# Introduction to Parallel Programming using GPUs

Let us take hypothetical application

100 seconds

| | Routine A | Routine B | Others | Total time | Speedup |
|---|---|---|---|---|---|
| Serial | 75 | 20 | 5 | 100 | 1 |


| | Routine A | Routine B | Others | Total time | Speedup |
|---|---|---|---|---|---|
| Serial | 75 | 20 | 5 | 100 | 1 |
| 3 Cores | 25 | 20 | 5 | 50 | 2 |

100 seconds

| | Routine A | Routine B | Others | Total time | Speedup |
|---|---|---|---|---|---|
| Serial | 75 | 20 | 5 | 100 | 1 |
| 3 Cores | 25 | 20 | 5 | 50 | 2 |
| 5 Cores | 15 | 20 | 5 | 40 | 2.5 |

100 seconds

| | Routine A | Routine B | Others | Total time | Speedup |
|---|---|---|---|---|---|
| Serial | 75 | 20 | 5 | 100 | 1 |
| 3 Cores | 25 | 20 | 5 | 50 | 2 |
| 5 Cores | 15 | 20 | 5 | 40 | 2.5 |
| 10 Cores | 7.5 | 20 | 5 | 32.5 | 3.07 |

100 seconds

| | Routine A | Routine B | Others | Total time | Speedup |
|---|---|---|---|---|---|
| Serial | 75 | 20 | 5 | 100 | 1 |
| 3 Cores | 25 | 20 | 5 | 50 | 2 |
| 5 Cores | 15 | 20 | 5 | 40 | 2.5 |
| 10 Cores | 7.5 | 20 | 5 | 32.5 | 3.07 |
| 15 Cores | 5 | 20 | 5 | 30 | 3.33 |

100 seconds

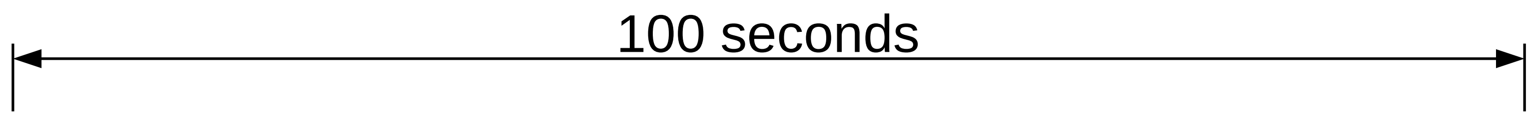
| | Routine A | Routine B | Others | Total time | Speedup |
|---|---|---|---|---|---|
| Serial | 75 | 20 | 5 | 100 | 1 |
| 3 Cores | 25 | 20 | 5 | 50 | 2 |
| 5 Cores | 15 | 20 | 5 | 40 | 2.5 |
| 10 Cores | 7.5 | 20 | 5 | 32.5 | 3.07 |
| 15 Cores | 5 | 20 | 5 | 30 | 3.33 |
| 25 Cores | 3 | 20 | 5 | 28 | 3.57 |

100 seconds

| | Routine A | Routine B | Others | Total time | Speedup |
|---|---|---|---|---|---|
| Serial | 75 | 20 | 5 | 100 | 1 |
| 3 Cores | 25 | 20 | 5 | 50 | 2 |
| 5 Cores | 15 | 20 | 5 | 40 | 2.5 |
| 10 Cores | 7.5 | 20 | 5 | 32.5 | 3.07 |
| 15 Cores | 5 | 20 | 5 | 30 | 3.33 |
| 25 Cores | 3 | 20 | 5 | 28 | 3.57 |
| 75 Cores | 1 | 20 | 5 | 26 | 3.84 |

100 seconds

| | Routine A | Routine B | Others | Total time | Speedup |
|---|---|---|---|---|---|
| Serial | 75 | 20 | 5 | 100 | 1 |
| 3 Cores | 25 | 20 | 5 | 50 | 2 |
| 5 Cores | 15 | 20 | 5 | 40 | 2.5 |
| 10 Cores | 7.5 | 20 | 5 | 32.5 | 3.07 |
| 15 Cores | 5 | 20 | 5 | 30 | 3.33 |
| 25 Cores | 3 | 20 | 5 | 28 | 3.57 |
| 75 Cores | 1 | 20 | 5 | 26 | 3.84 |
| Infinite # cores | 0 | 20 | 5 | 25 | 4 |




| | Routine A | Routine B | Others | Total time | Speedup |
|---|---|---|---|---|---|
| Serial | 75 | 20 | 5 | 100 | 1 |
| 5 Cores | 15 | 4 | 5 | 24 | 4.16 |
| 10 Cores | 7.5 | 2 | 5 | 14.5 | 6.89 |
| Infinite # cores | 0 | 0 | 5 | 5 | 20 |

| | Routine A | Routine B | Others | Total time | Speedup |
|---|---|---|---|---|---|
| Serial | 75 | 24 | 1 | 100 | 1 |
| Infinite # cores | 0 | 0 | 1 | 1 | 100 |

Optional reading :

Amdahl's law
Gustafson's law

# Parallel Programming Workflow

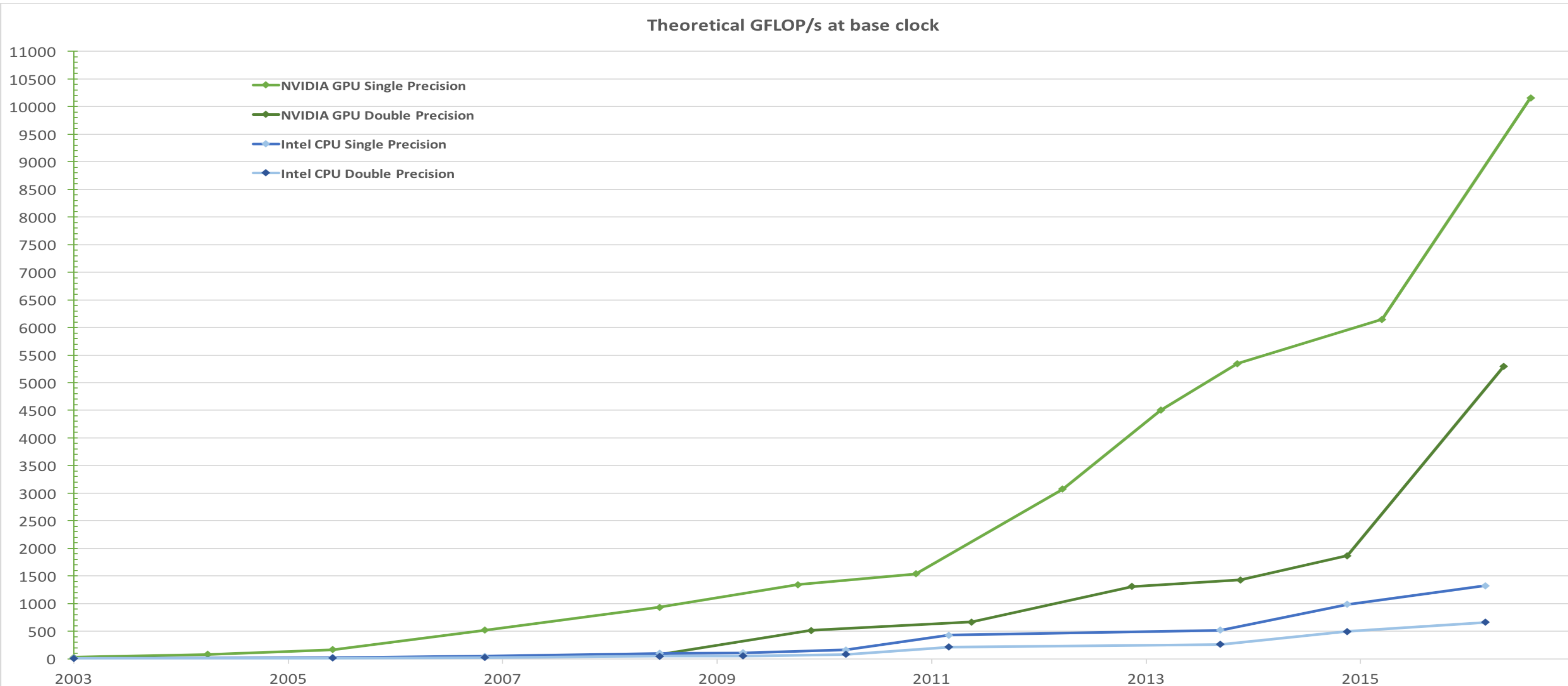
1) Take correct serial code with sample input testcases (small, medium, large) and example outputs (for verification)
2) Profile the code
3) Select the most time consuming routine / code
4) Parallelize this routine
5) Repeat steps 2,3,4 until all (most) of the routines are parallelized

# Sample input testcases and output verification

- Input Testcases
  - Small : smaller testcase for development and testing, should not take more than few minutes
  - Medium : Typical problem size you use regularly
  - Large : Problem size you really want to target once you have parallel version of the code
- Output verification
  - Floating point numbers will not match exactly. Not even serial codes compiled with different compilers on same processor or different processor architecture!
  - We should define method for error/difference calculation and acceptable error/difference tolerance.
- Number of testcases : varies; but should cover all the different parts of the codes – boundary conditions, different cell types etc

