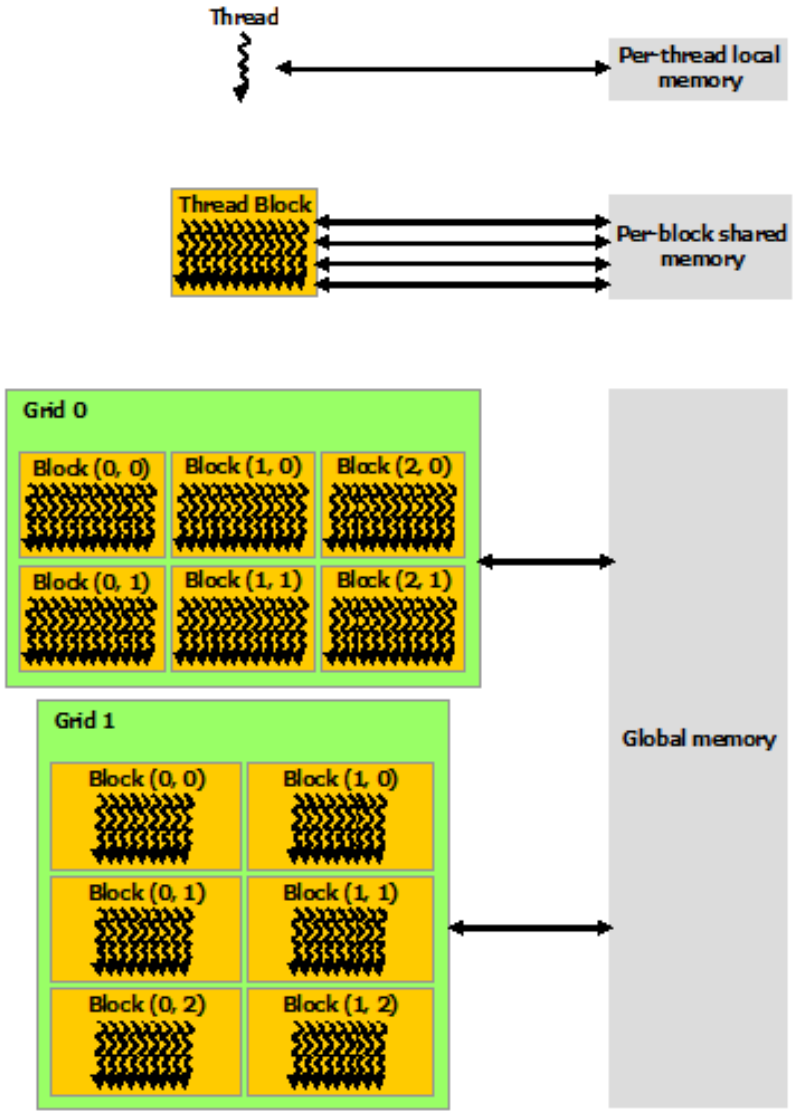
CUDA

Cooperation between threads

- Sharing of intermediate results, factoring of memory accesses
  - Beware, very inefficient global cooperation.
- A kernel launches a grid of thread blocks
- Communication intra-block by shared memory
- No inter-block cooperation

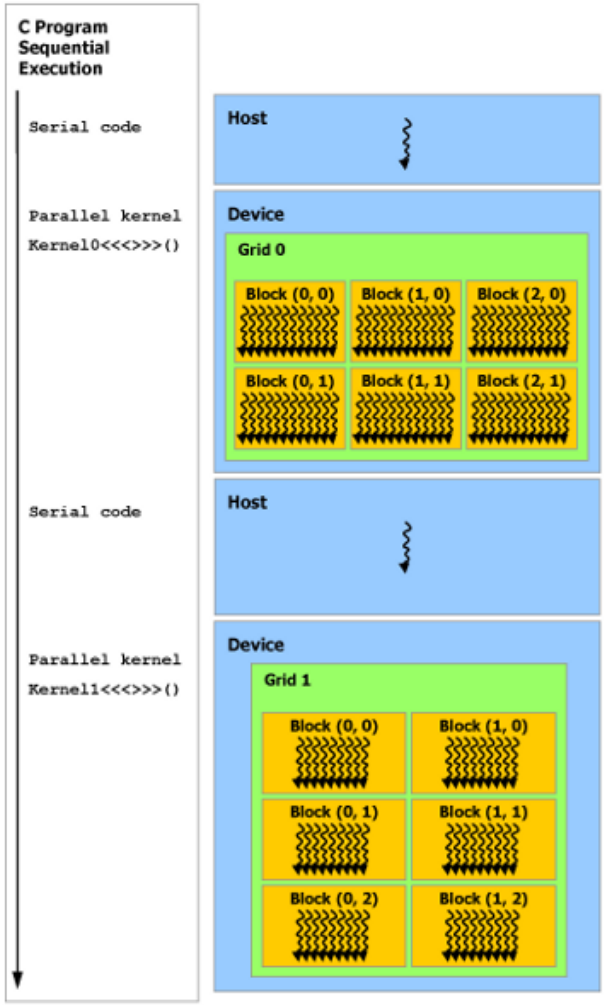
CUDA

Memory hierarchy



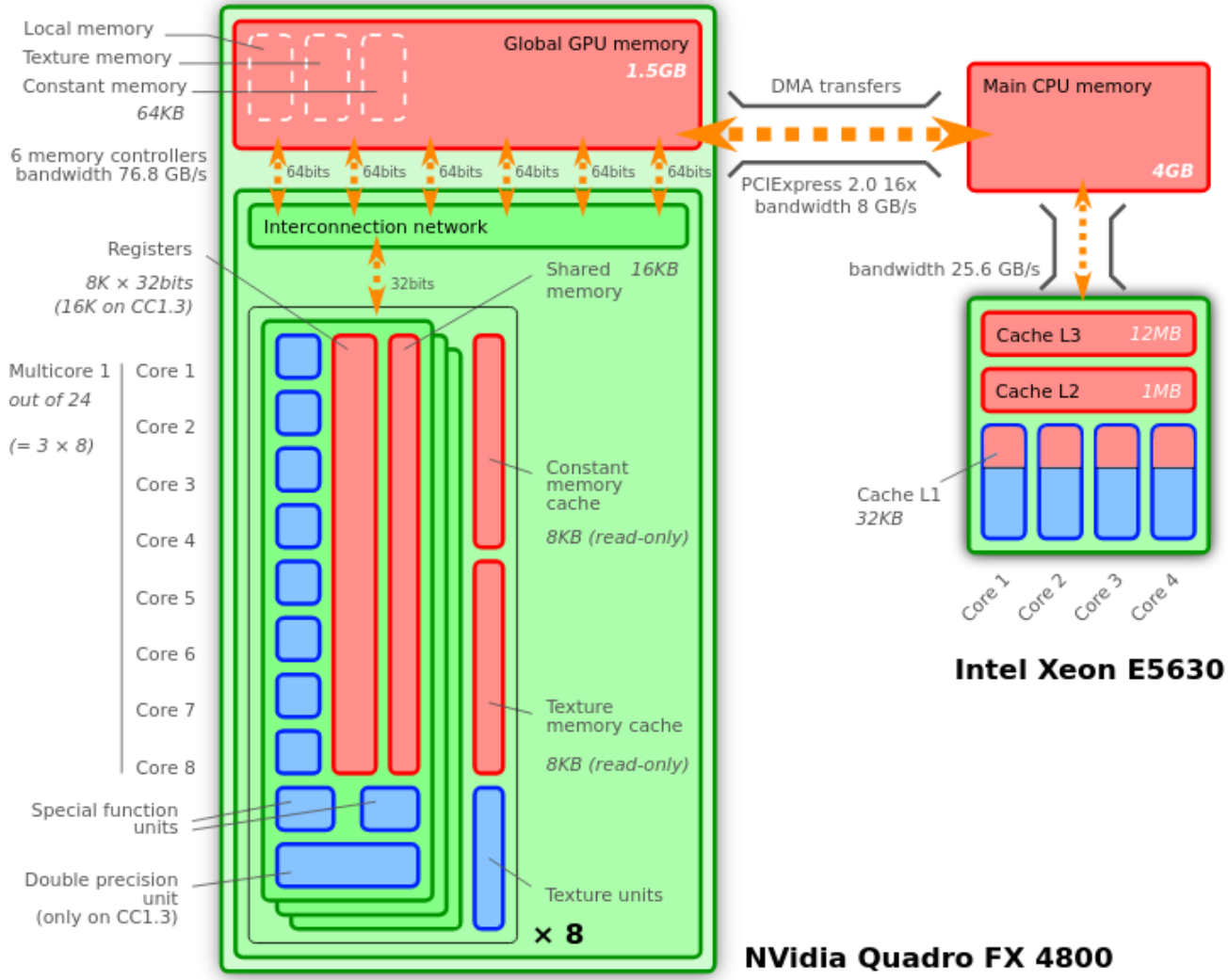
CUDA

Heterogeneous Programming



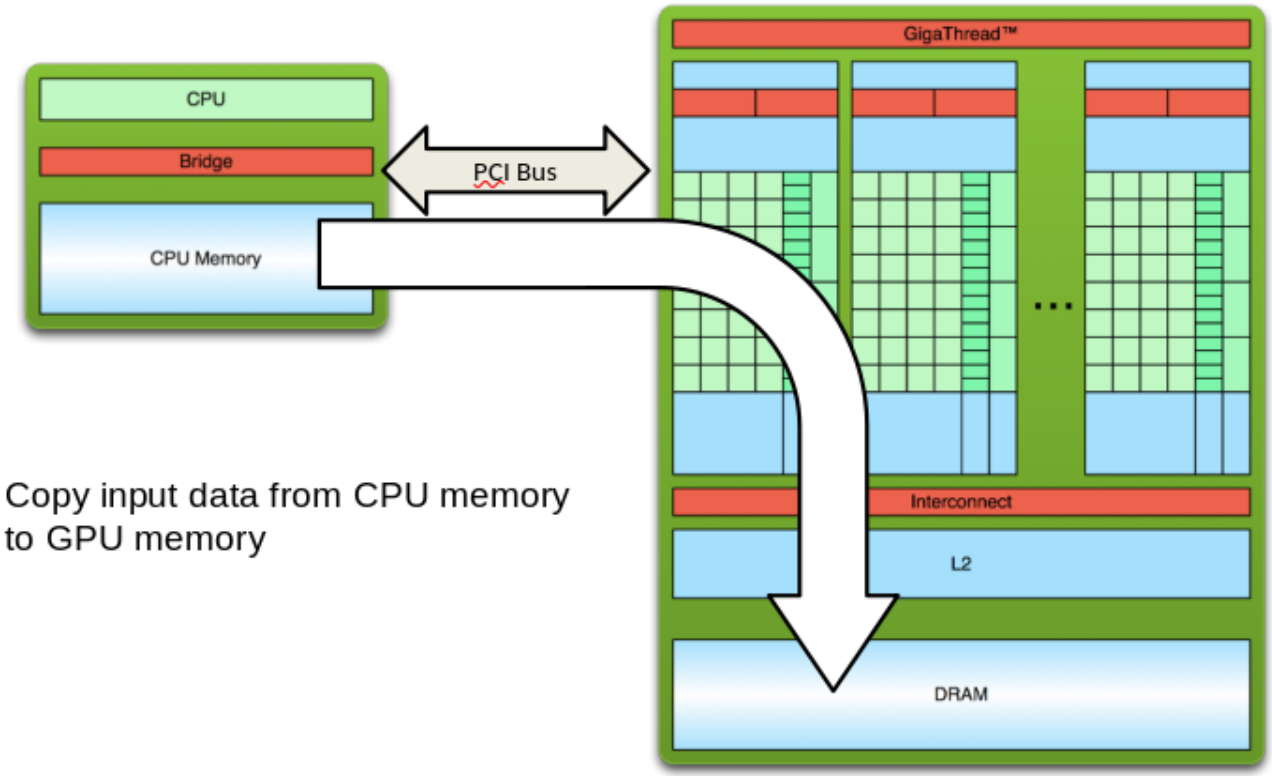
CUDA

- A GPU is a set of N small independent SIMD machines sharing a memory: *N* multiprocessors



