

Introduction to CUDA Programming

Sudsanguan Ngamsuriyaroj
Ekasit Kijsipongse
Ittipon Rassameeroj

Semester 1/2022

Topics

- Parallel computer architecture (revisited)
- GPU hardware and architecture
- CPU vs. GPU
- GPU Performance
- CUDA programming – basic concepts
- Simple examples to illustrate basic concepts

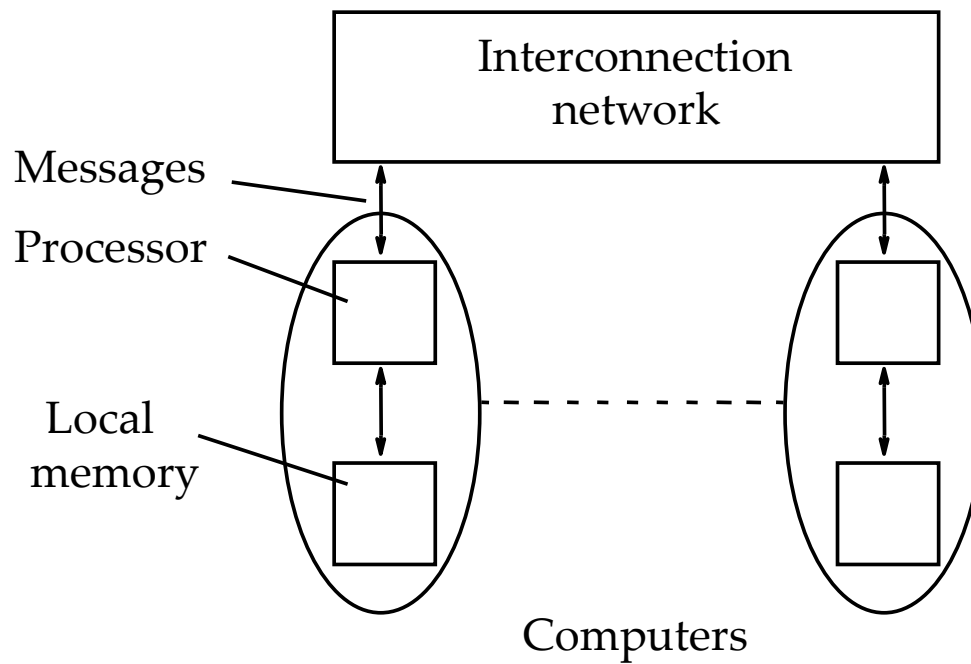
Parallel Computer Architecture

- Distributed Memory (Multicomputer)
 - Clusters Computer
 - NOW (Network of Workstations)
- Shared Memory (Multicore/Multiprocessor)
 - Symmetric Multiprocessors (SMP)
 - NUMA (Non-Uniform Memory Access)
- Processor Array
 - Graphic Processing Unit (GPU)



Distributed Memory (Multicomputer)

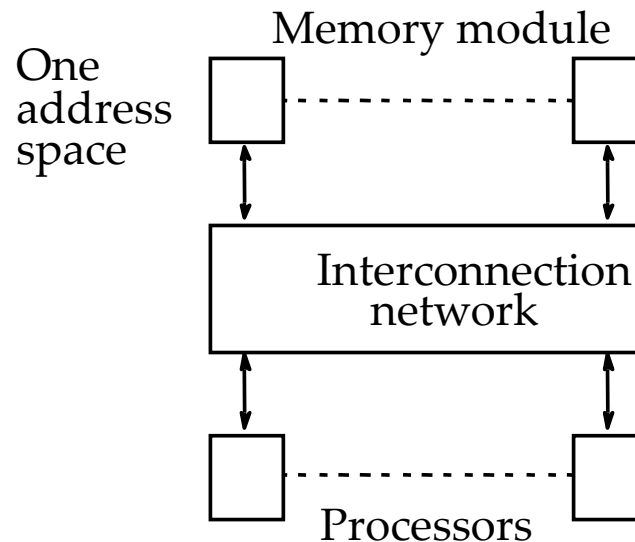
Complete computers connected through an interconnection network:



Shared Memory Multiprocessor

Multiple processors connected to multiple memory modules, such that each processor can access any memory module.