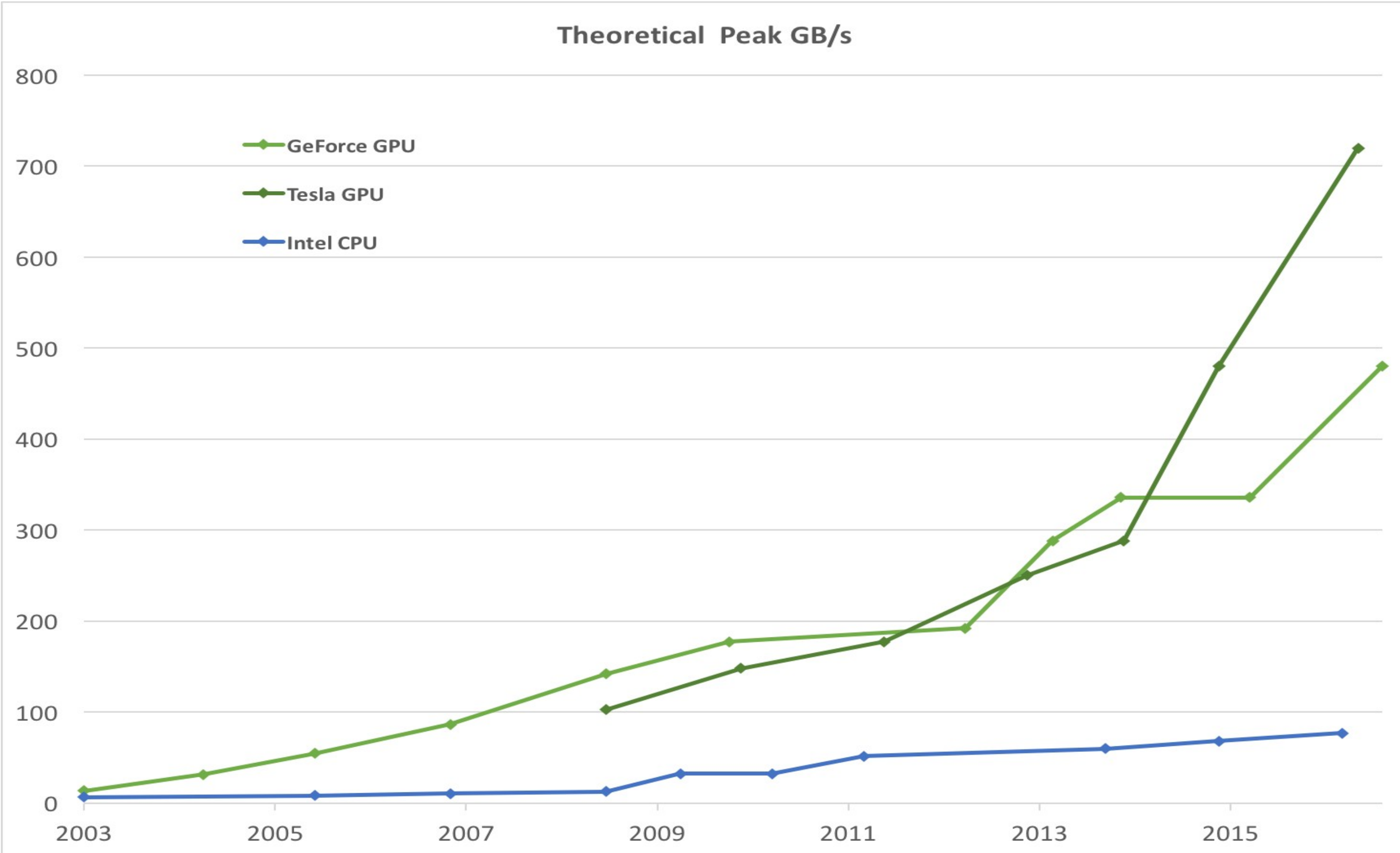
# GPGPU Programming

# Why GP-GPU?



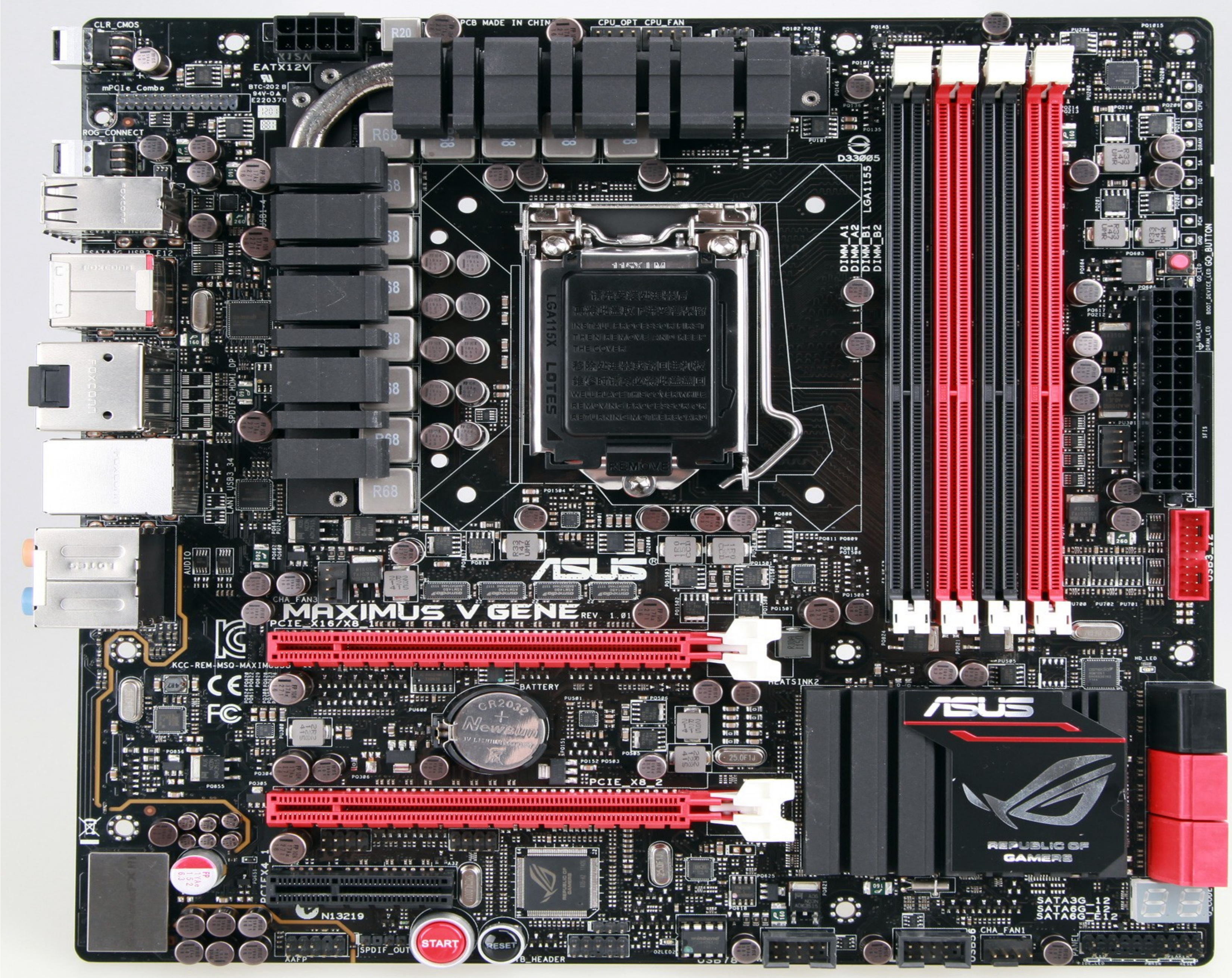

Source: CUDA C Programming Guide (NVIDIA)