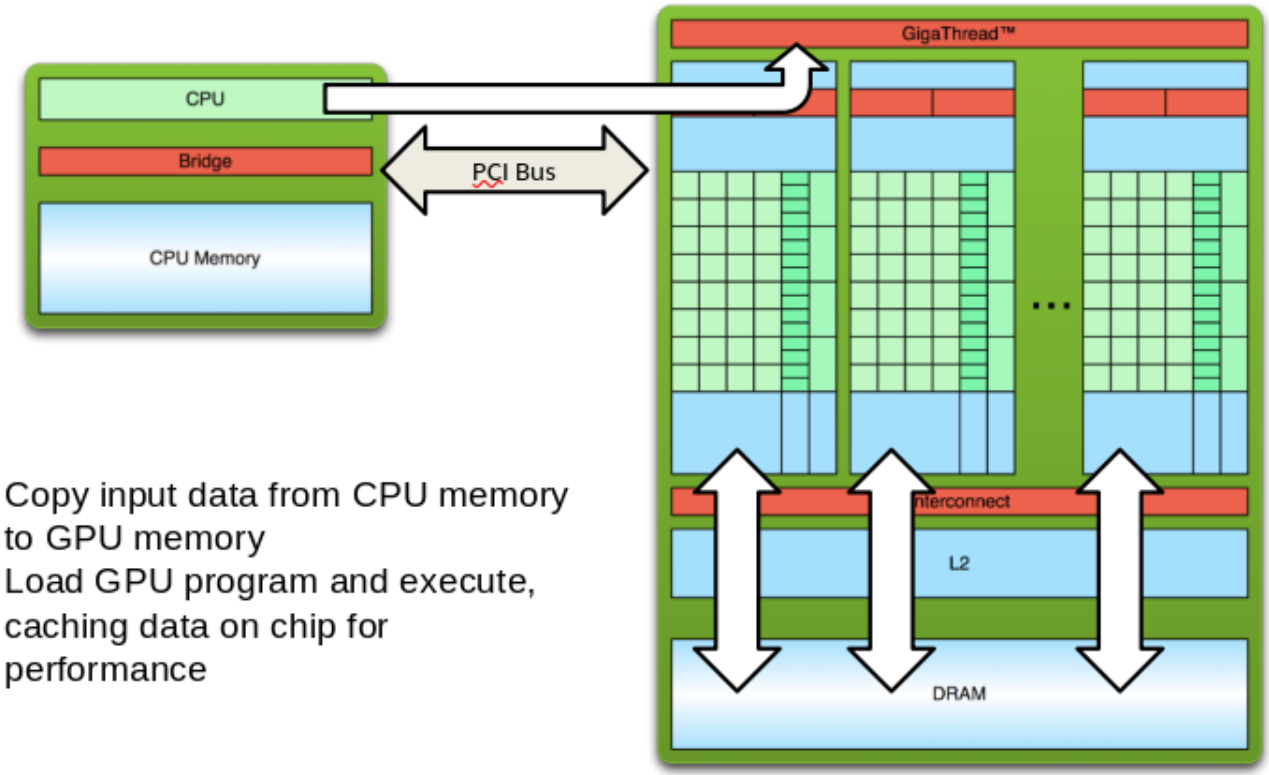
Example hardware of nVIDIA Tesla GPU architecture

Simple Processing Flow



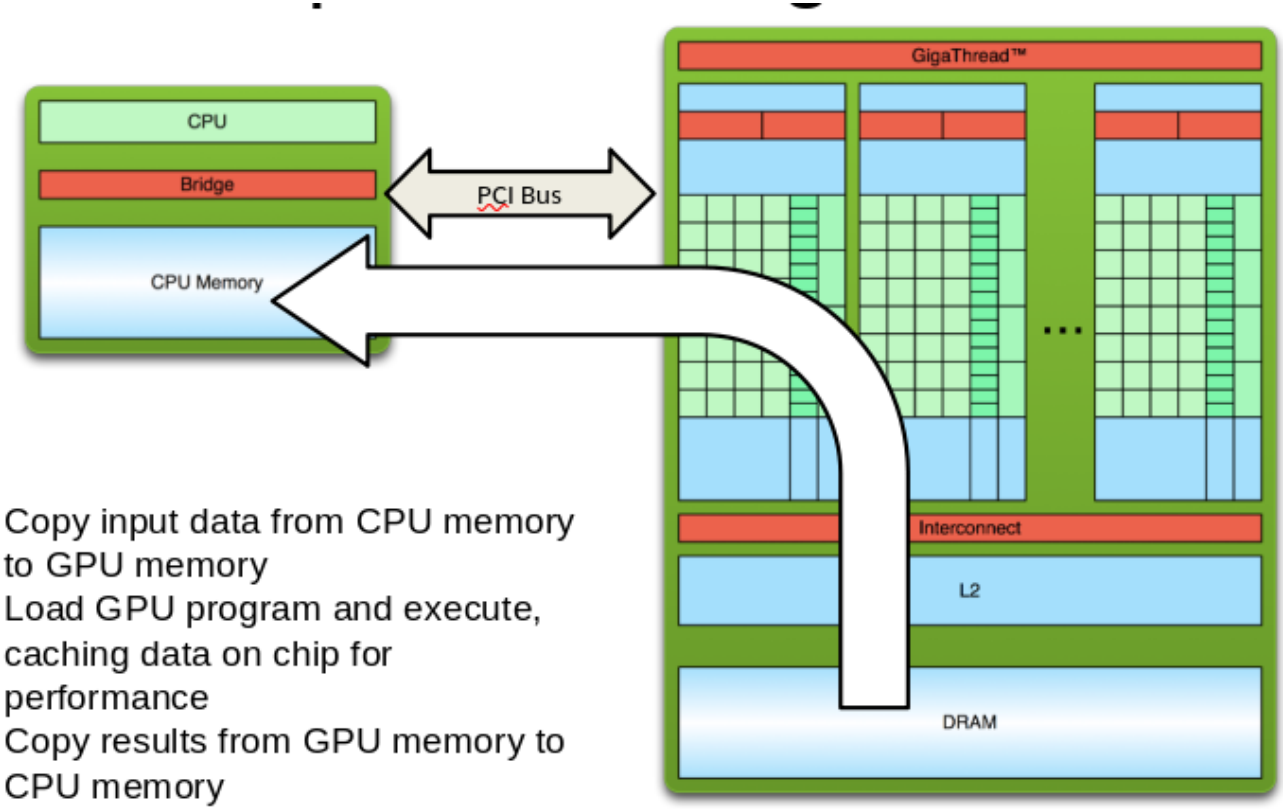
1. Copy input data from CPU memory to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance

Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory

CUDA

Cooperation between threads

- Sharing of intermediate results, factoring of memory accesses
  - Beware, very inefficient global cooperation.
- A kernel launches a grid of thread blocks
- Communication intra-block by shared memory
- No inter-block cooperation

CUDA: First Program

Hello World!

```
int main(void) {
                printf("Hello World!\n");
                return 0;
        }
```

- Standard C that runs on the host (CPU and system's memory)
- NVIDIA compiler (nvcc) can be used to compile programs with no device code

```
>nvcc hello_world.cu
>./a.out
Hello World!
>
```

CUDA: First Program

Hello World! with Device Code

```
#include <stdio.h>

__global__ void kernel(void) {
}

int main(void) {
        kernel<<<1,1>>>();
                printf("Hello World!\n");
                return 0;
        }
```

- A new empty function named kernel() qualified with __global__

CUDA: First Program

Hello World! with Device Code

```
#include <stdio.h>

__global__ void kernel(void) {
}

int main(void) {
        kernel<<<1,1>>>();
                printf("Hello World!\n");
                return 0;
        }
```

- A new empty function named kernel() qualified with __global__
- A call to this empty function embellished with $<<< 1, 1 >>>$

CUDA: First Program

Hello World! with Device Code

```
#include <stdio.h>

__global__ void kernel(void) {
}

int main(void) {
        kernel<<<1,1>>>();
                printf("Hello World!\n");
                return 0;
        }
```

- A new empty function named kernel() qualified with __global__
- A call to this empty function embellished with $<<<1,1>>>$
- CUDA C adds __global__ qualifier to standard C and alerts the compiler that a function should be compiled to run on a device instead of the host.

CUDA: First Program

Hello World! with Device Code

```
#include <stdio.h>

__global__ void kernel(void) {
}

int main(void) {
        kernel<<<1,1>>>();
                printf("Hello World!\n");
                return 0;
        }
```

- A new empty function named kernel() qualified with __global__
- A call to this empty function embellished with $<<<1,1>>>$
- CUDA C adds __global__ qualifier to standard C and alerts the compiler that a function should be compiled to run on a device instead of the host.
- nvcc separates source code into host and device components

CUDA: First Program

Hello World! with Device Code

```
#include <stdio.h>

__global__ void kernel(void) {
}

int main(void) {
        kernel<<<1,1>>>();
                printf("Hello World!\n");
                return 0;
        }
```

- A new empty function named kernel() qualified with __global__
- A call to this empty function embellished with $<<<1,1>>>$
- CUDA C adds __global__ qualifier to standard C and alerts the compiler that a function should be compiled to run on a device instead of the host.
- nvcc separates source code into host and device components
  - Device functions (kernel()) processed by NVIDIA compiler

CUDA: First Program

Hello World! with Device Code

```
#include <stdio.h>

__global__ void kernel(void) {
}

int main(void) {
        kernel<<<1,1>>>();
                printf("Hello World!\n");
                return 0;
        }
```

- A new empty function named kernel() qualified with __global__
- A call to this empty function embellished with $<<< 1, 1 >>>$
- CUDA C adds __global__ qualifier to standard C and alerts the compiler that a function should be compiled to run on a device instead of the host.
- nvcc separates source code into host and device components
  - Device functions (kernel()) processed by NVIDIA compiler
  - Host functions (main()) processed by standard host compiler gcc, cl.exe

CUDA : First Program

Hello World! with Device Code

```
kernel<<<1,1>>>();
```

- Triple angle brackets mark a call from host code to device code

CUDA : First Program

Hello World! with Device Code

```
kernel<<<1,1>>>();
```

- Triple angle brackets mark a call from host code to device code
  - Also called a “kernel launch”

CUDA : First Program

Hello World! with Device Code

```
kernel<<<1,1>>>();
```

- Triple angle brackets mark a call from host code to device code
  - Also called a "kernel launch"
  - We'll return to the parameters (1,1) in a moment

CUDA : First Program

Hello World! with Device Code

```
kernel<<<1,1>>>();
```

- Triple angle brackets mark a call from host code to device code
  - Also called a "kernel launch"
  - We'll return to the parameters (1,1) in a moment
- That's all that is required to execute a function on the GPU!

First Program

Hello World! with Device Code

```
#include <stdio.h>

__global__ void kernel(void) {
}

int main(void) {
        kernel<<<1,1>>>();
                printf("Hello World!\n");
                return 0;
        }
```

- kernel() does nothing for the moment.

CUDA

How test code ?

CUDA

Connection CESGA

- On LINUX, install openfortivpn : sudo dnf install openfortivpn or sudo apt install openfortivpn
- In a terminal : sudo openfortivpn gateway.cesga.es:443 -u cursoxxx@ce
- In another terminal : ssh cursoxxx@ft3.cesga.es
- compute –mem 1 –gpu <– mem = 1 Go
- scontrol show node c7257 <– hostname
- module load cuda
- nano hello.cu <– copy or create code
- nvcc hello.cu -o hello
- ./hello

CUDA

Connection CESGA

- Guide FT-3 (`https://cesga-docs.gitlab.io/ft3-user-guide/gpu_nodes.html`)
- 2 types de GPU dans le FT-3 : Nvidia A100 et Tesla T4 .
- Accès à T4 interactif via la commande "compute –gpu". Il n'y a que 10 nœuds avec T4 disponibles via cette option. S'il y a 10 utilisateurs demandant un T4, le suivant devra attendre que l'un des autres libère une de ces sessions.
- Accès aux 64 nœuds avec 2 A100 chacun. L'accès à ces nœuds se fait en envoyant des travaux à la file d'attente avec l'option "–gres=gpu" sans interactivité.
- Option intermédiaire, en fonction de l'occupation des A100 à tout moment, serait d'utiliser : salloc -I600 –gres=gpu –mem-per-cpu=3G -c 32 -t 10:00 srun -c 32 –pty –preserve-env /bin/-bash -i

CUDA

Connection CESGA

- Download and test deviceprop.cu

CUDA

Measuring performance

```
int main ( void ) {
float elapsed=0;
clock_t cpu_startTime, cpu_endTime;
double cpu_ElapseTime=0;
cudaEvent_t start, stop;
cpu_startTime = clock();
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel <<<1,1>>>() ;
cudaEventRecord(stop, 0);
cudaEventSynchronize (stop);
cudaEventElapsedTime(&elapsed, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
printf("The elapsed time in gpu was %.8f ms\n", elapsed);
printf ( "Hello World ! \n" ) ;
cpu_endTime = clock();
cpu_ElapseTime = ((cpu_endTime - cpu_startTime)/(double)
    CLOCKS_PER_SEC);
printf("The elapsed time in cpu was %.8f ms\n",
    cpu_ElapseTime);
return 0;}
```

CUDA

Let's test the code : hello.cu

CUDA

Parallel Programming in CUDA C/C++

CUDA

Parallel Programming in CUDA C/C++

- But wait ... GPU computing is about massive parallelism!