- Symmetric Multiprocessors (SMP)
- NUMA (Non-Uniform Memory Access)



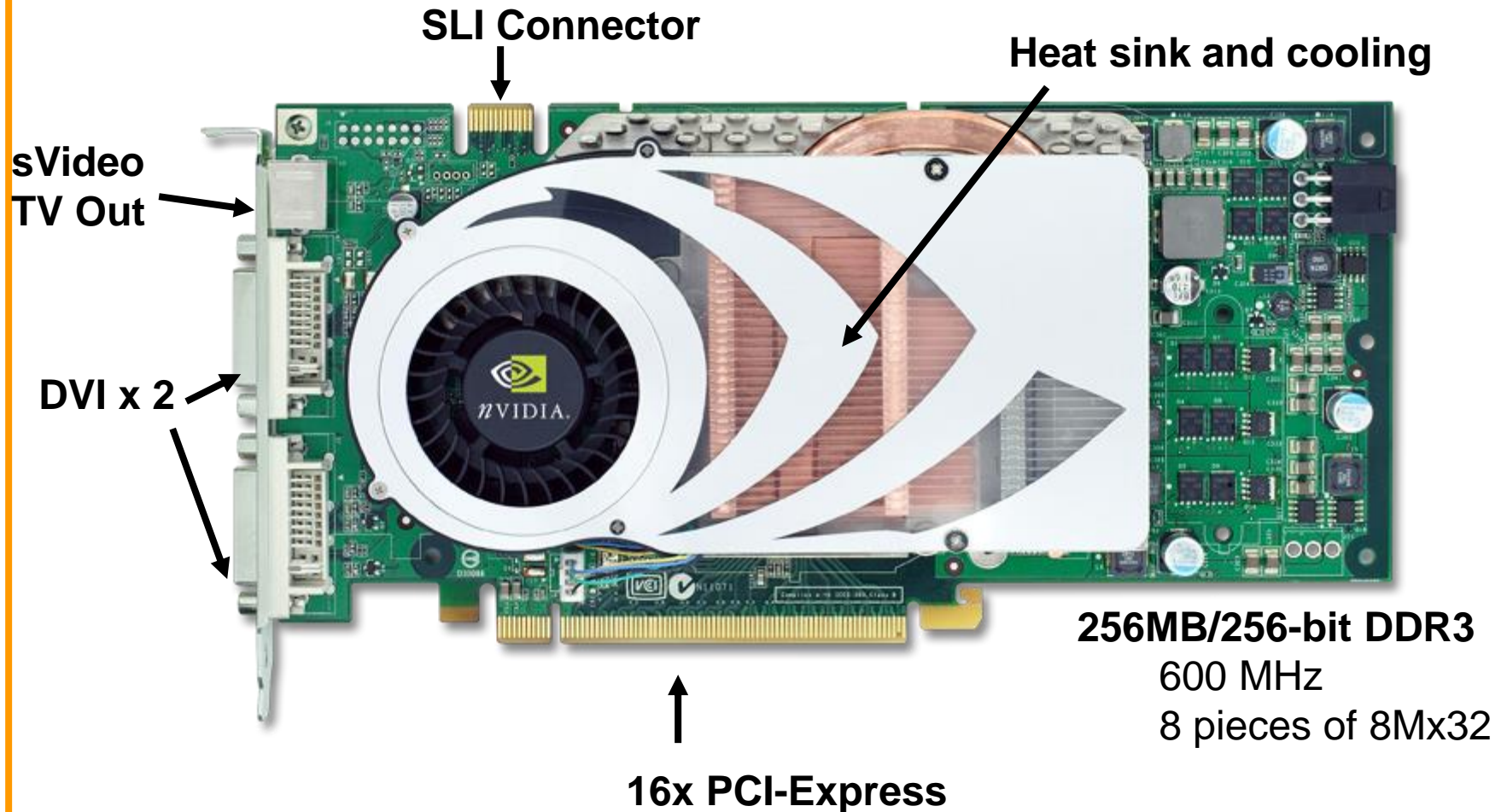
Graphic Processing Units

- Processor array for graphic rendering
- GPU video card
 - Nvidia
 - AMD (ATI)
- Available on desktop and notebook (Gaming PC)