# Why GP-GPU?

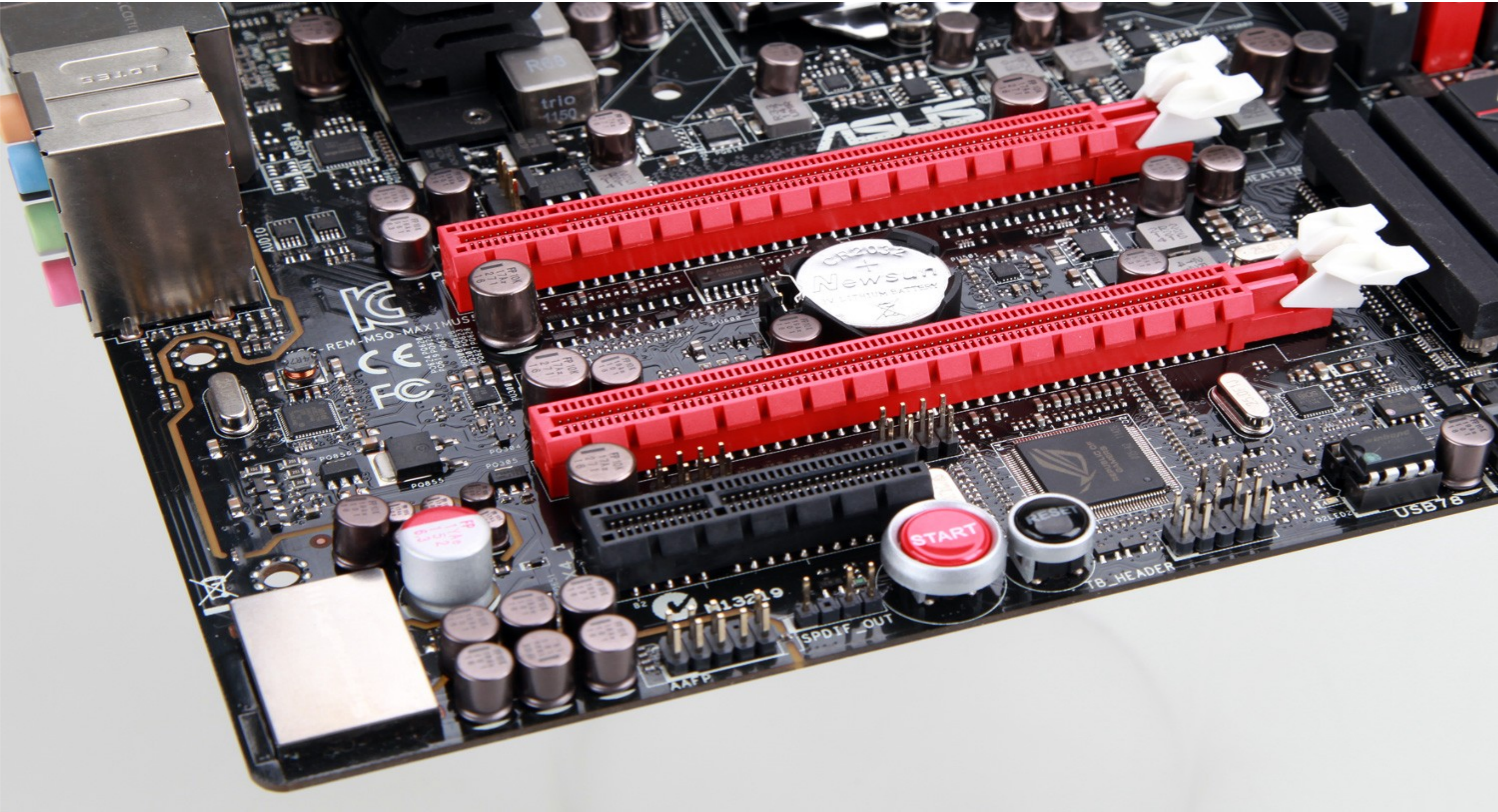



Source: CUDA C Programming Guide (NVIDIA)

Where is my GPU?


ASUS
MAXIMUS V GENE
REV. 1.01
PCIE_X16/X8_1
PCIE_X8_2
PCIEX4_1
LGA1155
DIMM_A1
DIMM_A2
DIMM_B1
DIMM_B2
EATX12V
CPU_OPT CPU_FAN
CLR_CMOS
mPCIe_Combo
ROG_CONNECT
CHA_FAN3
CHA_FAN1
BATTERY
CR2032
HEATSINK2
REPUBLIC OF GAMERS
SATA3G_12
SATA6G_12
SATA6G_E12
START
RESET
GO_BUTTON
N13219
SPDIF_OUT
AAFP
AUDIO
PCB MADE IN CHINA

www.asus.com

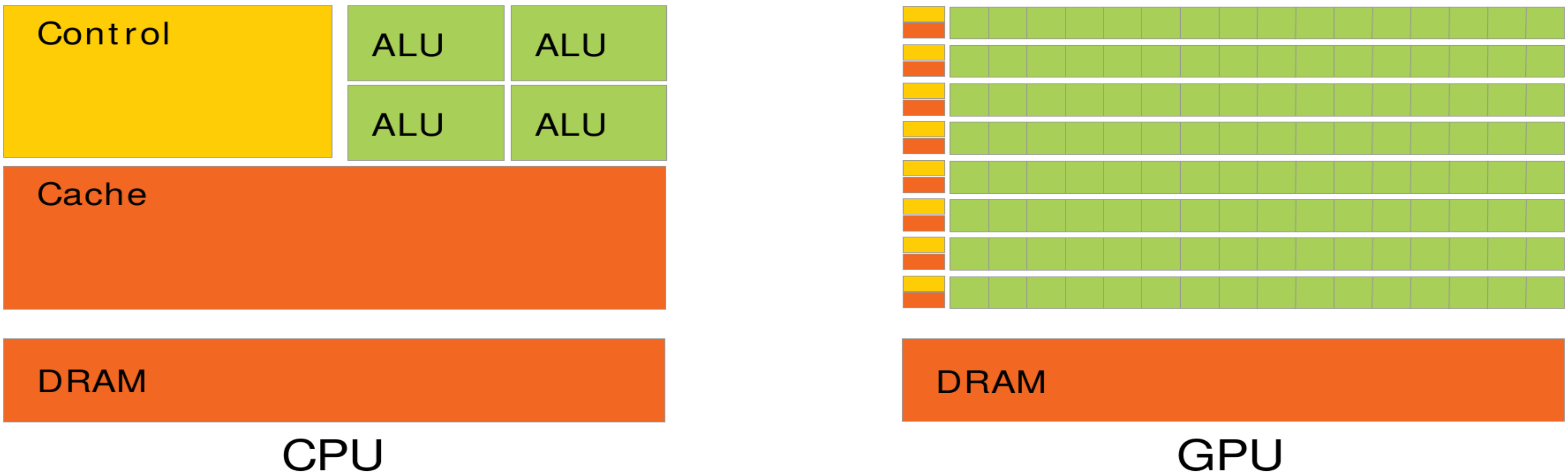

START
RESET
SPDIF_OUT
AAFP
TB_HEADER
USB78
CR2032
Newsun
AUDIO


www.asus.com

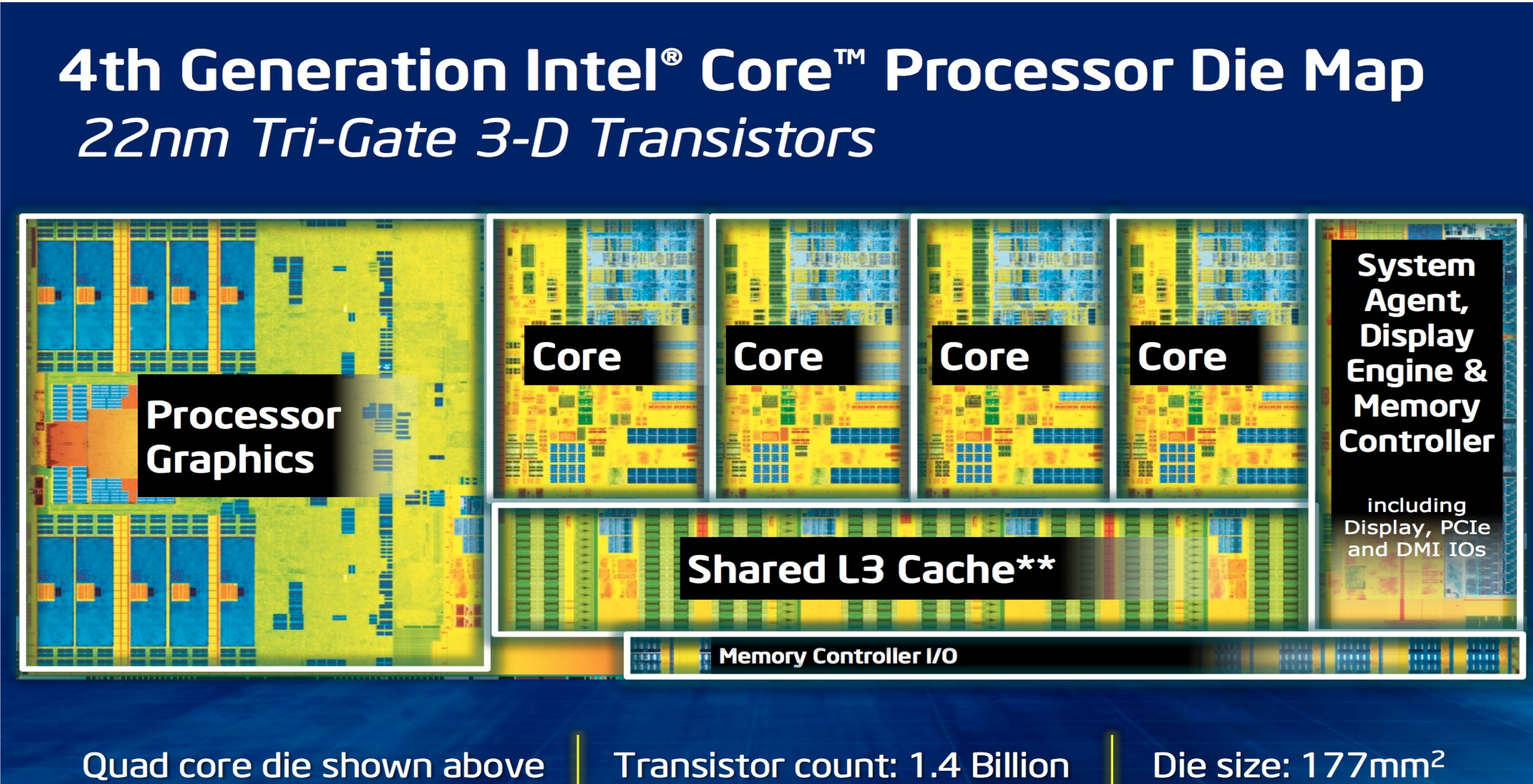

www.asus.com

# How it differs from CPU?

# CPU vs GPU




Source: CUDA C Programming Guide (NVIDIA)