CUDA

Parallel Programming in CUDA C/C++

- But wait ... GPU computing is about massive parallelism!
- We need a more interesting example ...

CUDA

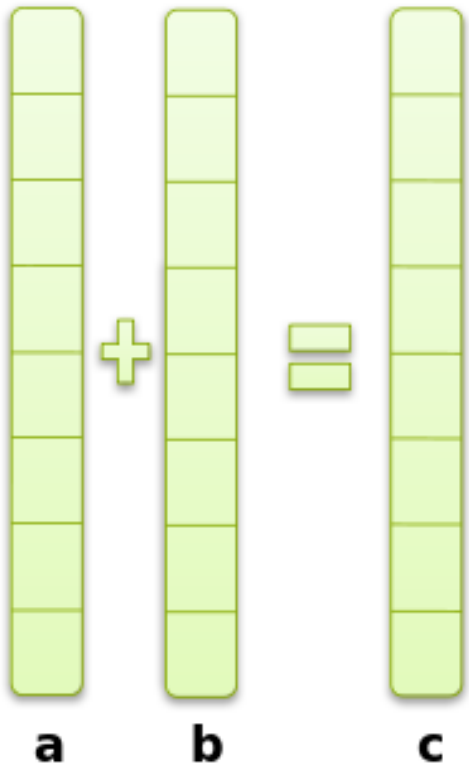
Parallel Programming in CUDA C/C++

- But wait ... GPU computing is about massive parallelism!
- We need a more interesting example ...
- We'll start by adding two integers and build up to vector addition

CUDA

Parallel Programming in CUDA C/C++

- But wait ... GPU computing is about massive parallelism!
- We need a more interesting example ...
- We'll start by adding two integers and build up to vector addition



CUDA: First Program

```
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

CUDA: First Program

```
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- As before __global__ is a CUDA C/C++ keyword meaning

CUDA: First Program

```
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- As before __global__ is a CUDA C/C++ keyword meaning
  - add() will execute on the device

CUDA: First Program

```
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

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

CUDA: First Program

```
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

CUDA: First Program

```
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- add() run on the device, so a,b and c must point to device memory

CUDA: First Program

```
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- add() run on the device, so a,b and c must point to device memory
- we need to allocate memory on GPU

CUDA: Memory management

- Host and device memory are separate entities

CUDA: Memory management

- Host and device memory are separate entities
  - Device pointers point to GPU memory

CUDA: Memory management

- Host and device memory are separate entities
  - Device pointers point to GPU memory
    - ★ May be passed to/from host code

CUDA: Memory management

- Host and device memory are separate entities
  - Device pointers point to GPU memory
    - ★ May be passed to/from host code
    - ★ May not be dereferenced in host code

CUDA: Memory management

- Host and device memory are separate entities
  - Device pointers point to GPU memory
    - ★ May be passed to/from host code
    - ★ May not be dereferenced in host code
  - Host pointers point to CPU memory

CUDA: Memory management

- Host and device memory are separate entities
  - Device pointers point to GPU memory
    - ★ May be passed to/from host code
    - ★ May not be dereferenced in host code
  - Host pointers point to CPU memory
    - ★ May be passed to/from device code

CUDA: Memory management

- Host and device memory are separate entities
  - Device pointers point to GPU memory
    - ★ May be passed to/from host code
    - ★ May not be dereferenced in host code
  - Host pointers point to CPU memory
    - ★ May be passed to/from device code
    - ★ May not be dereferenced in device code

CUDA: Memory management

- Host and device memory are separate entities
  - Device pointers point to GPU memory
    - ★ May be passed to/from host code
    - ★ May not be dereferenced in host code
  - Host pointers point to CPU memory
    - ★ May be passed to/from device code
    - ★ May not be dereferenced in device code
- Simple CUDA API for handling device memory

CUDA: Memory management

- Host and device memory are separate entities
  - Device pointers point to GPU memory
    - ★ May be passed to/from host code
    - ★ May not be dereferenced in host code
  - Host pointers point to CPU memory
    - ★ May be passed to/from device code
    - ★ May not be dereferenced in device code
- Simple CUDA API for handling device memory
  - cudaMalloc(), cudaFree(), cudaMemcpy()

CUDA: Memory management

- Host and device memory are separate entities
  - Device pointers point to GPU memory
    - ★ May be passed to/from host code
    - ★ May not be dereferenced in host code
  - Host pointers point to CPU memory
    - ★ May be passed to/from device code
    - ★ May not be dereferenced in device code
- Simple CUDA API for handling device memory
  - cudaMalloc(), cudaFree(), cudaMemcpy()
  - Similar to the C equivalents malloc(), free(), memcpy()

CUDA: First Program

```
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- Let's take a look at main()

CUDA: First Program

```
#include <stdio.h>

__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}

int main(void) {
    int h_a, h_b, h_c; // host copies of a,b,c
    int *d_a, *d_b, *d_c; //device copie of a,b,c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

    // Setup input values
        h_a = 2;
        h_b = 7;
```

CUDA: First Program

```
    // Copy inputs to device
    cudaMemcpy(d_a, &h_a, size,
        cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &h_b, size,
        cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&h_c, d_c, size,
        cudaMemcpyDeviceToHost);

    printf("Result = %d\n",h_c);

    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

CUDA

Let's test the code addseq.cu with measuring performance

CUDA: Unified Memory (SCC K40m and P100)

- Unified memory was added in cuda/6.0.

CUDA: Unified Memory (SCC K40m and P100)

- Unified memory was added in cuda/6.0.
- Only supported on the SCC when using the K40m or P100 GPUs

CUDA: Unified Memory (SCC K40m and P100)

- Unified memory was added in cuda/6.0.
- Only supported on the SCC when using the K40m or P100 GPUs
- With Unified Memory the CUDA driver will manage memory transfers using the cudaMallocManaged() function.

CUDA: Unified Memory (SCC K40m and P100)

- Unified memory was added in cuda/6.0.
- Only supported on the SCC when using the K40m or P100 GPUs
- With Unified Memory the CUDA driver will manage memory transfers using the cudaMallocManaged() function.
- Managed memory is still freed using cudaFree()

CUDA: Unified Memory (SCC K40m and P100)

- Unified memory was added in cuda/6.0.
- Only supported on the SCC when using the K40m or P100 GPUs
- With Unified Memory the CUDA driver will manage memory transfers using the cudaMallocManaged() function.
- Managed memory is still freed using cudaFree()
- The P100 will offer the best performance when using this feature.

CUDA: Unified Memory (SCC K40m and P100)

- Unified memory was added in cuda/6.0.
- Only supported on the SCC when using the K40m or P100 GPUs
- With Unified Memory the CUDA driver will manage memory transfers using the cudaMallocManaged() function.
- Managed memory is still freed using cudaFree()
- The P100 will offer the best performance when using this feature.
- Unified Memory simplifies memory management in a CUDA code.

CUDA: Unified Memory (SCC K40m and P100)

- Unified memory was added in cuda/6.0.
- Only supported on the SCC when using the K40m or P100 GPUs
- With Unified Memory the CUDA driver will manage memory transfers using the cudaMallocManaged() function.
- Managed memory is still freed using cudaFree()
- The P100 will offer the best performance when using this feature.
- Unified Memory simplifies memory management in a CUDA code.
- For more details see: `https://devblogs.nvidia.com/unified-memory-cuda-beg`

CUDA: First Program

```
include <stdio.h>

__global__ void add(int *a, int *b, int *c) {*c = *a +
    *b;}

int main(void) {
    int *a, *b, *c;  // host AND device
    int size = sizeof(int);
        // Allocate space for device copies of a, b, c
        cudaMallocManaged(&a, size);
            cudaMallocManaged(&b, size);
        cudaMallocManaged(&c, size);
        // Setup input values
        *a = 2; *b = 7;
    // Launch add() kernel on GPU. Data values are
    // sent to the host when accessed in the kernel
    add<<<1,1>>>(a,b,c);
    // Wait for GPU to finish before accessing on host
    cudaDeviceSynchronize();
    // access will auto-transfer data back to the host
    printf("%d %d %d\n",*a, *b, *c);
    cudaFree(a); cudaFree(b); cudaFree(c); // Cleanup
    return 0;
}
```

CUDA

Let's test the code addseqm.cu with measuring performance - compare with addseq.cu

CUDA: Moving to parallel

GPU computing is about massive parallelism

- So how do we run code in parallel on the device?

CUDA: Moving to parallel

GPU computing is about massive parallelism

- So how do we run code in parallel on the device?

```
add<<<1,1>>>();
```

CUDA: Moving to parallel

GPU computing is about massive parallelism

- So how do we run code in parallel on the device?

```
add<<<1,1>>>();
```

```
add<<<N,1>>>();
```

CUDA: Moving to parallel

GPU computing is about massive parallelism

- So how do we run code in parallel on the device?

```
add<<<1,1>>>();
```

```
add<<<N,1>>>();
```

- Instead of executing add() once, execute N times in parallel !!

CUDA: Vector Addition on the Device

- With add() running in parallel we can do vector addition

CUDA: Vector Addition on the Device

- With add() running in parallel we can do vector addition
- Terminology: each parallel invocation of add() is referred to as a block

CUDA: Vector Addition on the Device

- With add() running in parallel we can do vector addition
- Terminology: each parallel invocation of add() is referred to as a block
  - The set of blocks is referred to as a grid

CUDA: Vector Addition on the Device

- With add() running in parallel we can do vector addition
- Terminology: each parallel invocation of add() is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using blockIdx.x

CUDA: Vector Addition on the Device

- With add() running in parallel we can do vector addition
- Terminology: each parallel invocation of add() is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using blockIdx.x

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

CUDA: Vector Addition on the Device

- With add() running in parallel we can do vector addition
- Terminology: each parallel invocation of add() is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using blockIdx.x

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

- By using blockIdx.x to index into the array, each block handles a different index

CUDA: Vector Addition on the Device

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

CUDA: Vector Addition on the Device

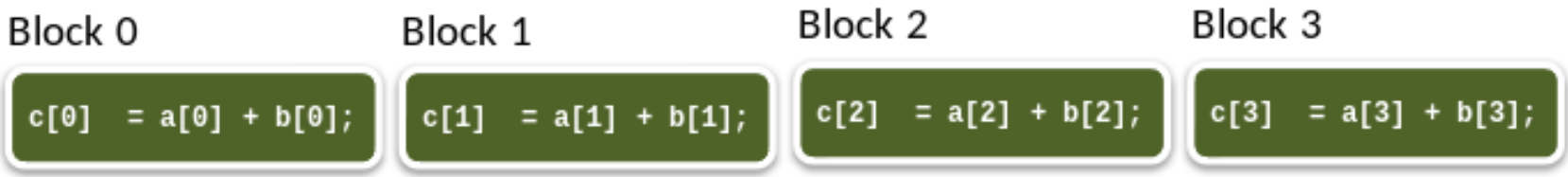
```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] +
            b[blockIdx.x];
}
```

- On the device, each block can execute in parallel:

CUDA: Vector Addition on the Device

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

- On the device, each block can execute in parallel:



CUDA: Vector Addition on the Device

- Returning to our parallelized add() kernel

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

CUDA: Vector Addition on the Device

- Returning to our parallelized add() kernel

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

- Let's take a look at main() ...

CUDA: Vector Addition on the Device