GeForce 7800 GTX (2005)

Board Details



Nvidia GeForce 210 (2009)



GPU Engine Specs:

CUDA Cores	16
Graphics Clock (MHz)	589 MHz
Processor Clock (MHz)	1402 MHz

Memory Specs:

Memory Clock (MHz)	500
Standard Memory Config	512MB or 1 GB DDR2
Memory Interface Width	64-bit
Memory Bandwidth (GB/sec)	8.0

Nvidia Tesla C1060 (2008)



Form Factor	10.5" x 4.376", Dual Slot
# of Streaming Processor Cores	240
Frequency of processor cores	1.3GHz
Single Precision floating point performance (peak)	933 GFLOP/s
Double Precision floating point performance (peak)	78 GFLOP/s
Floating Point Precision	IEEE 754 single & double
Total Dedicated Memory	4GB GDDR3
Memory Speed	800MHz
Memory Interface	512-bit
Memory Bandwidth	102GB/sec
Max Power Consumption	200 W peak, 160 W typical
System Interface	PCIe x16

Source: http://www.nvidia.com/object/product_tesla_c1060_us.html

Nvidia Tesla V100 (2017)

Extreme Performance for AI And HPC

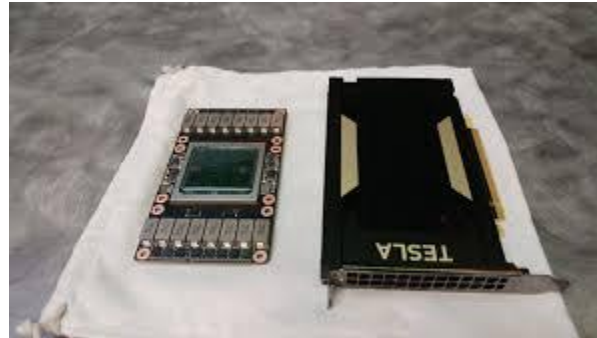


Figure 1. NVIDIA Tesla V100 SXM2 Module with Volta GV100 GPU

Nvidia Tesla GPUs

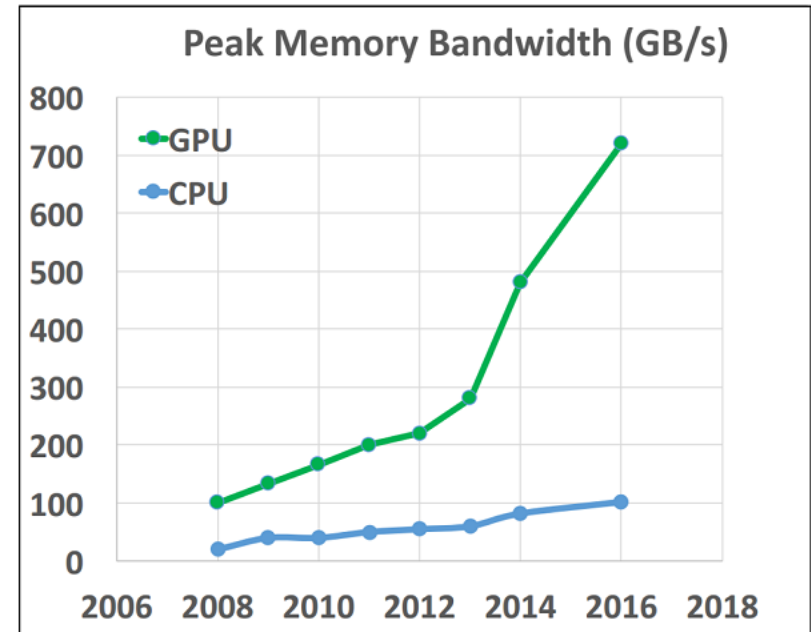
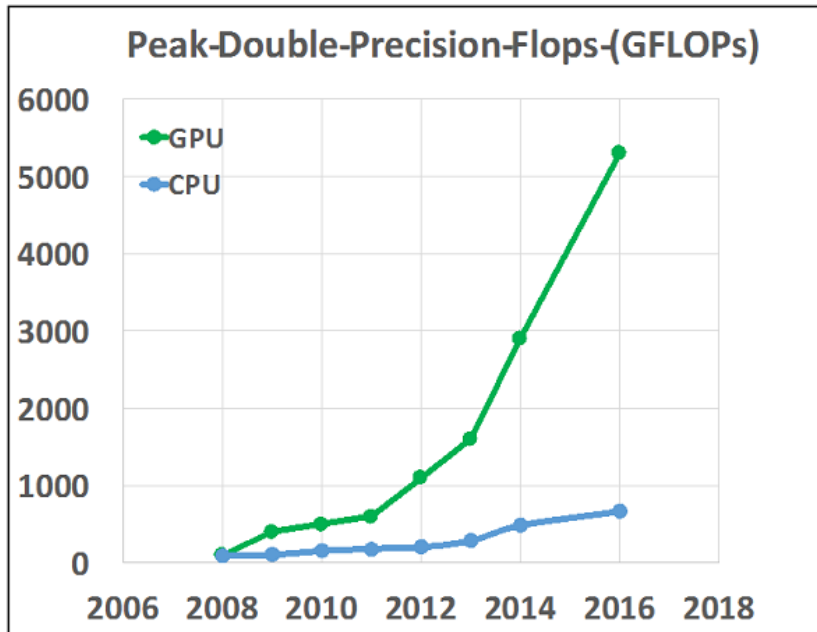
Table 1. Comparison of NVIDIA Tesla GPUs