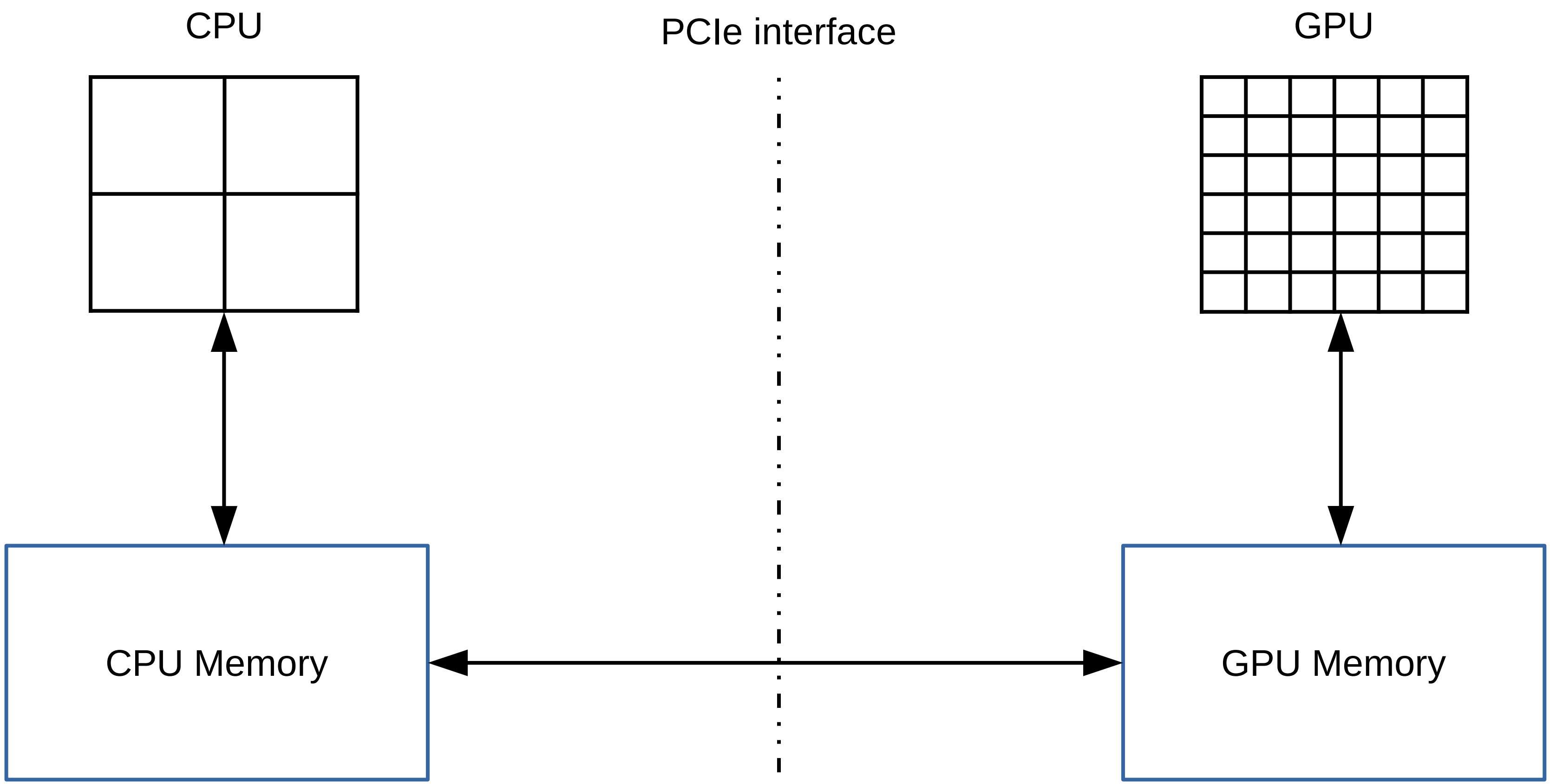

http://images.anandtech.com/doci/7003/Screen%20Shot%202013-05-31%20at%207.59.16%20PM.png

# CPU vs GPU

- CPU
  - Optimized for serial thread performance
  - Good for complex tasks
  - Few, large, complex cores
  - Large number of transistors are allocated for Caches, Instruction Level Parallelism
- GPU
  - Optimized for data parallel, throughput computations
  - Large number of very small, simple cores
  - More number of transistors are allocated for computations

How to program these GPUs for general purpose computing?

# Programmer’s View



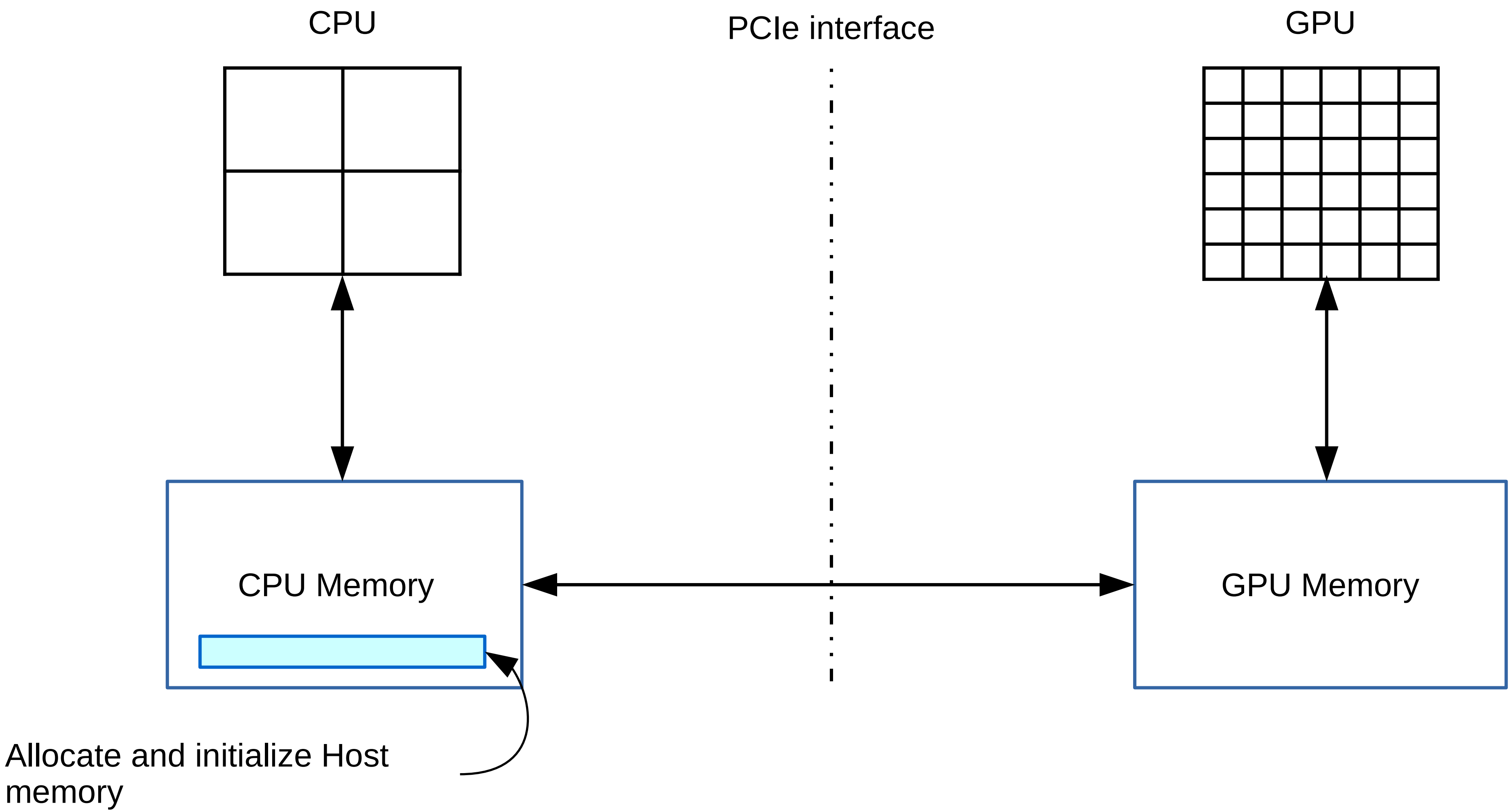
# Programming GPUs

- CUDA
- OpenCL
- OpenMP 4.0+
- OpenACC
- . . .

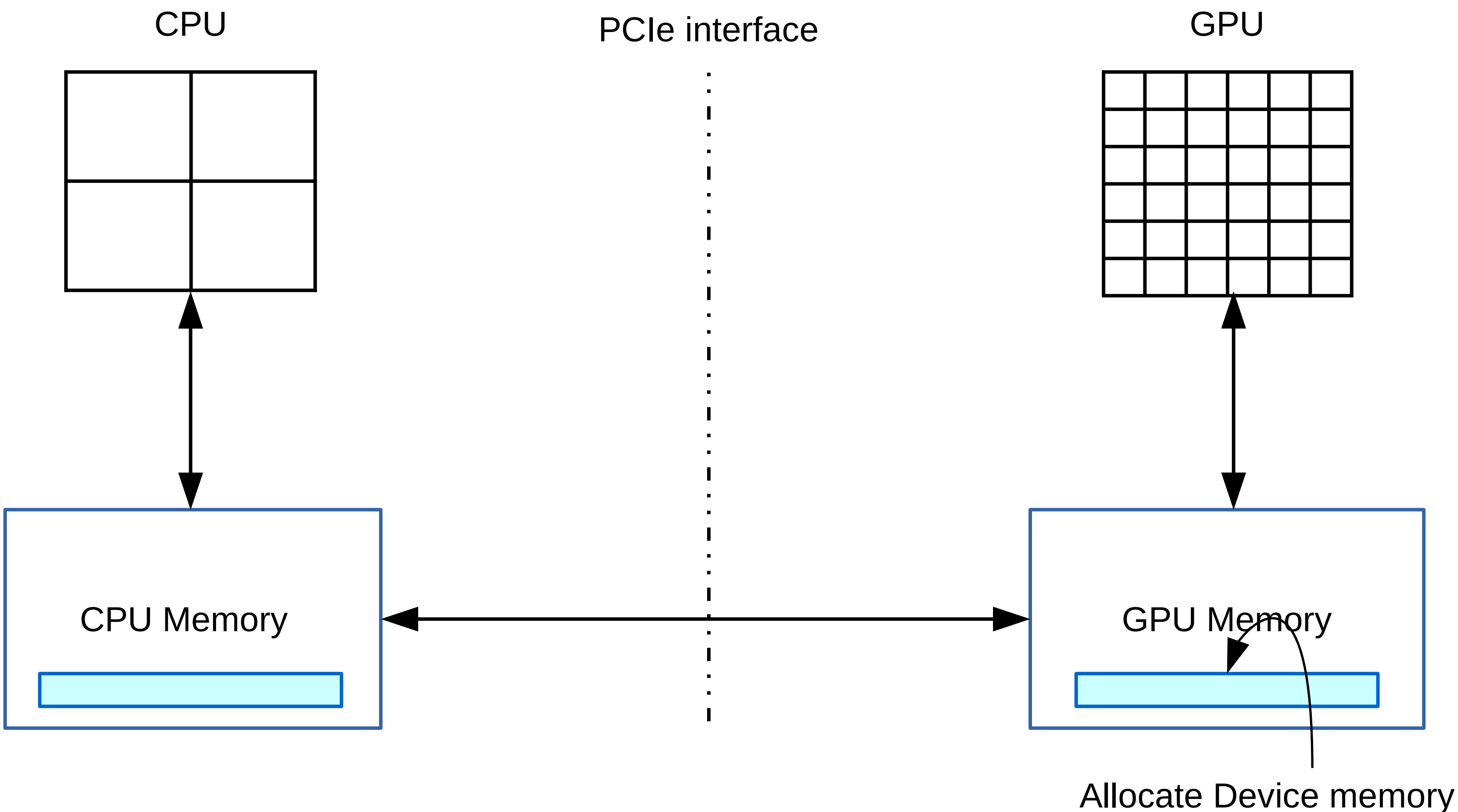
# Pseudo Code (old method)