```
#define N 512
int main(void) {
    int *h_a, *h_b, *h_c;    // host copies of a, b, c
    int *d_a, *d_b, *d_c;    // device copies of a,
        b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and
        setup input values
    h_a = (int *)malloc(size); random_ints(h_a, N);
    h_b = (int *)malloc(size); random_ints(h_b, N);
    h_c = (int *)malloc(size);
```

CUDA: Vector Addition on the Device

```
// Copy inputs to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
```

CUDA: Vector Addition on the Device

```
// Copy inputs to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);
```

CUDA: Vector Addition on the Device

```
    // Copy inputs to device
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(h_a); free(h_b); free(h_c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

CUDA

Let's test the code vecaddpar.cu with measuring performance

```
void random_ints(int* a, int N)
{
        int i;
        for (i = 0; i < N; ++i)
        a[i] = rand();
}
```

CUDA

- Exit or Ctrl-D
- compute –mem 2 –gpu

Let's test the code vecaddpar.cu with measuring performance with N=1024, 10240, 102400, 1024000, 10240000 (user argv)

CUDA: Review

- Difference between host and device

CUDA: Review

- Difference between host and device
  - Host CPU

CUDA: Review

- Difference between host and device
  - Host CPU
  - Device GPU

CUDA: Review

- Difference between host and device
  - Host CPU
  - Device GPU
- Using __global__ to declare a function as device code

CUDA: Review

- Difference between host and device
  - Host CPU
  - Device GPU
- Using __global__ to declare a function as device code
  - Executes on the device

CUDA: Review

- Difference between host and device
  - Host CPU
  - Device GPU
- Using __global__ to declare a function as device code
  - Executes on the device
  - Called from the host

CUDA: Review

- Difference between host and device
  - Host CPU
  - Device GPU
- Using __global__ to declare a function as device code
  - Executes on the device
  - Called from the host
- Passing parameters from host code to a device function

CUDA: Review

- Basic device memory management

CUDA: Review

- Basic device memory management
  - cudaMalloc()
  - cudaMemcpy()
  - cudaFree()

CUDA: Review

- Basic device memory management
  - cudaMalloc()
  - cudaMemcpy()
  - cudaFree()
- Launching parallel kernels

CUDA: Review

- Basic device memory management
  - cudaMalloc()
  - cudaMemcpy()
  - cudaFree()
- Launching parallel kernels
  - Launch N copies of add() with add$<<< N, 1 >>>$(...);

CUDA: Review

- Basic device memory management
  - cudaMalloc()
  - cudaMemcpy()
  - cudaFree()
- Launching parallel kernels
  - Launch N copies of add() with add$<<< N, 1 >>>$(...);
  - Use blockIdx.x to access block index

CUDA: Threads

- Terminology: a block can be split into parallel threads

CUDA: Threads

- Terminology: a block can be split into parallel threads
- Let's change add() to use parallel threads instead of parallel blocks

CUDA: Threads

- Terminology: a block can be split into parallel threads
- Let's change add() to use parallel threads instead of parallel blocks

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

CUDA: Threads

- Terminology: a block can be split into parallel threads
- Let's change add() to use parallel threads instead of parallel blocks

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

- We use threadIdx.x instead of blockIdx.x

CUDA: Threads

- Terminology: a block can be split into parallel threads
- Let's change add() to use parallel threads instead of parallel blocks

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

- We use threadIdx.x instead of blockIdx.x
- Need to make one change in main() ...

CUDA: Vector Addition on the Device

```
#define N 512
int main(void) {
    int *h_a *h_b, *h_c;     // host copies of a, b, c
    int *d_a, *d_b, *d_c;    // device copies of a,
        b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and
        setup input values
    h_a = (int *)malloc(size); random_ints(h_a, N);
    h_b = (int *)malloc(size); random_ints(h_b, N);
    h_c = (int *)malloc(size);
```

CUDA: Vector Addition on the Device

```
// Copy inputs to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
```

CUDA: Vector Addition on the Device

```
// Copy inputs to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);
```

CUDA: Vector Addition on the Device

```
    // Copy inputs to device
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(h_a); free(h_b); free(h_c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

CUDA

Let's test the code vecaddpartargt.cu with measuring performance with N=1024, 10240, 102400, 1024000, 10240000 (user argv)

CUDA: Combining Blocks and Threads

- We've seen parallel vector addition using:
  - Many blocks with one thread each
  - One block with many threads
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that ...
- Use blockIdx.x to access block index
- First let's discuss data indexing ...

CUDA: Indexing Arrays with Blocks and Threads

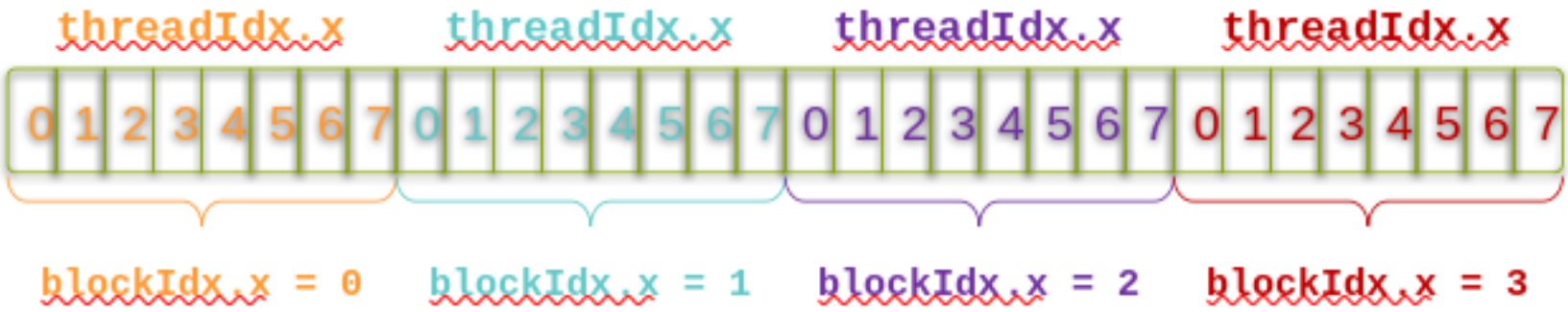
- No longer as simple as using blockIdx.x and threadIdx.x

CUDA: Indexing Arrays with Blocks and Threads

- No longer as simple as using blockIdx.x and threadIdx.x
  - Consider indexing an array with one element per thread (8 threads/block)

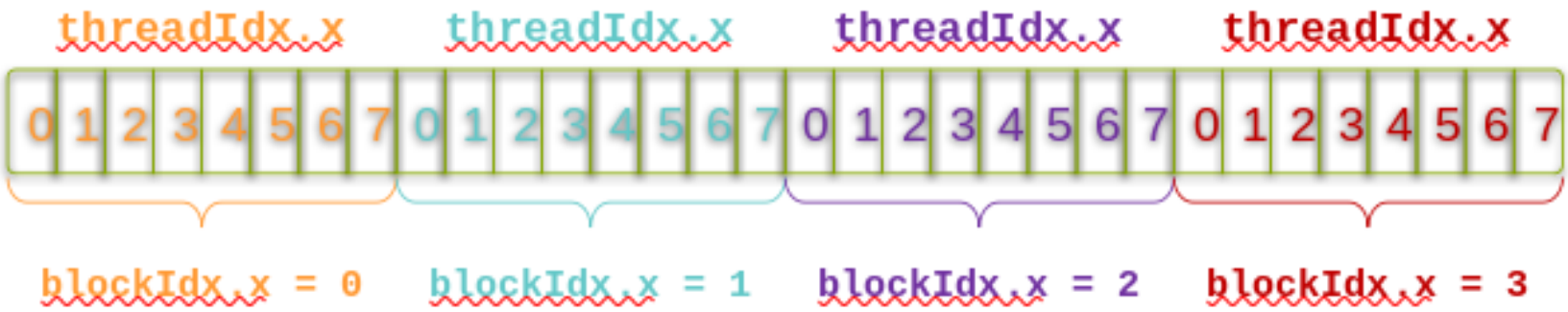
CUDA: Indexing Arrays with Blocks and Threads

- No longer as simple as using blockIdx.x and threadIdx.x
  - Consider indexing an array with one element per thread (8 threads/block)



CUDA: Indexing Arrays with Blocks and Threads

- No longer as simple as using blockIdx.x and threadIdx.x
  - Consider indexing an array with one element per thread (8 threads/block)



- With M threads/block a unique index for each thread is given by:

CUDA: Indexing Arrays with Blocks and Threads

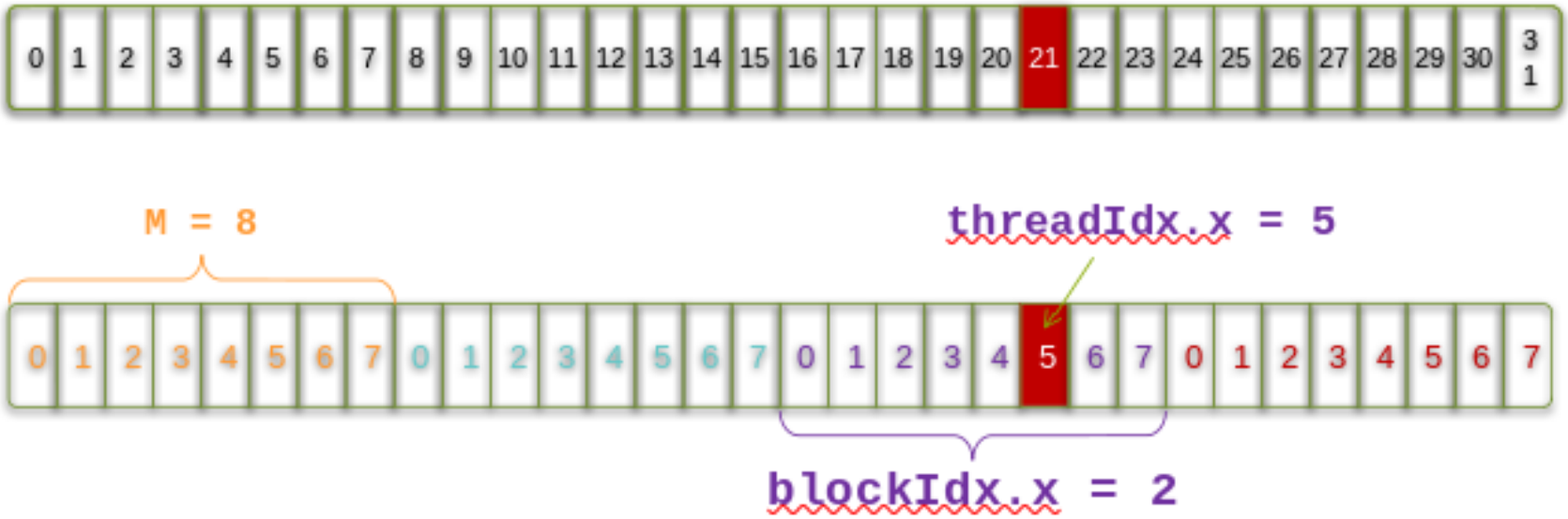
- No longer as simple as using blockIdx.x and threadIdx.x
  - Consider indexing an array with one element per thread (8 threads/block)

- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

CUDA: Indexing Arrays - Example

- Which thread will operate on the red element?



- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
          = 5           +     2      * 8;
          = 21;
```

CUDA: Indexing Arrays with Blocks and Threads

- Use the built-in variable blockDim.x for threads per block

CUDA: Indexing Arrays with Blocks and Threads

- Use the built-in variable blockDim.x for threads per block

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

CUDA: Indexing Arrays with Blocks and Threads

- Use the built-in variable blockDim.x for threads per block

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

- Combined version of add() to use parallel threads and parallel blocks

CUDA: Indexing Arrays with Blocks and Threads

- Use the built-in variable blockDim.x for threads per block

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

- Combined version of add() to use parallel threads and parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
int index = threadIdx.x + blockIdx.x *
    blockDim.x;
c[index] = a[index] + b[index];
}
```

CUDA: Indexing Arrays with Blocks and Threads

- Use the built-in variable blockDim.x for threads per block

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

- Combined version of add() to use parallel threads and parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
int index = threadIdx.x + blockIdx.x *
    blockDim.x;
c[index] = a[index] + b[index];
}
```

- What changes need to be made in main()?