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS ¹	5	6.8	10.6	15.7
Peak FP64 TFLOPS ¹	1.7	.21	5.3	7.8
Peak Tensor TFLOPS ¹	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm ²	601 mm ²	610 mm ²	815 mm ²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

¹ Peak TFLOPS rates are based on GPU Boost Clock

GPU Performance

- Calculation: 1 TFLOPS vs. 100 GFLOPS
- Memory Bandwidth: ~10x



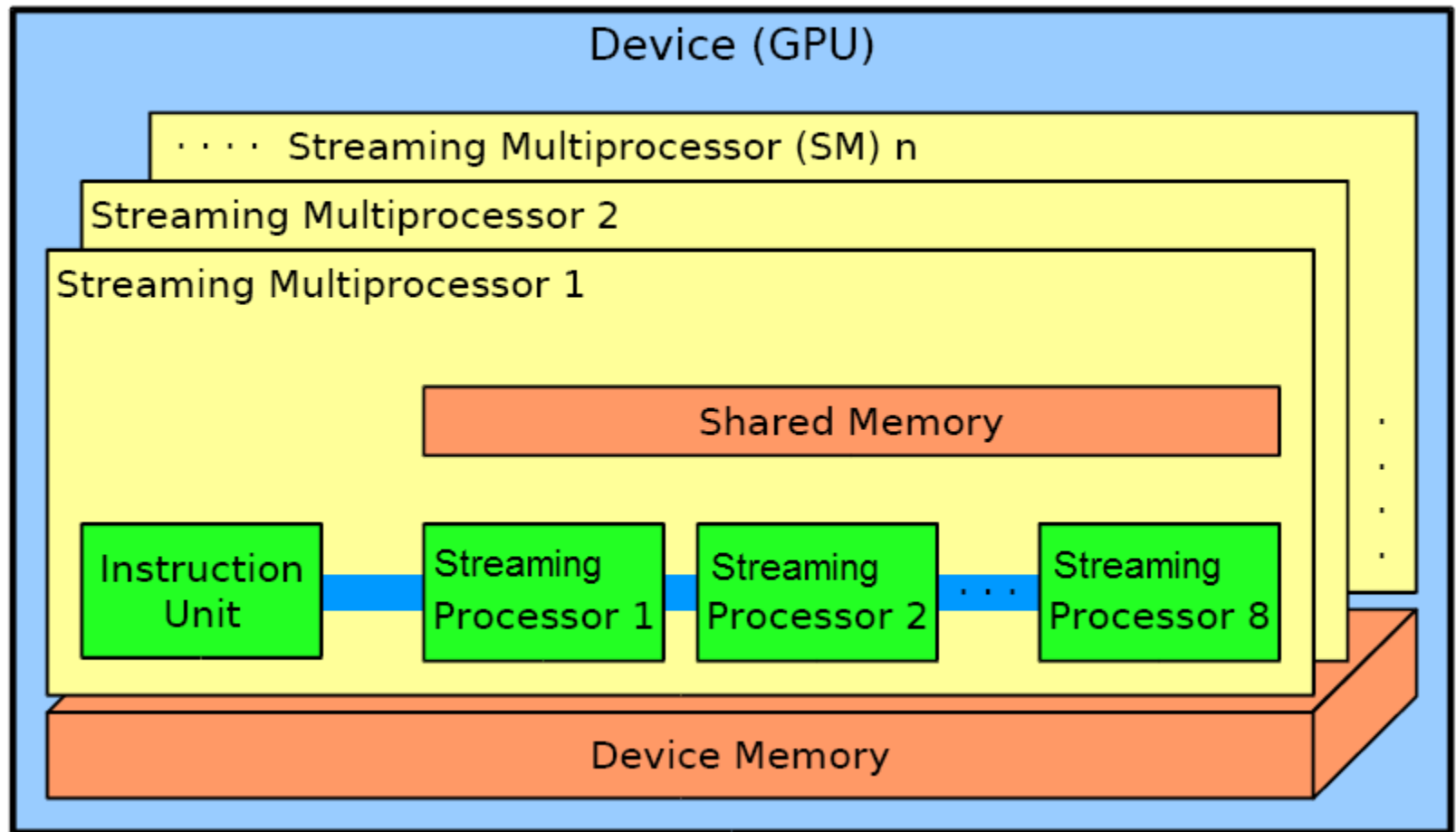
- GPU in every PC – massive volume and potential impact