1. Allocate and initialize memory on Host (CPU)
2. Allocate memory on Device (GPU)
3. Transfer data from Host memory to Device memory
4. Launch “kernel” on the Device – large number of threads execute in parallel on Device
5. Transfer results from Device memory to Host memory
6. De-allocate all the memory and terminate the program

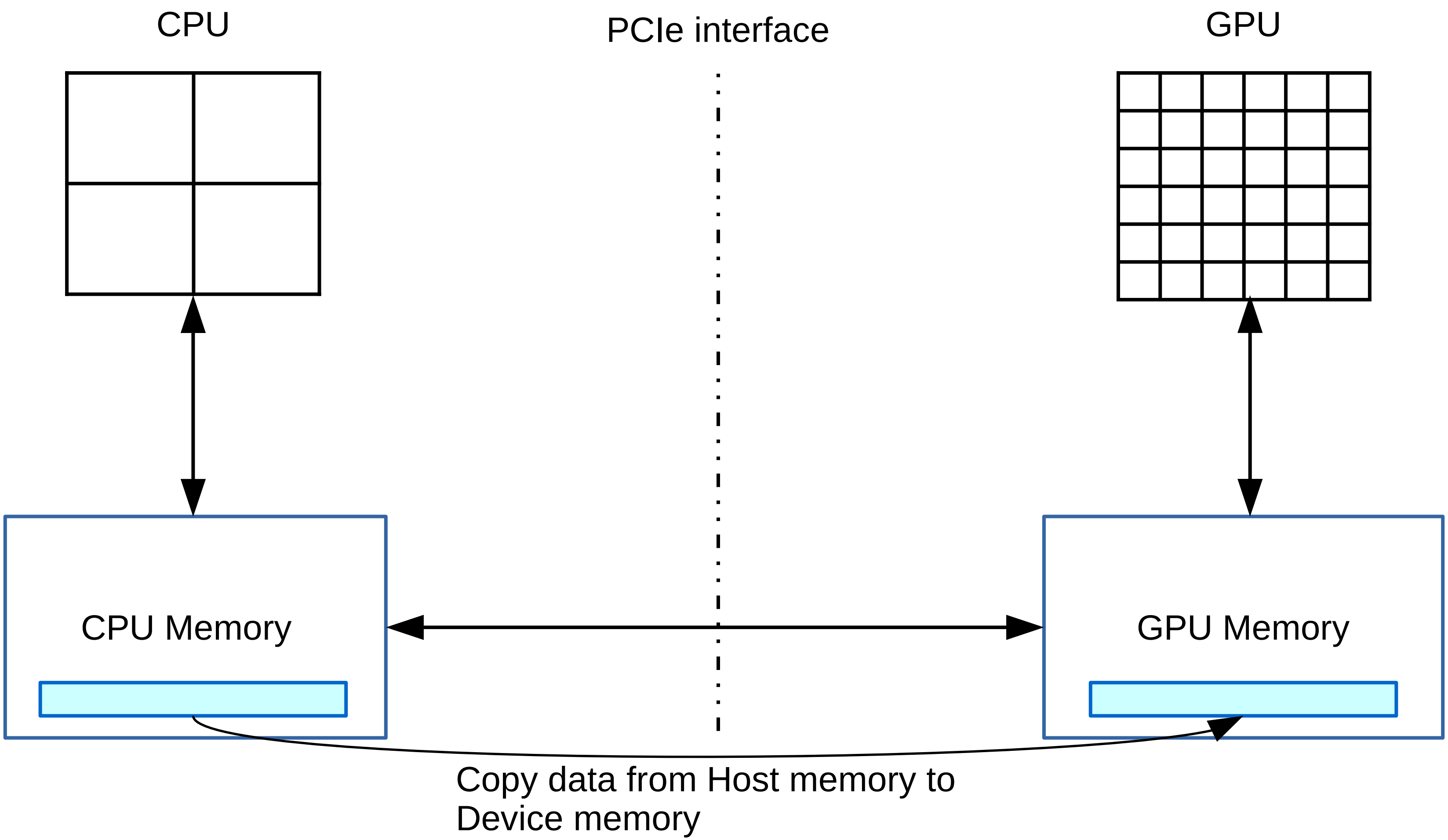
# Step 1


CPU
PCIe interface
GPU
CPU Memory
GPU Memory
Allocate and initialize Host memory

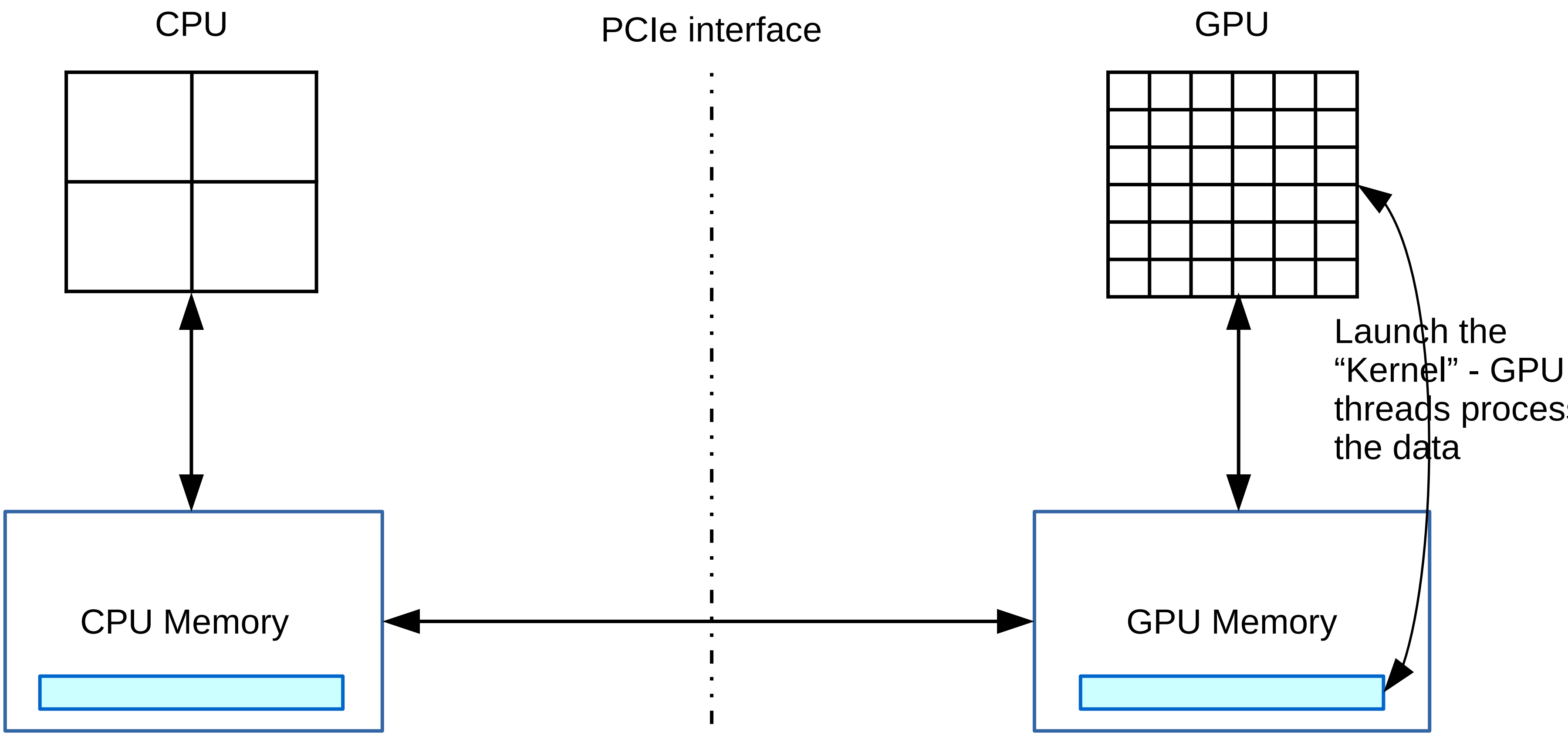