CUDA: Addition with Blocks and Threads

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *h_a, *h_b, *h_c;                    // host copies
        of a, b, c
    int *d_a, *d_b, *d_c;                    // device copies
        of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and
        setup input values
    h_a = (int *)malloc(size); random_ints(h_a, N);
    h_b = (int *)malloc(size); random_ints(h_b, N);
    h_c = (int *)malloc(size);
```

CUDA: Vector Addition on the Device

```
// Copy inputs to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);
```

CUDA: Vector Addition on the Device

```
// Copy inputs to device
cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a,
    d_b, d_c);
```

CUDA: Vector Addition on the Device

```
    // Copy inputs to device
    cudaMemcpy(d_a, h_a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, h_b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a,
        d_b, d_c);

    // Copy result back to host
    cudaMemcpy(h_c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(h_a); free(h_b); free(h_c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

CUDA

Let's test the code vecaddpartargbt.cu with measuring performance with N=1024, 10240, 102400, 1024000, 10240000 (user argv)

CUDA: Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of blockDim.x

CUDA: Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of blockDim.x
- Avoid accessing beyond the end of the arrays:

CUDA: Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of blockDim.x
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int
    n) {
int index = threadIdx.x + blockIdx.x *
    blockDim.x;
if (index < n)
    c[index] = a[index] + b[index];
}
```

CUDA: Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of blockDim.x
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int
    n) {
int index = threadIdx.x + blockIdx.x *
    blockDim.x;
if (index < n)
    c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

CUDA: Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of blockDim.x
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int
    n) {
int index = threadIdx.x + blockIdx.x *
    blockDim.x;
if (index < n)
    c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

CUDA: Why Bother with Threads?

- Threads seem unnecessary

CUDA: Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity

CUDA: Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?

CUDA: Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:

CUDA: Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate

CUDA: Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize

CUDA: Why Bother with Threads?

- Threads seem unnecessary
  - They add a level of complexity
  - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
  - Communicate
  - Synchronize
- To look closer, we need a new example ...

CUDA: Review

- Launching parallel kernels

CUDA: Review

- Launching parallel kernels
  - Launch N copies of add() with add$<<< N/M, M >>>$(...);

CUDA: Review

- Launching parallel kernels
  - Launch N copies of add() with add$<<< N/M, M >>>$(...);
  - Use blockIdx.x to access block index

CUDA: Review

- Launching parallel kernels
  - Launch N copies of add() with add$<<< N/M, M >>>$(...);
  - Use blockIdx.x to access block index
  - Use threadIdx.x to access thread index within block

CUDA: Review

- Launching parallel kernels
  - Launch N copies of add() with add$<<<N/M, M>>>$(...);
  - Use blockIdx.x to access block index
  - Use threadIdx.x to access thread index within block
- Allocate elements to threads:

CUDA: Review

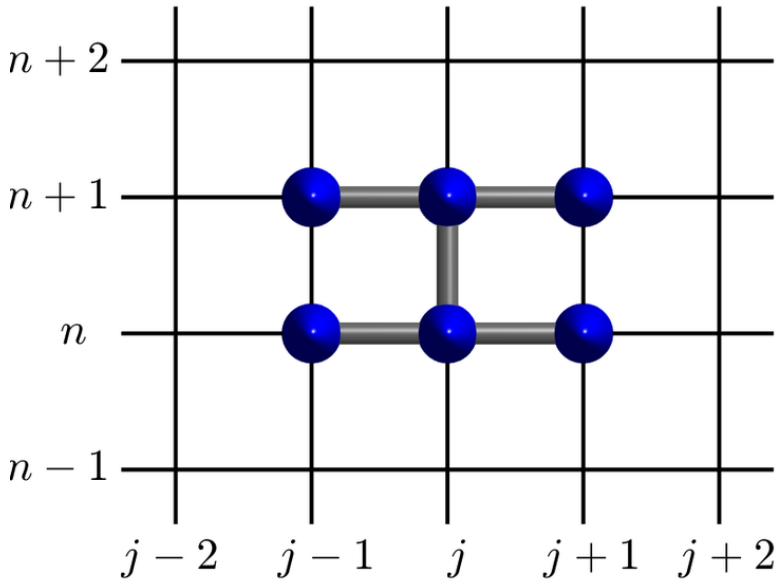
- Launching parallel kernels
  - Launch N copies of add() with add$<<< N/M, M >>>$(...);
  - Use blockIdx.x to access block index
  - Use threadIdx.x to access thread index within block
- Allocate elements to threads:

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

CUDA: Cooperating Threads

1D Stencil :

- In mathematics, especially the areas of numerical analysis concentrating on the numerical solution of partial differential equations, a stencil is a geometric arrangement of a nodal group that relate to the point of interest by using a numerical approximation routine.
- Stencils are the basis for many algorithms to numerically solve partial differential equations (PDE).



CUDA: Cooperating Threads

CUDA: Cooperating Threads

1D Stencil :

CUDA: Cooperating Threads

1D Stencil :

- In mathematics, especially the areas of numerical analysis concentrating on the numerical solution of partial differential equations, a stencil is a geometric arrangement of a nodal group that relate to the point of interest by using a numerical approximation routine.

CUDA: Cooperating Threads

1D Stencil :

- In mathematics, especially the areas of numerical analysis concentrating on the numerical solution of partial differential equations, a stencil is a geometric arrangement of a nodal group that relate to the point of interest by using a numerical approximation routine.
- Stencils are the basis for many algorithms to numerically solve partial differential equations (PDE).

CUDA: Cooperating Threads

1D Stencil :

- Consider applying a 1D stencil to a 1D array of elements

CUDA: Cooperating Threads

1D Stencil :

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius

CUDA: Cooperating Threads

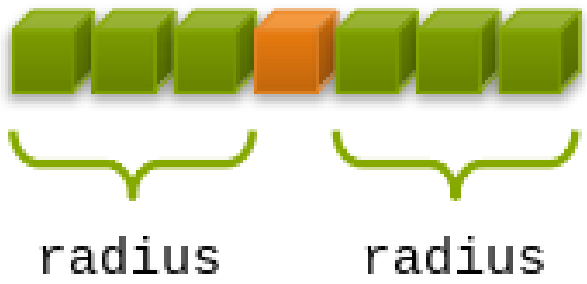
1D Stencil :

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:

CUDA: Cooperating Threads

1D Stencil :

- Consider applying a 1D stencil to a 1D array of elements
  - Each output element is the sum of input elements within a radius
- If radius is 3, then each output element is the sum of 7 input elements:



CUDA: Implementing Within a Block

1D Stencil :

- Each thread processes one output element

CUDA: Implementing Within a Block

1D Stencil :

- Each thread processes one output element
  - blockDim.x elements per block

CUDA: Implementing Within a Block

1D Stencil :

- Each thread processes one output element
  - blockDim.x elements per block
- Input elements are read several times

CUDA: Implementing Within a Block

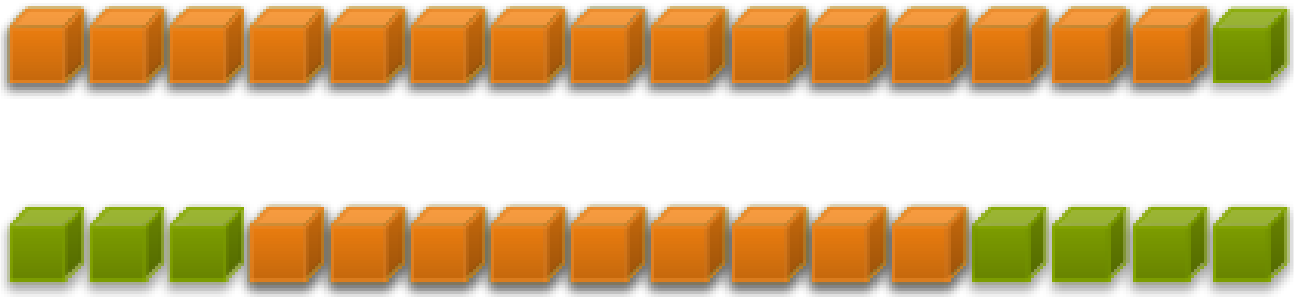
1D Stencil :

- Each thread processes one output element
  - blockDim.x elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times

CUDA: Implementing Within a Block

1D Stencil :

- Each thread processes one output element
  - blockDim.x elements per block
- Input elements are read several times
  - With radius 3, each input element is read seven times



CUDA: Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory

CUDA: Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed

CUDA: Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using __shared__, allocated per block

CUDA: Sharing Data Between Threads

- Terminology: within a block, threads share data via shared memory
- Extremely fast on-chip memory, user-managed
- Declare using __shared__, allocated per block
- Data is not visible to threads in other blocks

CUDA: Implementing With Shared Memory

- Cache data in shared memory

CUDA: Implementing With Shared Memory

- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory

CUDA: Implementing With Shared Memory

- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements

CUDA: Implementing With Shared Memory

- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory

CUDA: Implementing With Shared Memory

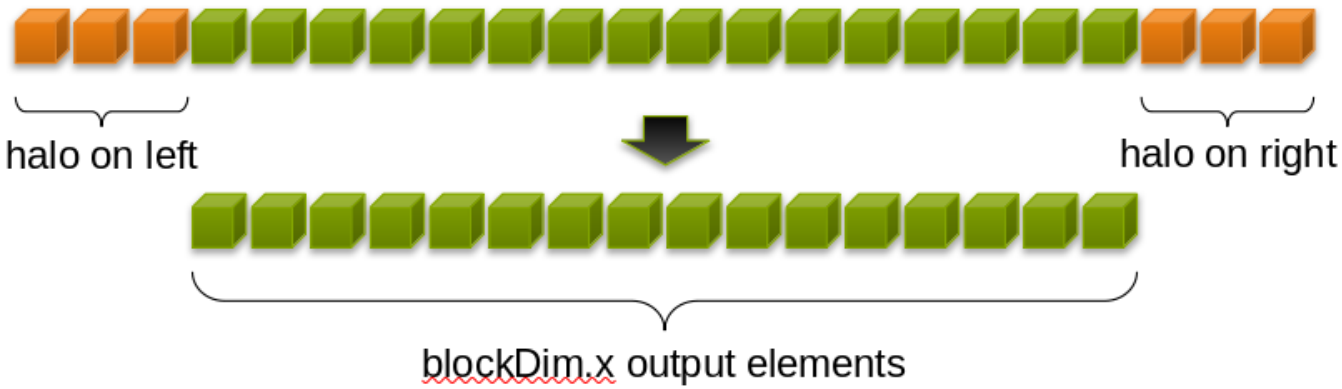
- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory
  - Input elements are read several times

CUDA: Implementing With Shared Memory

- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory
  - Input elements are read several times
  - Each block needs a halo of radius elements at each boundary

CUDA: Implementing With Shared Memory

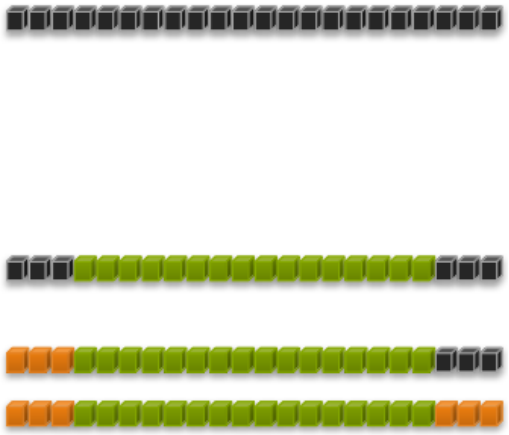
- Cache data in shared memory
  - Read (blockDim.x + 2 * radius) input elements from global memory to shared memory
  - Compute blockDim.x output elements
  - Write blockDim.x output elements to global memory
  - Input elements are read several times
  - Each block needs a halo of radius elements at each boundary



CUDA: Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
__shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
int gindex = threadIdx.x + blockIdx.x * blockDim.x;
int lindex = threadIdx.x + RADIUS;

// Read input elements into shared memory
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex - RADIUS] = in[gindex - RADIUS];
  temp(lindex + BLOCK_SIZE] =
    in[gindex + BLOCK_SIZE];
```



CUDA: Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ;
         offset++)
      result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

CUDA: Data Race!