CPU vs. GPU - Hardware



- CPU
 - General purpose compute cores
 - Each core can do complex a task
- GPU
 - More transistors devoted to data processing (compute cores) than cache and control
 - Each core has simple and specific instructions

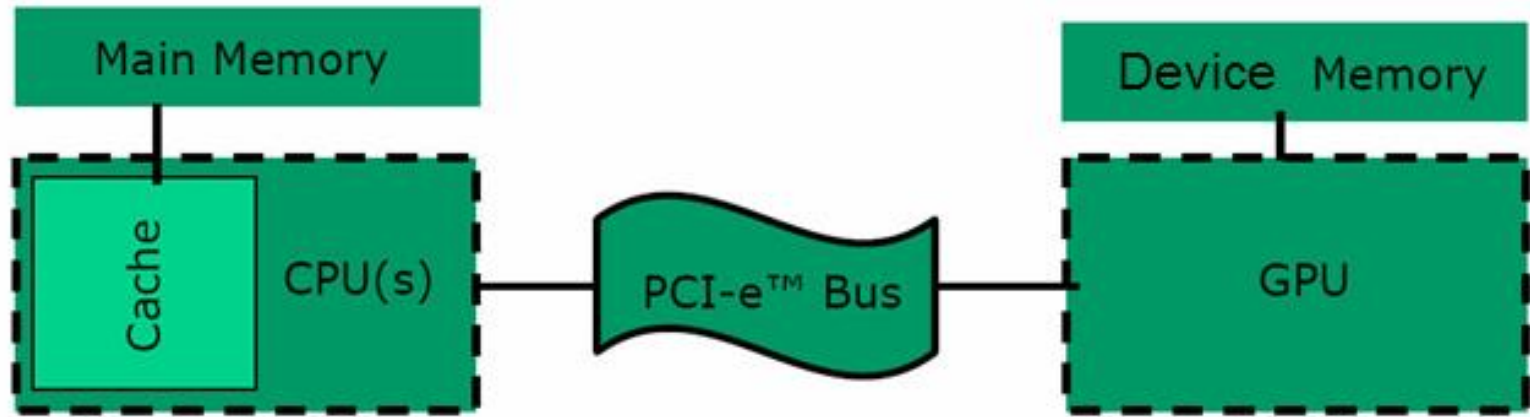
GPU Architecture Overview



GPU Architecture Overview

- Device (GPU) contains
 - High throughput global memory (or device memory)
 - A set of Streaming Multiprocessors (SM)
- Each Stream Multiprocessor contains
 - Instruction Units
 - Groups of Streaming Processor (SP) cores
 - Shared memory, shared by all SPs on the same SM
- All SPs in the same group works on the same instruction => SIMD (Single Instruction Multiple Data)