# Step 2


CPU
PCIe interface
GPU
CPU Memory
GPU Memory
Allocate Device memory

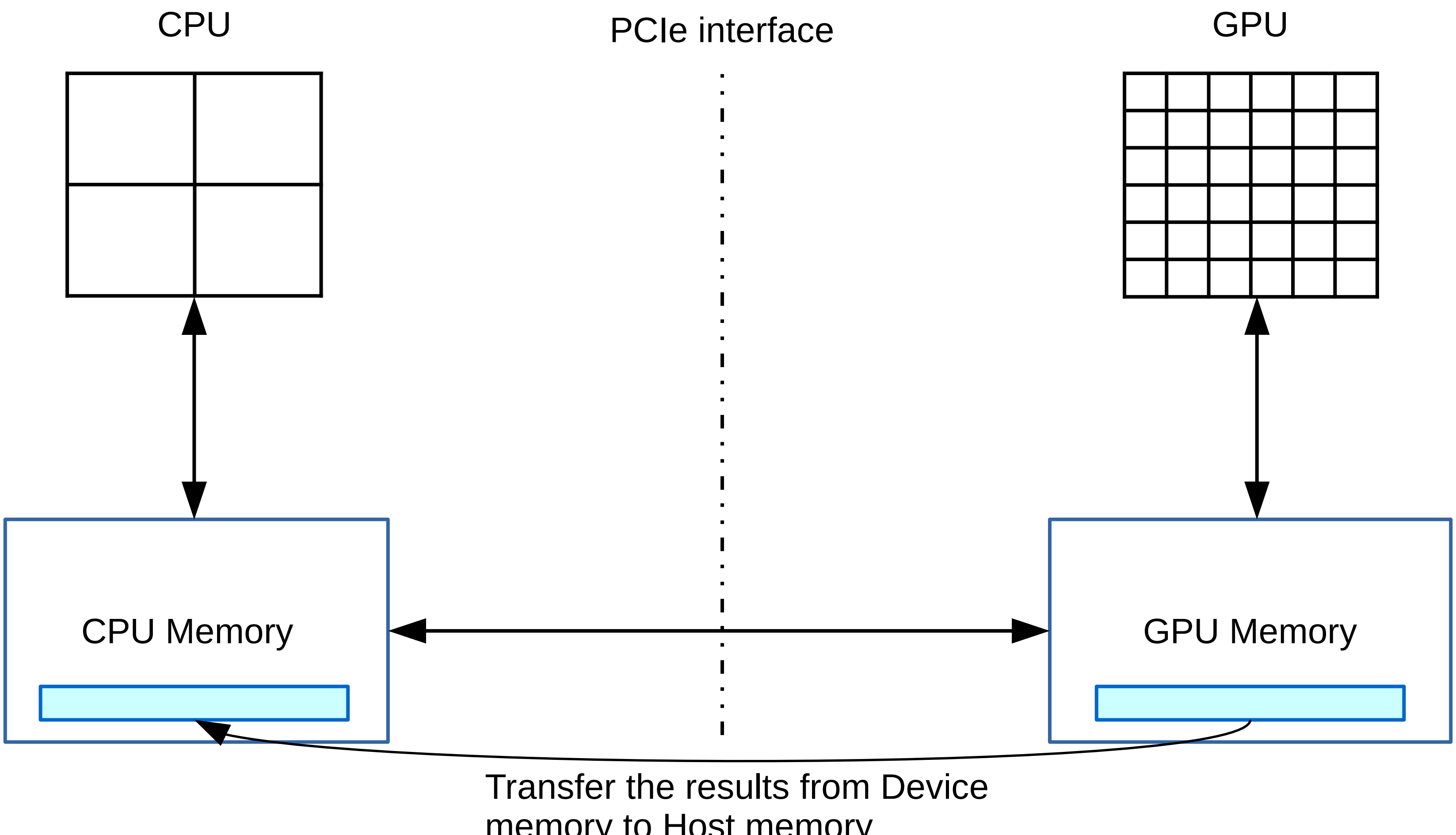
# Step 3


CPU
PCIe interface
GPU
CPU Memory
GPU Memory
Copy data from Host memory to
Device memory

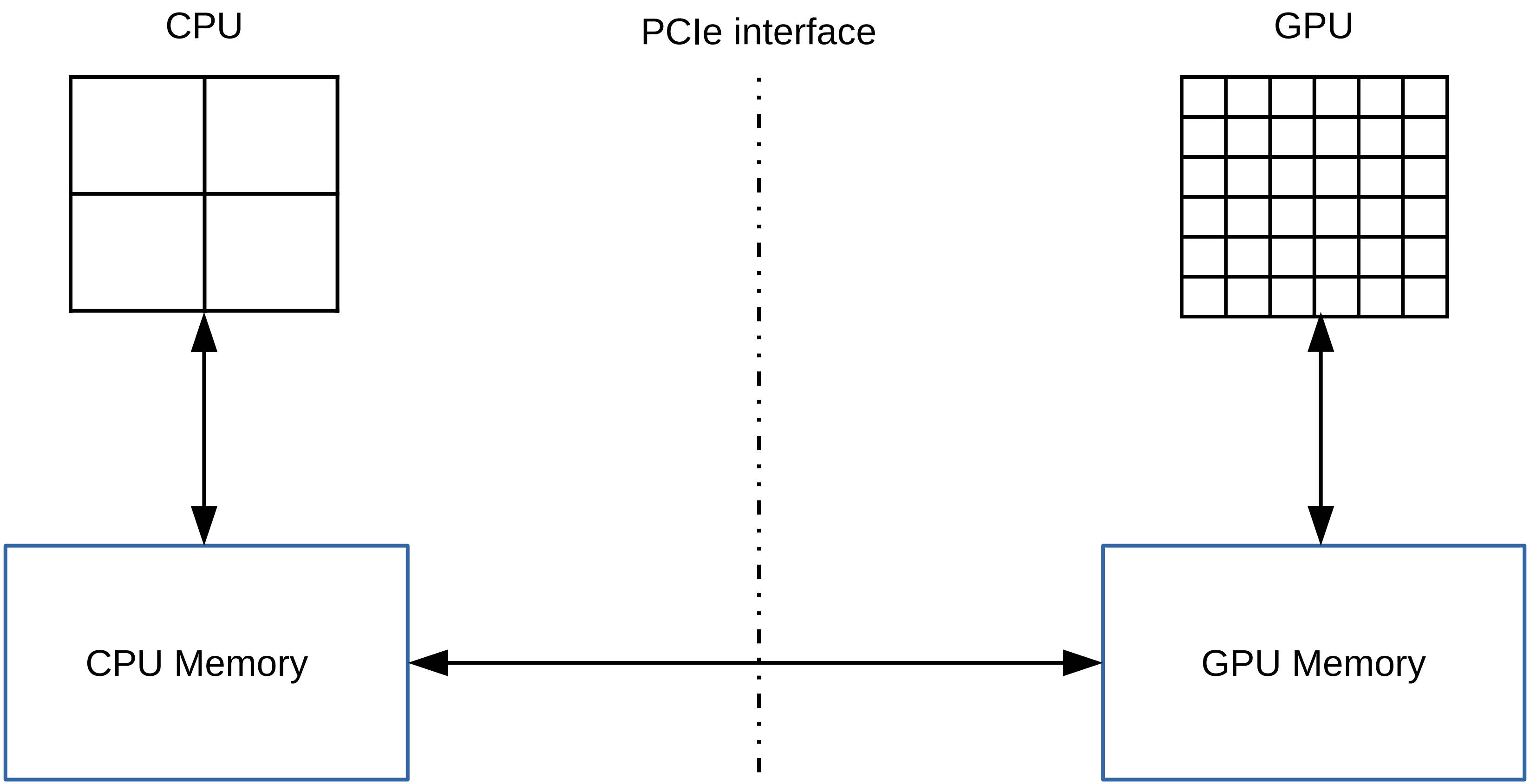
# Step 4


CPU
PCIe interface
GPU
Launch the
“Kernel” - GPU
threads process
the data
CPU Memory
GPU Memory

# Step 5


CPU
PCIe interface
GPU
CPU Memory
GPU Memory
Transfer the results from Device
memory to Host memory