- The stencil example will not work ...
- Suppose thread 15 reads the halo before thread 0 has fetched it

```
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
  temp[lindex – RADIUS = in[gindex – RADIUS];
  temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
}

int result = 0;
result += temp[lindex + 1];
```

Store at temp[18]

Skipped, threadIdx > RADIUS

Load from temp[19]

CUDA: Synchronize

- void __syncthreads();
- Synchronizes all threads within a block
  - Used to prevent RAW (Read after Write), WAR (Write after Read), WAW (Write after Write) hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

CUDA: Synchronize

- void __syncthreads();
- Synchronizes all threads within a block
  - Used to prevent RAW (Read after Write), WAR (Write after Read), WAW (Write after Write) hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

CUDA: Synchronize

- void __syncthreads();

CUDA: Synchronize

- void __syncthreads();
- Synchronizes all threads within a block

CUDA: Synchronize

- void __syncthreads();
- Synchronizes all threads within a block
  - Used to prevent RAW (Read after Write), WAR (Write after Read), WAW (Write after Write) hazards

CUDA: Synchronize

- void __syncthreads();
- Synchronizes all threads within a block
  - Used to prevent RAW (Read after Write), WAR (Write after Read), WAW (Write after Write) hazards
- All threads must reach the barrier

CUDA: Synchronize

- void __syncthreads();
- Synchronizes all threads within a block
  - Used to prevent RAW (Read after Write), WAR (Write after Read), WAW (Write after Write) hazards
- All threads must reach the barrier
  - In conditional code, the condition must be uniform across the block

CUDA: Stencil Kernel

```
__global__ void stencil_1d(int *in, int *out) {
__shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
int gindex = threadIdx.x + blockIdx.x * blockDim.x;
int lindex = threadIdx.x + radius;

// Read input elements into shared memory
temp[lindex] = in[gindex];
if (threadIdx.x < RADIUS) {
    temp[lindex - RADIUS] = in[gindex - RADIUS];
    temp[lindex + BLOCK_SIZE] = in[gindex +
        BLOCK_SIZE];
}

// Synchronize (ensure all the data is available)
__syncthreads();
```

CUDA: Stencil Kernel

```
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ;
         offset++)
         result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}
```

CUDA: Review

- Launching parallel kernels

CUDA: Review

- Launching parallel kernels
  - Launch N blocks with M threads per block with kernel<<<N, M>>>(...);

CUDA: Review

- Launching parallel kernels
  - Launch N blocks with M threads per block with kernel<<<N, M>>>(...);
  - Use blockIdx.x to access block index within grid

CUDA: Review

- Launching parallel kernels
  - Launch N blocks with M threads per block with kernel<<<N, M>>>(...);
  - Use blockIdx.x to access block index within grid
  - Use threadIdx.x to access thread index within block

CUDA: Review

- Launching parallel kernels
  - Launch N blocks with M threads per block with kernel$<<<N, M>>>(...)$;
  - Use blockIdx.x to access block index within grid
  - Use threadIdx.x to access thread index within block
- Allocate elements to threads:

CUDA: Review

- Launching parallel kernels
  - Launch N blocks with M threads per block with kernel<<<N, M>>>(...);
  - Use blockIdx.x to access block index within grid
  - Use threadIdx.x to access thread index within block
- Allocate elements to threads:

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

CUDA: Review

- Use __shared__ to declare a variable/array in shared memory

CUDA: Review

- Use __shared__ to declare a variable/array in shared memory
  - Data is shared between threads in a block

CUDA: Review

- Use __shared__ to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks

CUDA: Review

- Use __shared__ to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use __syncthreads() as a barrier

CUDA: Review

- Use __shared__ to declare a variable/array in shared memory
  - Data is shared between threads in a block
  - Not visible to threads in other blocks
- Use __syncthreads() as a barrier
  - Use to prevent data hazards

CUDA: Managing the device

Coordinating Host & Device

CUDA: Managing the device

Coordinating Host & Device

- Kernel launches are asynchronous

CUDA: Managing the device

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately

CUDA: Managing the device

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

CUDA: Managing the device

Coordinating Host & Device

- Kernel launches are asynchronous
  - Control returns to the CPU immediately
- CPU needs to synchronize before consuming the results

`cudaMemcpy()`	Blocks the CPU until the copy is complete Copy begins when all preceding CUDA calls have completed
`cudaMemcpyAsync()`	Asynchronous, does not block the CPU
`cudaDeviceSynchronize()`	Blocks the CPU until all preceding CUDA calls have completed

CUDA: Managing the device

Reporting Errors

CUDA: Managing the device

Reporting Errors

- All CUDA API calls return an error code (cudaError_t)

CUDA: Managing the device

Reporting Errors

- All CUDA API calls return an error code (cudaError_t)
  - Error in the API call itself

CUDA: Managing the device

Reporting Errors

- All CUDA API calls return an error code (cudaError_t)
  - Error in the API call itself
  - OR

CUDA: Managing the device

Reporting Errors

- All CUDA API calls return an error code (cudaError_t)
  - Error in the API call itself
  - OR
  - Error in an earlier asynchronous operation (e.g. kernel)

CUDA: Managing the device

Reporting Errors

- All CUDA API calls return an error code (cudaError_t)
  - Error in the API call itself
  - OR
  - Error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error:

CUDA: Managing the device

Reporting Errors

- All CUDA API calls return an error code (cudaError_t)
  - Error in the API call itself
  - OR
  - Error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

CUDA: Managing the device

Reporting Errors

- All CUDA API calls return an error code (cudaError_t)
  - Error in the API call itself
  - OR
  - Error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

CUDA: Managing the device

Reporting Errors

- All CUDA API calls return an error code (cudaError_t)
  - Error in the API call itself
  - OR
  - Error in an earlier asynchronous operation (e.g. kernel)
- Get the error code for the last error:

```
cudaError_t cudaGetLastError(void)
```

- Get a string to describe the error:

```
cudaError_t cudaGetLastError(void)

printf("%s\n",
    cudaGetErrorString(cudaGetLastError()));
```

CUDA: Managing the device

Device Management

CUDA: Managing the device

Device Management

- Application can query and select GPUs

CUDA: Managing the device

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp
    *prop, int device)
```

CUDA: Managing the device

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp
    *prop, int device)
```

- Multiple threads can share a device

CUDA: Managing the device

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp
    *prop, int device)
```

- Multiple threads can share a device
- A single thread can manage multiple devices

CUDA: Managing the device

Device Management

- Application can query and select GPUs

```
cudaGetDeviceCount(int *count)
cudaSetDevice(int device)
cudaGetDevice(int *device)
cudaGetDeviceProperties(cudaDeviceProp
    *prop, int device)
```

- Multiple threads can share a device
- A single thread can manage multiple devices

```
cudaSetDevice(i) to select current device
cudaMemcpy(...) for peer-to-peer copies
    (requires OS and device support)
```

CUDA: Review

- What have we learned?

CUDA: Review

- What have we learned?
  - Write and launch CUDA C/C++ kernels

CUDA: Review

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - ★ __global__, blockIdx.x, threadIdx.x, <<<>>>

CUDA: Review

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - ★ __global__, blockIdx.x, threadIdx.x, <<<>>>
  - Manage GPU memory

CUDA: Review

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - ★ __global__, blockIdx.x, threadIdx.x, <<<>>>
  - Manage GPU memory
    - ★ cudaMalloc(), cudaMemcpy(), cudaFree()

CUDA: Review

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - ★ __global__, blockIdx.x, threadIdx.x, <<<>>>
  - Manage GPU memory
    - ★ cudaMalloc(), cudaMemcpy(), cudaFree()
  - Manage communication and synchronization

CUDA: Review

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - ★ __global__, blockIdx.x, threadIdx.x, <<<>>>
  - Manage GPU memory
    - ★ cudaMalloc(), cudaMemcpy(), cudaFree()
  - Manage communication and synchronization
    - ★ __shared__, __syncthreads()

CUDA: Review

- What have we learned?
  - Write and launch CUDA C/C++ kernels
    - ★ __global__, blockIdx.x, threadIdx.x, <<<>>>
  - Manage GPU memory
    - ★ cudaMalloc(), cudaMemcpy(), cudaFree()
  - Manage communication and synchronization
    - ★ __shared__, __syncthreads()
    - ★ cudaMemcpy() vs cudaMemcpyAsync(), cudaDeviceSynchronize()

CUDA: Compute Capability

- The compute capability of a device describes its architecture, e.g.
- Number of registers
- Sizes of memories
- Features & capabilities
- For an update-to-date list see Wikipedia:
- `https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications`
- Examples GPUs:

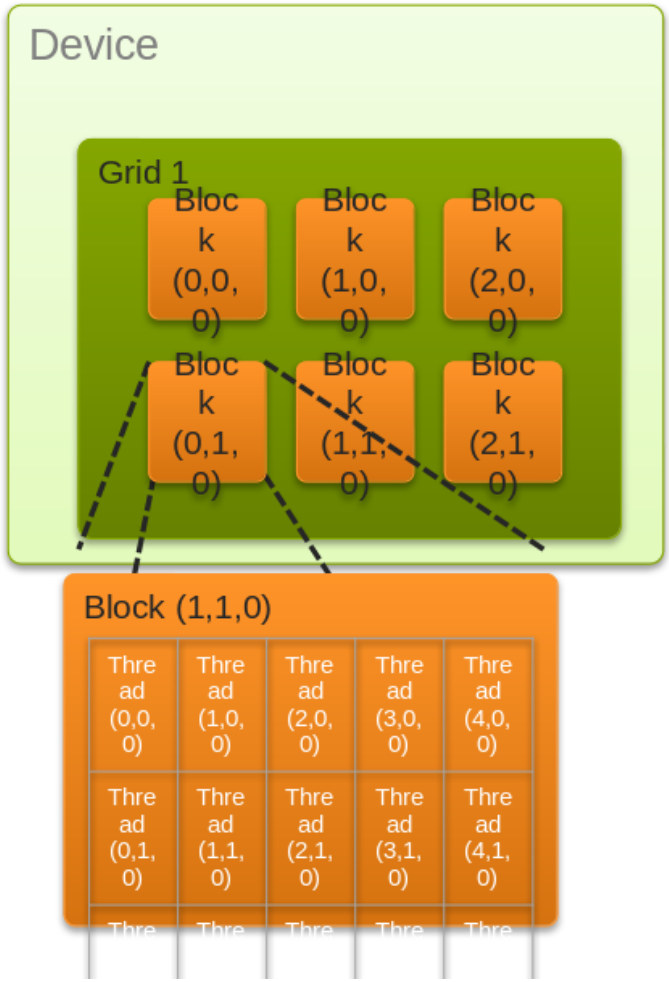
GPU	Compute Capability
M2050	2.0
M2070	2.0
K40m	3.5
P100	6.0

- Features & capabilities

This lesson has concentrated on Fermi devices with Compute Capability >= 2.0

CUDA: Stencil Kernel

- A kernel is launched as a grid of blocks of threads
  - blockIdx and threadIdx are 3D
  - We showed only one dimension (x)
- Built-in variables:
  - threadIdx
  - blockIdx
  - blockDim
  - gridDim



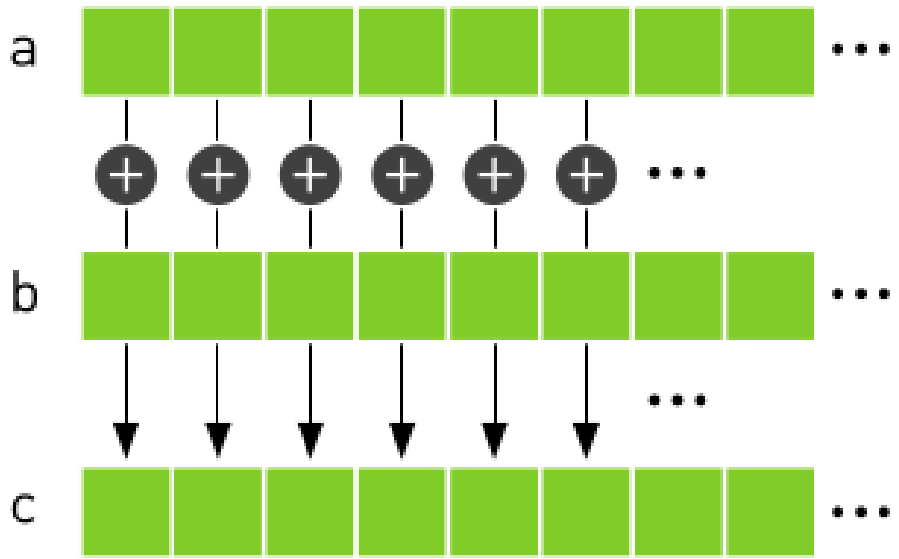
CUDA Parallel Programming

CUDA-Reminders

- Previously, we saw how easy it was to get a standard C function to start running on a device.
- By adding the __global__ qualifier to the function and by calling it using a special angle bracket syntax, we executed the function on a GPU.
- This was extremely simple but it can be also extremely inefficient.
- Why ? Because NVIDIA's hardware engineering have optimized graphics processors to perform hundreds of computations in parallel.
- However, thus far we have only ever launched a kernel that runs serially on the GPU.

CUDA-Parallel Programming

Summing vectors



CUDA-Parallel Programming

```
#define N 10
void add( int *a, int *b, int *c ) {
    for (i=0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}

int main( void ) {
    int a[N], b[N], c[N];
    // fill the arrays 'a' and 'b'
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
    add( a, b, c );
    // display the results
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
    return 0;
}
```

CUDA-Parallel Programming

It's work on only one CPU

```
void add( int *a, int *b, int *c ) {
    for (i=0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}
```

CUDA-Parallel Programming

We propose to modify the function in the following way as follows

```
void add( int *a, int *b, int *c ) {
    int tid = 0; // ???
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1; // ???
    }
}
```

CUDA-Parallel Programming

We compute the sum within a while loop where the index tid ranges from 0 to N-1 (processors number).