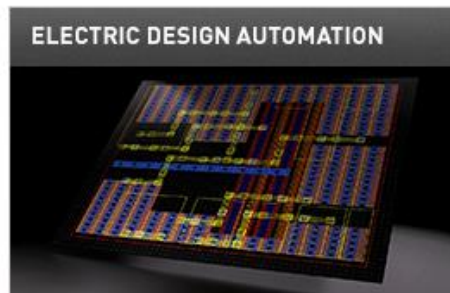
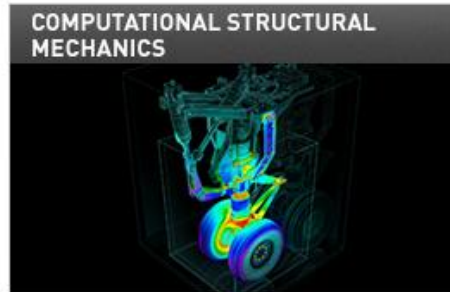
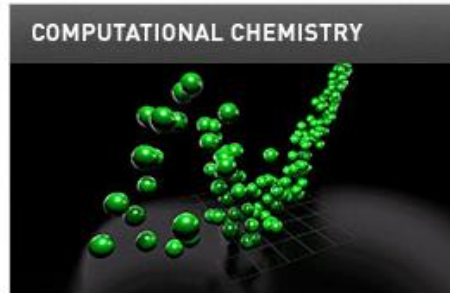
Host-GPU Communication



General Purpose Graphic Processing Unit (GPGPU)

- General Purpose computation using GPU and graphics API in applications other than 3D graphics
- Parallel applications leverage GPU
 - Large data arrays, streaming throughput
 - Fine-grain SIMD parallelism
 - Low-latency floating point (FP) computation
- CUDA (Compute Unified Device Architecture)
 - General purpose programming model developed by Nvidia
 - It is an extension to the ANSI C programming language
 - Low learning curve

GPU Applications

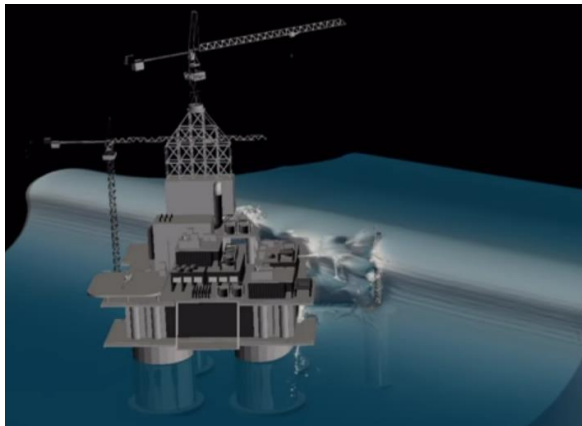


<http://www.nvidia.com/object/gpu-applications-domain.html>

GPU in Physics Simulation Demo



https://www.youtube.com/watch?v=ffgRC3kvA_k



<https://www.youtube.com/watch?v=B8mP9E75D08>

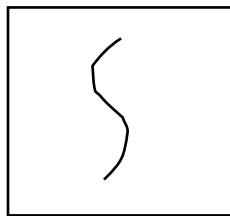
GPU for Mining Cryptocurrency



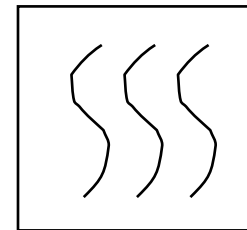
<https://www.nme.com/news/gaming-news/700000-graphics-cards-were-bought-by-crypto-miners-in-early-2021-alone-2973241>