# Step 6

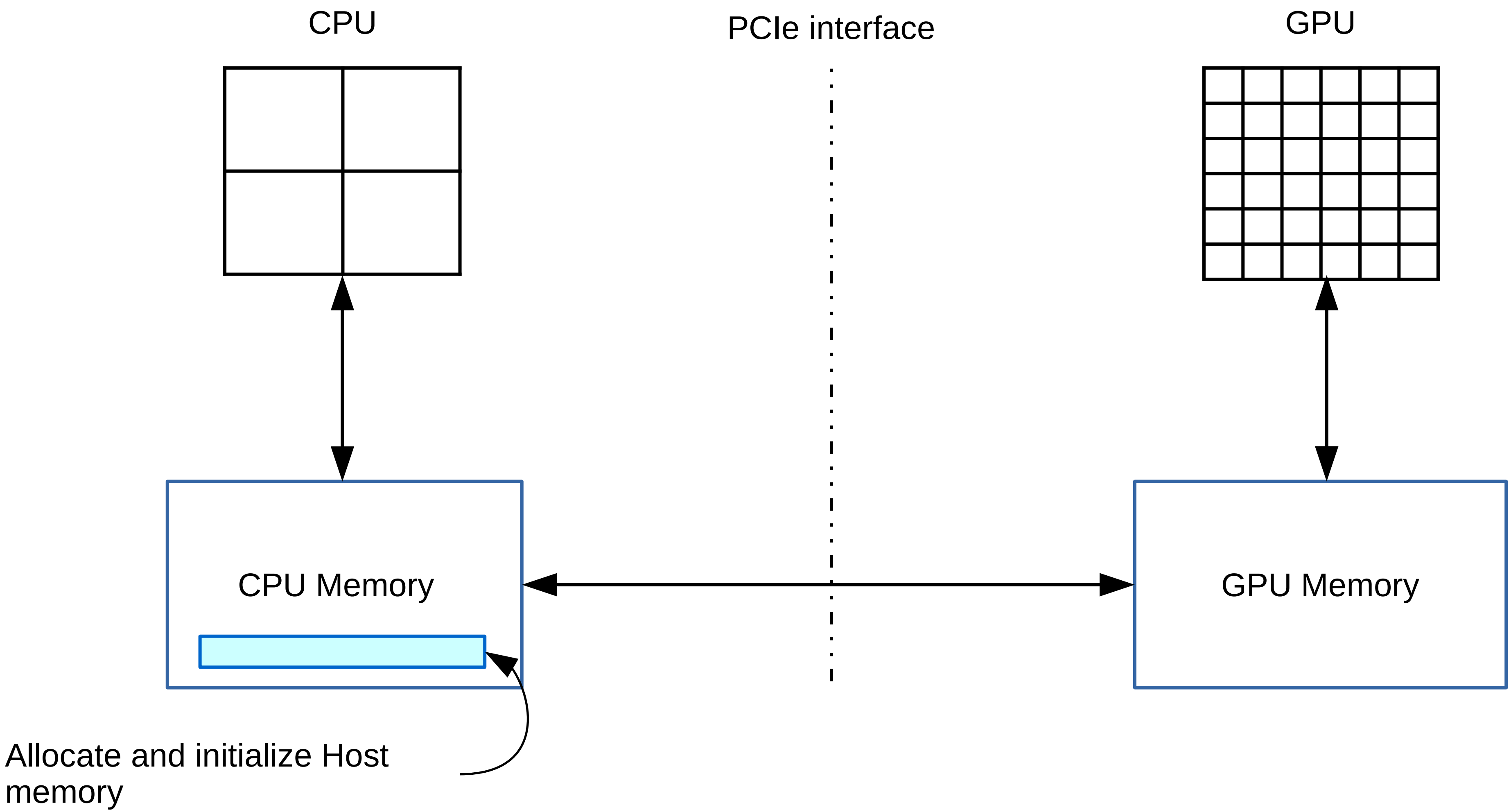



De-allocate the memories and
terminate the program

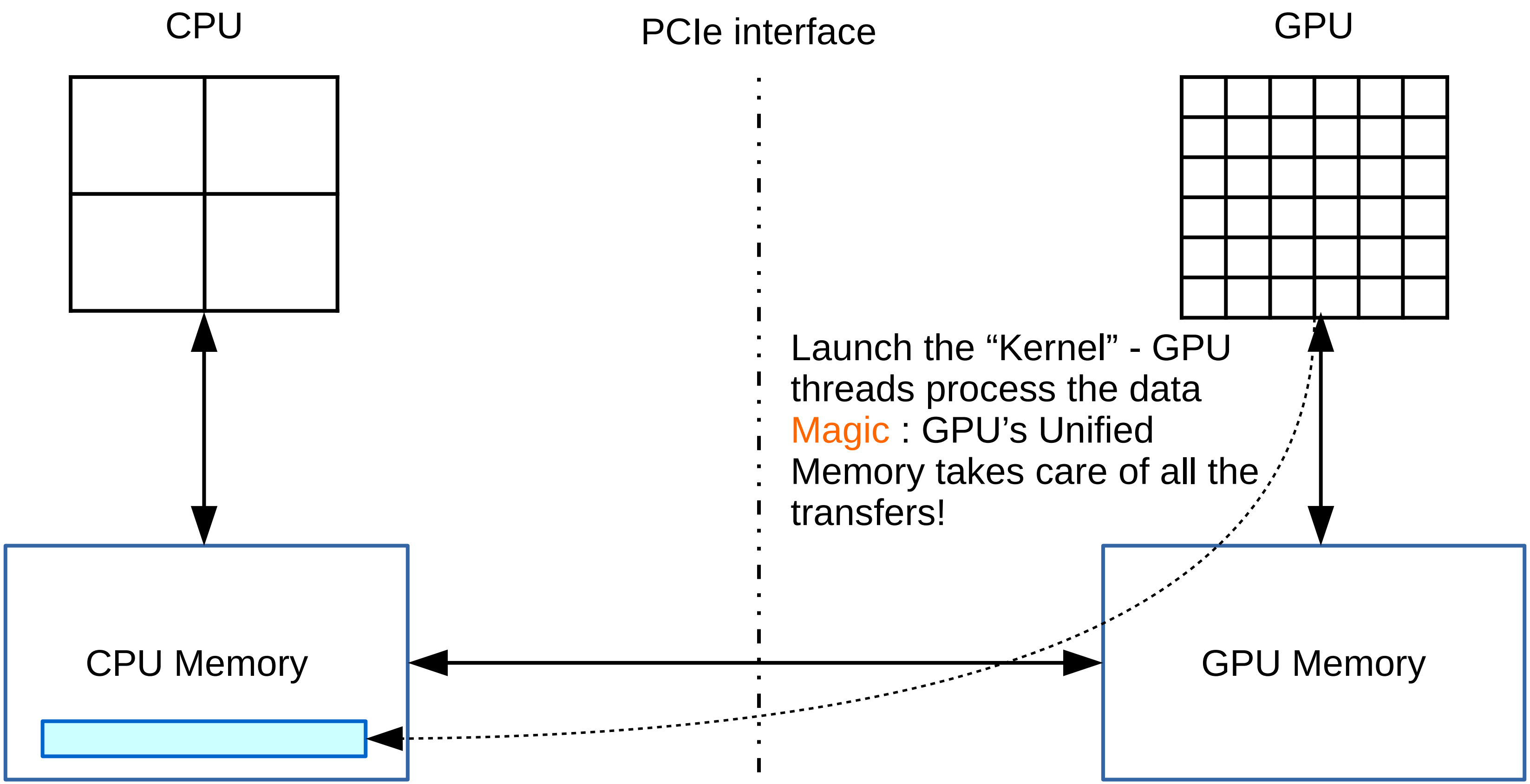
# Pseudo Code (new method)

1. Allocate and initialize Host Memory
2. Launch “kernel” on the Device – large number of threads execute in parallel on Device (Magic Step!)
3. De-allocate all the memory and terminate the program