```
void add( int *a, int *b, int *c ) {
    int tid = 0; // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1; // ???
    }
}
```

CUDA-Parallel Programming

```
void add( int *a, int *b, int *c ) {
    int tid = 0; // this is CPU zero, so we start at zero
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1; // we have one CPU, so we increment by
            one
    }
}
```

CUDA-Parallel Programming

For example, with a dual-core processor, one could change the increment to 2 and have one core initialize the loop with tid = 0 and another with tid = 1. The first core would add the even-indexed elements, and the second core would add the odd- indexed elements.

CPU CORE 1

```
void add( int *a, int *b, int *c )
{
    int tid = 0;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

CPU CORE 2

```
void add( int *a, int *b, int *c )
{
    int tid = 1;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 2;
    }
}
```

CUDA-Parallel Programming

- It is possible to make the same addition very similarly on a GPU by writing add() as a device function.
- This should look similar to previous code.

CUDA-Parallel Programming

```
define N 10
int main( void ) {
    int a[N], b[N], c[N], *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    cudaMalloc((void**)&dev_a, N * sizeof(int));
    cudaMalloc((void**)&dev_b, N * sizeof(int));
    cudaMalloc((void**)&dev_c, N * sizeof(int));
    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) { a[i] = -i;  b[i] = i * i; }
    // copy the arrays 'a' and 'b' to the GPU
    cudaMemcpy(dev_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b,b,N*sizeof(int),cudaMemcpyHostToDevice);

    add<<<N,1>>>( dev_a, dev_b, dev_c );

    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy(c,dev_c,N*sizeof(int),cudaMemcpyDeviceToHost);
    // display the results
    for (int i=0; i<N; i++) { printf( "%d + %d = %d\n", a[i], b
    // free the memory allocated on the GPU
    cudaFree( dev_a ); cudaFree( dev_b ); cudaFree( dev_c );
    return 0; }
```

CUDA-Parallel Programming

- In this code, we allocate 3 arrays on the device with cudaMalloc().
- We clean up with cudaFree().
- We copy the input data to device and the result to the host.
- And we execute the device code in add() from the host.

CUDA-Parallel Programming

- But why we fill the input array on CPU ?
- Simply for performance on the GPU !

CUDA-Parallel Programming

- We use $add <<< N, 1 >>> (dev\_a, dev\_b, dev\_c)$;
- So N blocks with one thread !
- The GPU runs N copies of our kernel code.
- The block is used is referenced with the variable blockIdx.x

CUDA-Parallel Programming

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x; // handles the data at its
        thread id
    if (tid < N)
        c[tid] = a[tid] + b[tid];
```

CUDA-Parallel Programming

- When we launched the kernel, we specified N as the number of parallel blocks.
- We call the collection of parallel blocks a grid.
- This specifies to the runtime system that we want a one-dimensional grid of N blocks.
- These threads will have varying values for blockIdx.x, the first taking value 0 and the last taking value N-1.

CUDA-Parallel Programming

For example, on four blocks we obtain :

BLOCK 1

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 0;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 2

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 1;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 3

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 2;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

BLOCK 4

```
__global__ void
add( int *a, int *b, int *c ) {
    int tid = 3;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

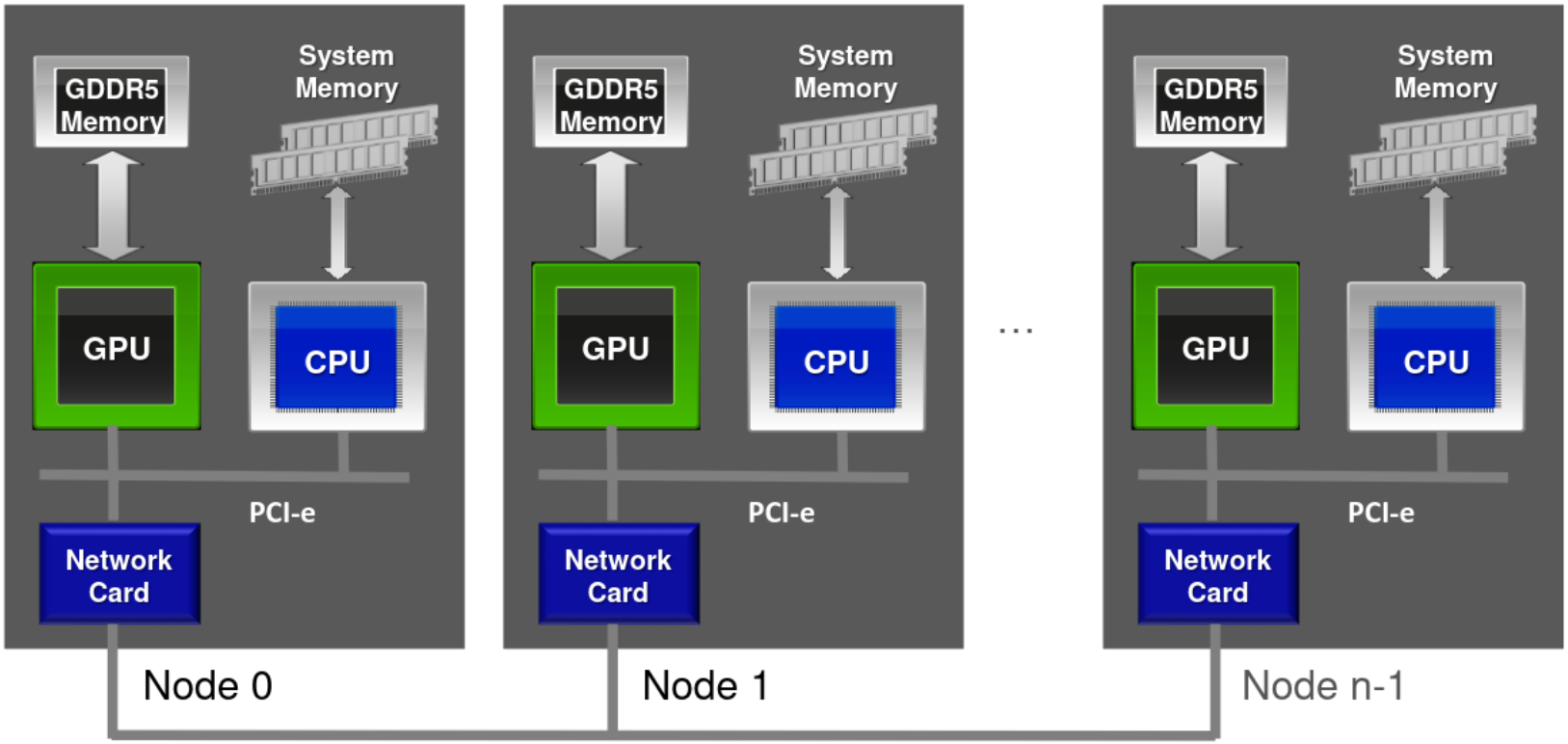
CUDA and MPI-Parallel Programming

MPI and CUDA

https://on-demand.gputechconf.com/gtc/2014/presentations/S4236-multi-gpu-programming-mpi.pdf

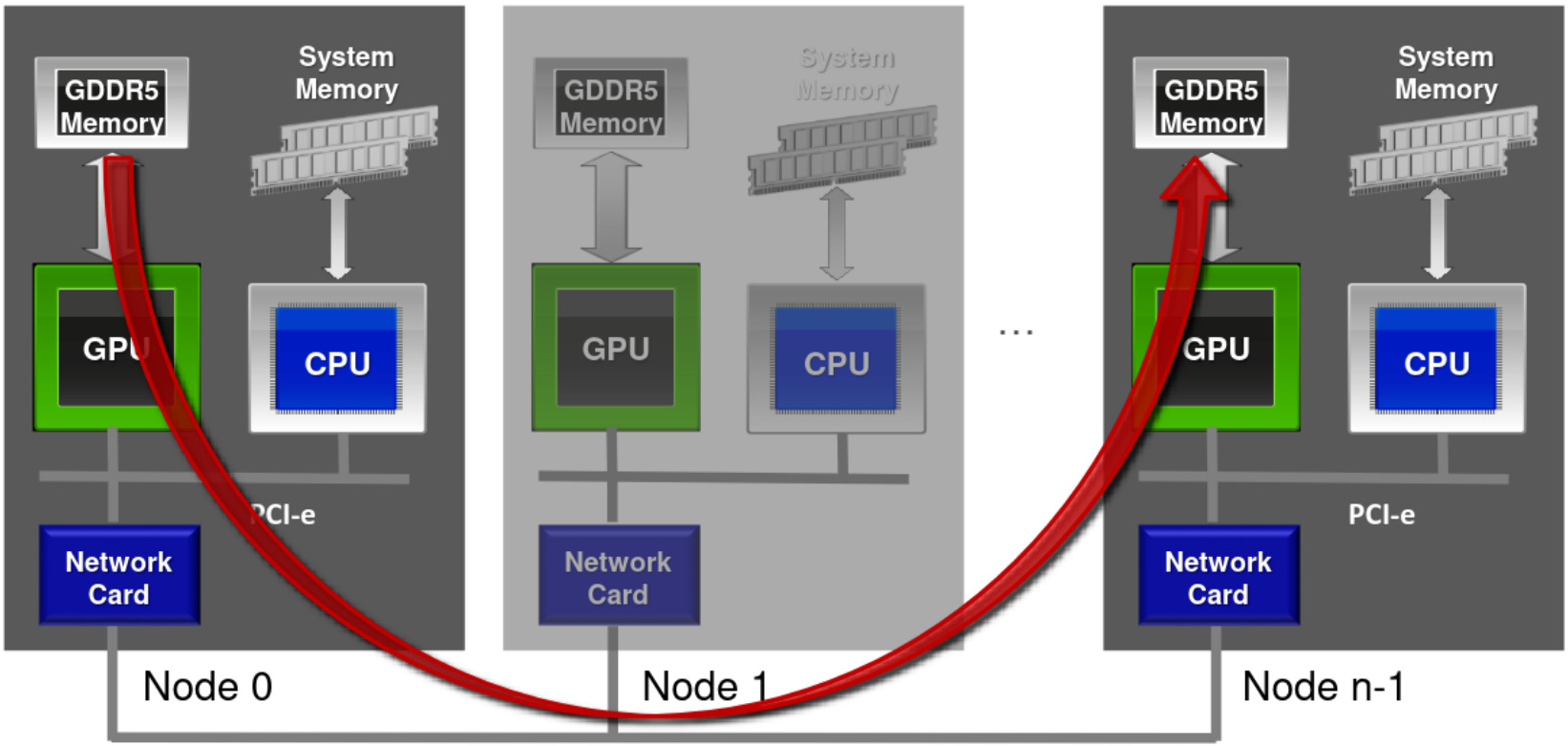
CUDA and MPI-Parallel Programming

MPI+CUDA



CUDA and MPI-Parallel Programming

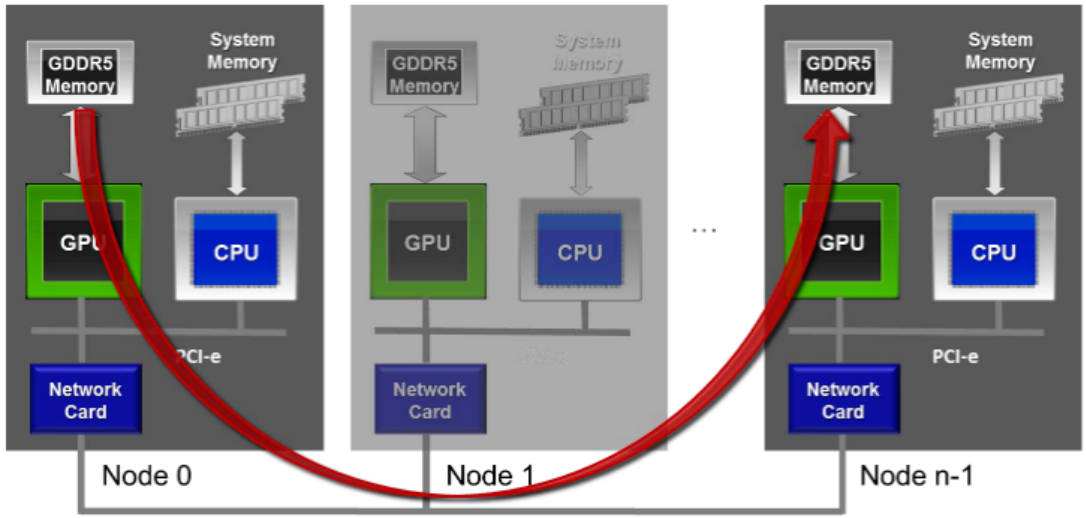
MPI+CUDA



WHAT YOU WILL LEARN

- What MPI is
- How to use MPI for inter GPU communication with CUDA an OpenACC
- What CUDA-aware MPI is
- What Multi Process Service is and how to use it
- How to use NVIDIA Tools in an MPI environment
- How to hide MPI communication times

MPI+CUDA

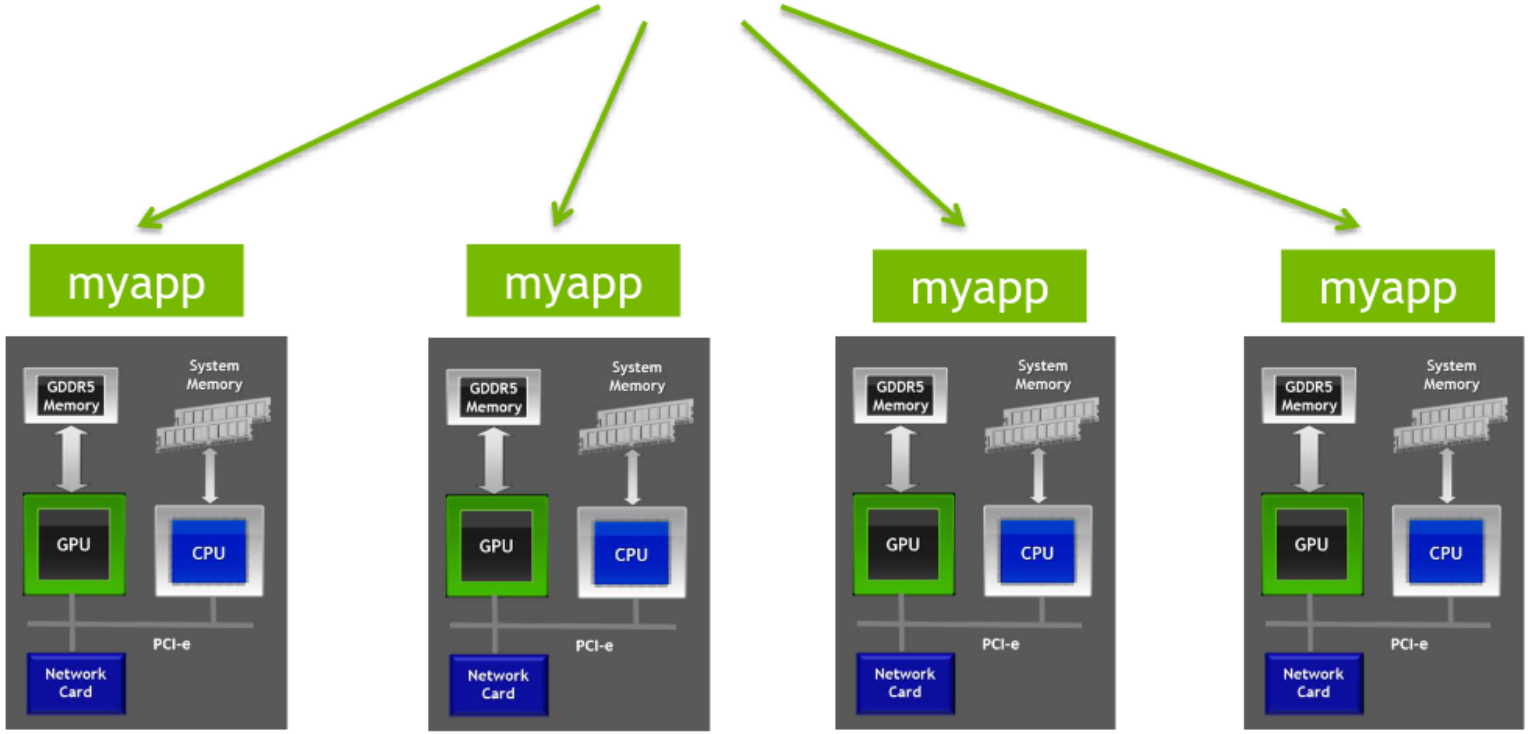



```
//MPI rank 0
MPI_Send(s_buf_d,size,MPI_CHAR,n-1,tag,MPI_COMM_WORLD);

//MPI rank n-1
MPI_Recv(r_buf_d,size,MPI_CHAR,0,tag,MPI_COMM_WORLD,&stat);
```

MPI - COMPILING AND LAUNCHING

```
$ mpicc -o myapp myapp.c
$ mpirun -np 4 ./myapp <args>
```

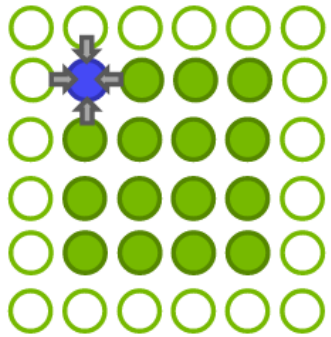


EXAMPLE: JACOBI SOLVER - SINGLE GPU

While not converged

- Do Jacobi step:

```
for (int i=1; i < n-1; i++)
  for (int j=1; j < m-1; j++)
    u_new[i][j] = 0.0f - 0.25f*(u[i-1][j] + u[i+1][j]
                              + u[i][j-1] + u[i][j+1])
```



- Swap `u_new` and `u`
- Next iteration

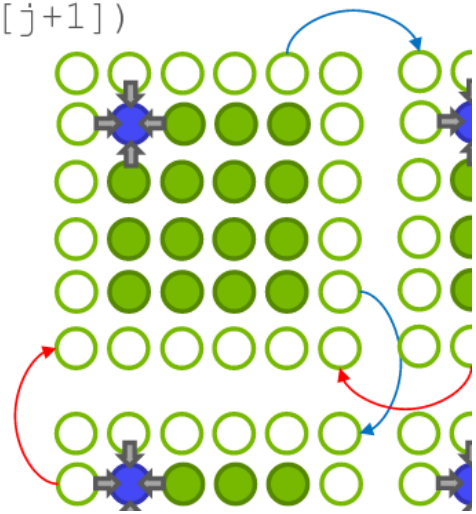
EXAMPLE: JACOBI SOLVER - MULTI GPU

While not converged

- Do Jacobi step:

```
for (int i=1; i < n-1; i++)
  for (int j=1; j < m-1; j++)
    u_new[i][j] = 0.0f - 0.25f*(u[i-1][j] + u[i+1][j]
                               + u[i][j-1] + u[i][j+1])
```

- Exchange halo with 2 4 neighbor
- Swap `u_new` and `u`
- Next iteration



EXAMPLE: JACOBI SOLVER

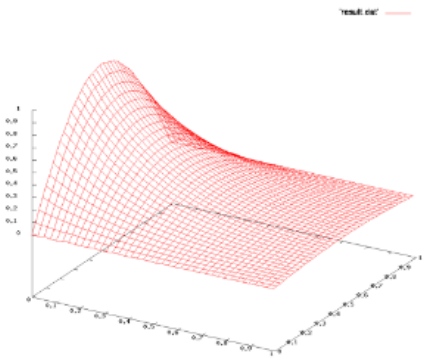
- Solves the 2D-Laplace equation on a rectangle

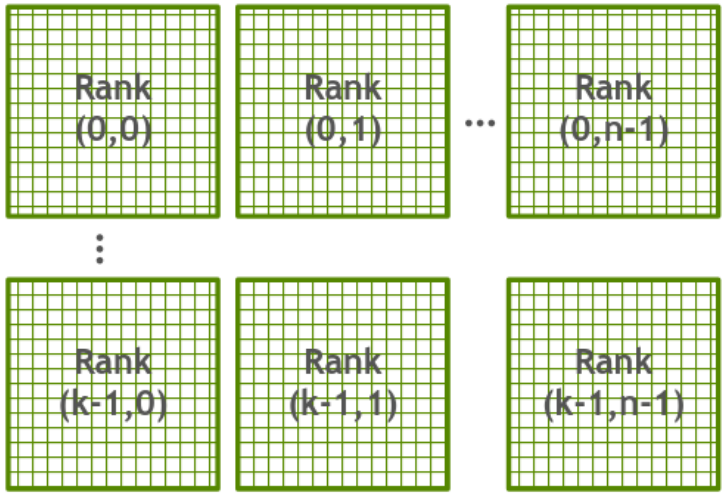
$$\Delta u(x,y) = 0 \; \forall \; (x,y) \in \Omega \backslash \delta\Omega$$

  - Dirichlet boundary conditions (constant values on boundaries)

$$u(x,y) = f(x,y) \in \delta\Omega$$

- 2D domain decomposition with n x k domains

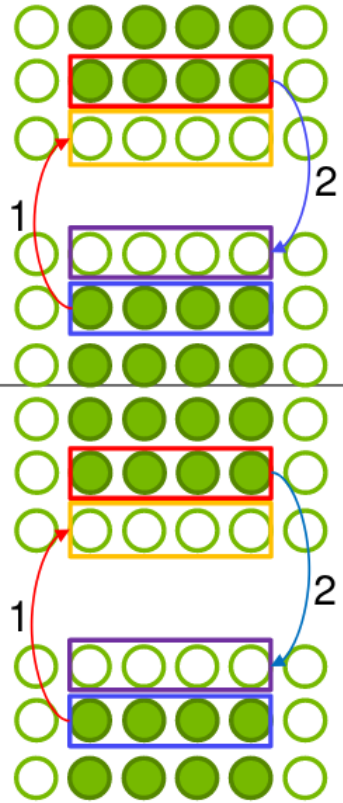




EXAMPLE: JACOBI - TOP/BOTTOM HALO UPDATE

```
MPI_Sendrecv(u_new+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
MPI_Sendrecv(u_new+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```



CUDA and MPI-Parallel Programming

CUDA

```
MPI_Sendrecv(u_new_d+offset_first_row, m-2, MPI_DOUBLE, t_nb, 0,
             u_new_d+offset_bottom_bondary, m-2, MPI_DOUBLE, b_nb, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Sendrecv(u_new_d+offset_last_row, m-2, MPI_DOUBLE, b_nb, 1,
             u_new_d+offset_top_bondary, m-2, MPI_DOUBLE, t_nb, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```