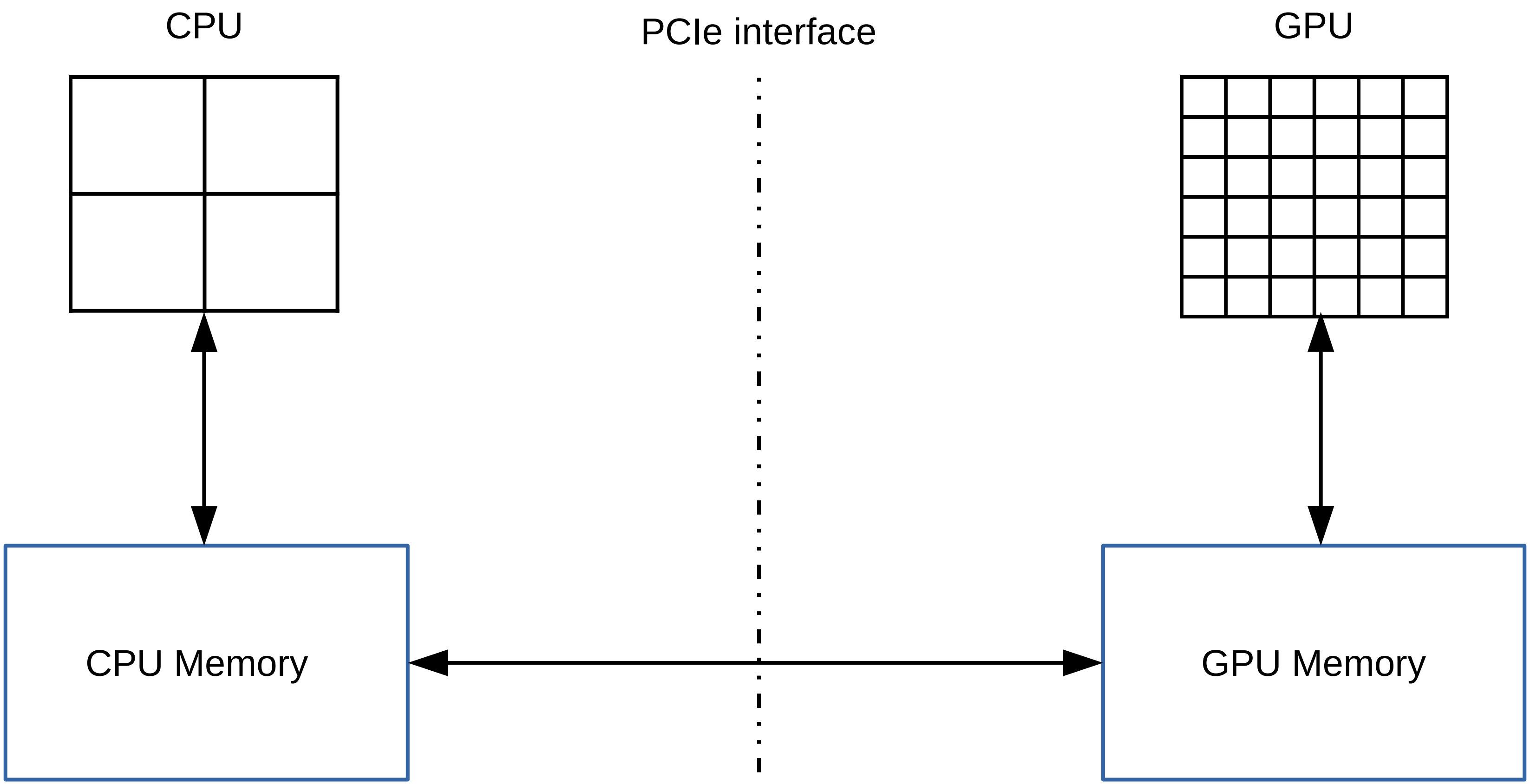
# Step 1


CPU
PCIe interface
GPU
CPU Memory
GPU Memory
Allocate and initialize Host memory

# Step 2 (Magic Step!)



# Step 3



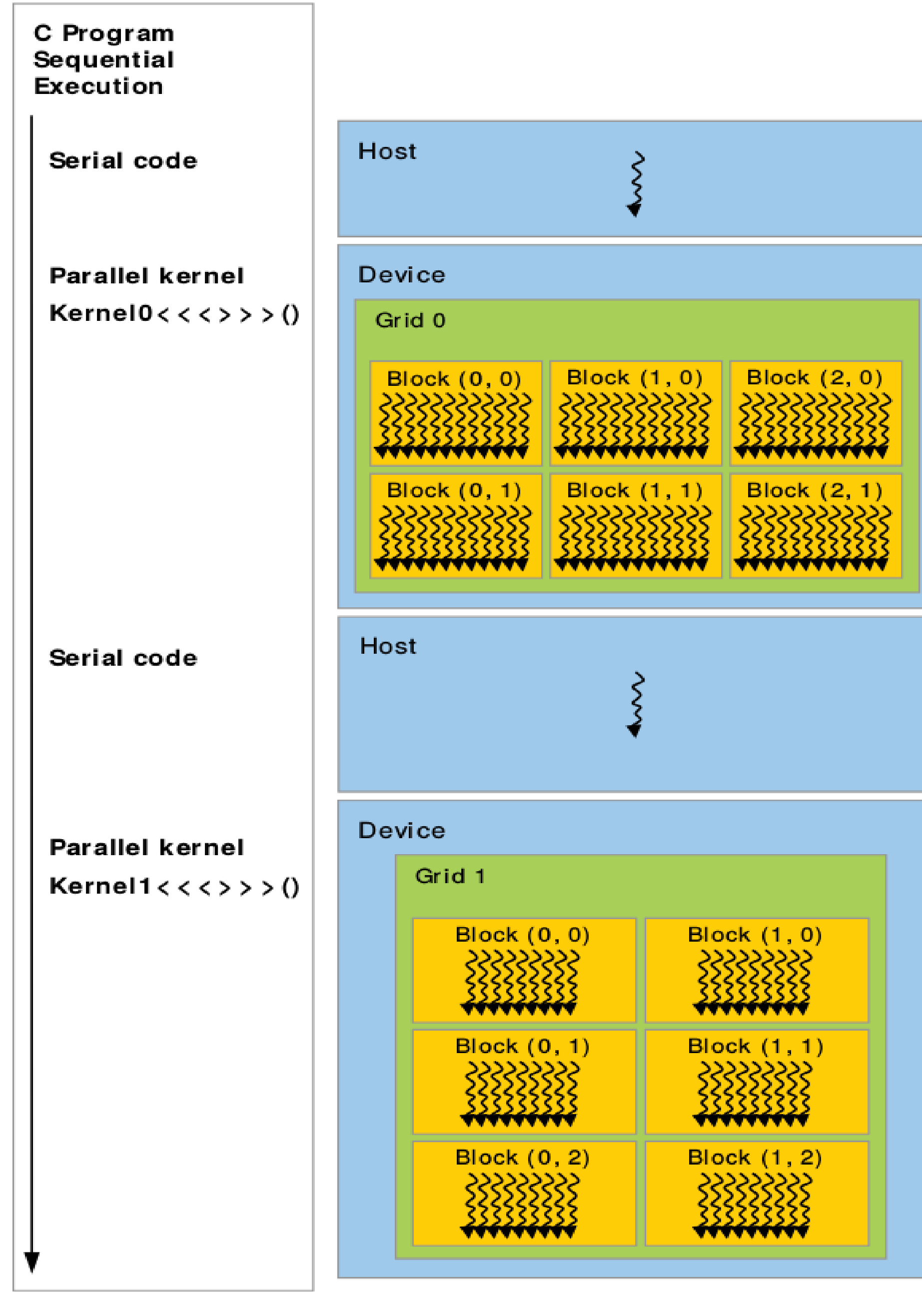

De-allocate the memory and
terminate the program

# Introduction to CUDA

# CUDA

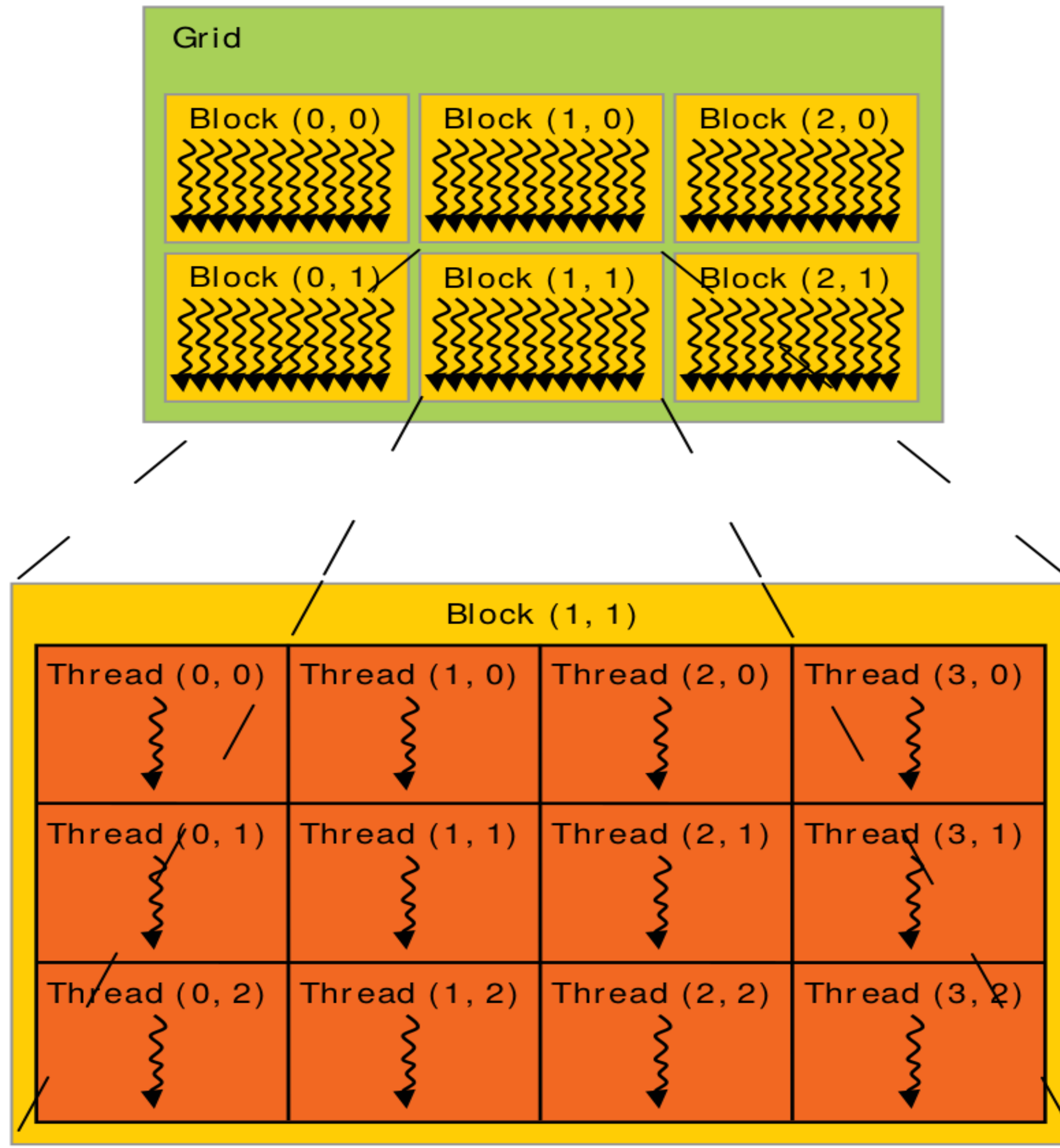
- Parallel computing platform and application programming interface (API)
- Nvidia Corp.-  General Purpose Graphics Processing Unit (GPGPU)
- Supports most of the today's Nvidia's gaming cards.
- Platforms
  - Geforce : Desktop
  - Quadro : Workstation
  - Tesla : Datacenter
  - Tegra, Jetson, Drive : Embeded

# Structure of a CUDA Code




Source: CUDA C Programming Guide (NVIDIA)

# Threads, Blocks and Grid(s)




Source: CUDA C Programming Guide (NVIDIA)

# Vector addition (old method)

```
int m[200], n[200], p[200],*md, *nd,*pd;
int size = 200 * sizeof(int);
// Initialize array m and array n
...
cudaMalloc(&md, size);
cudaMemcpy(md, m, size, cudaMemcpyHostToDevice);
cudaMalloc(&nd, size);
cudaMemcpy(nd, n, size, cudaMemcpyHostToDevice);
cudaMalloc(&pd, size);
arradd<<< 1,200 >>>(md,nd,pd);
cudaMemcpy(p, pd, size, cudaMemcpyDeviceToHost);
cudaFree(md);
cudaFree(nd);
cudaFree(pd);
```

# “Kernel” function

```
__global__ void arradd(int* md, int* nd, int* pd)
{
int myid = threadIdx.x;

pd[myid] = md[myid] + nd[myid];
}
```

# Questions?

Thank you.