Shared Memory Programming

- Programming model for Shared-Memory Multiprocessor and GPU
- A process can have multiple threads
- A thread is a flow of program execution
- Programmers create threads using language constructs, e.g. Pthreads, Java Threads, Windows Threads
- Nvidia GPU uses **CUDA** Threads

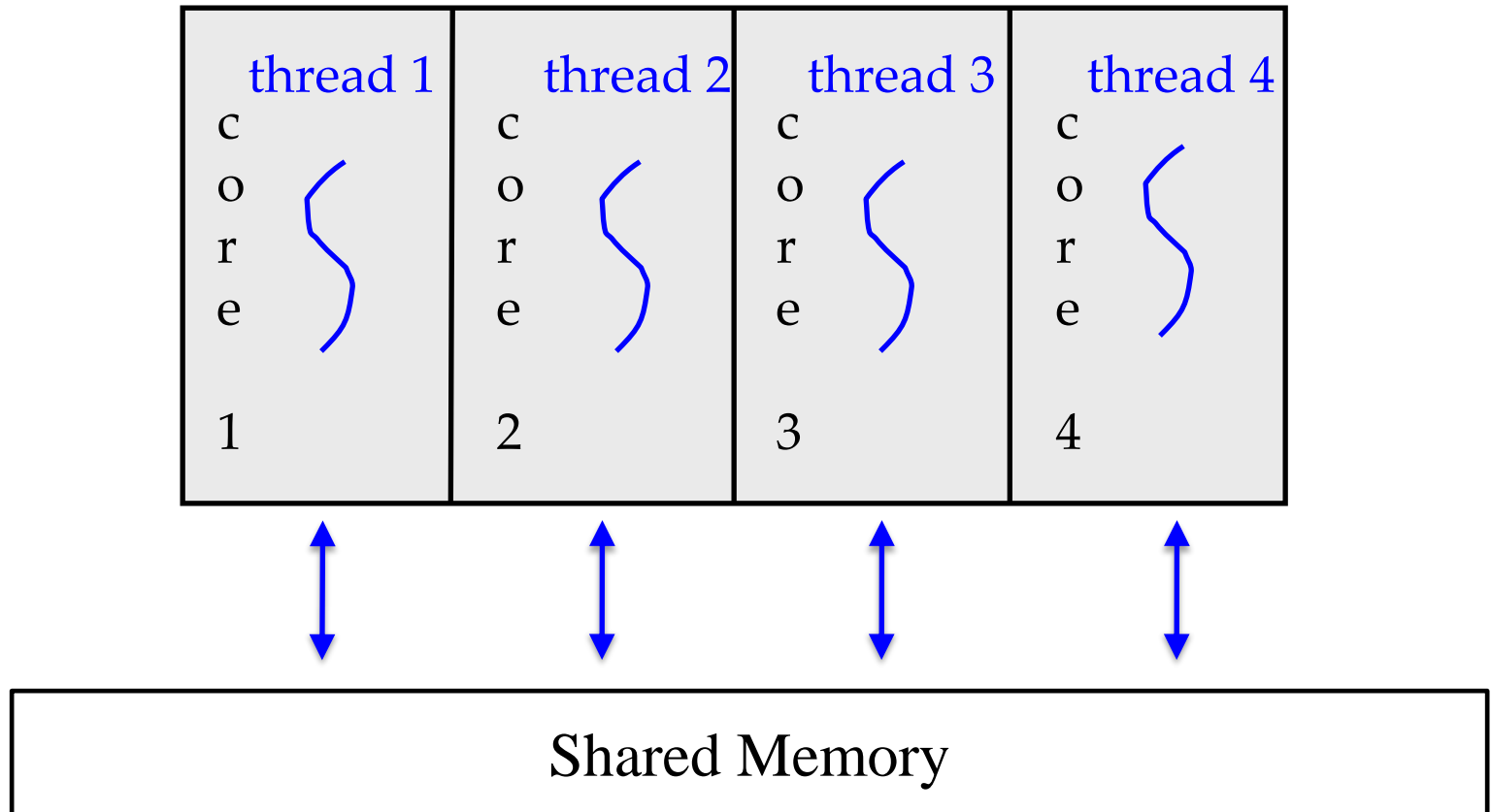


All processes have a primary thread



A process can have multiple threads

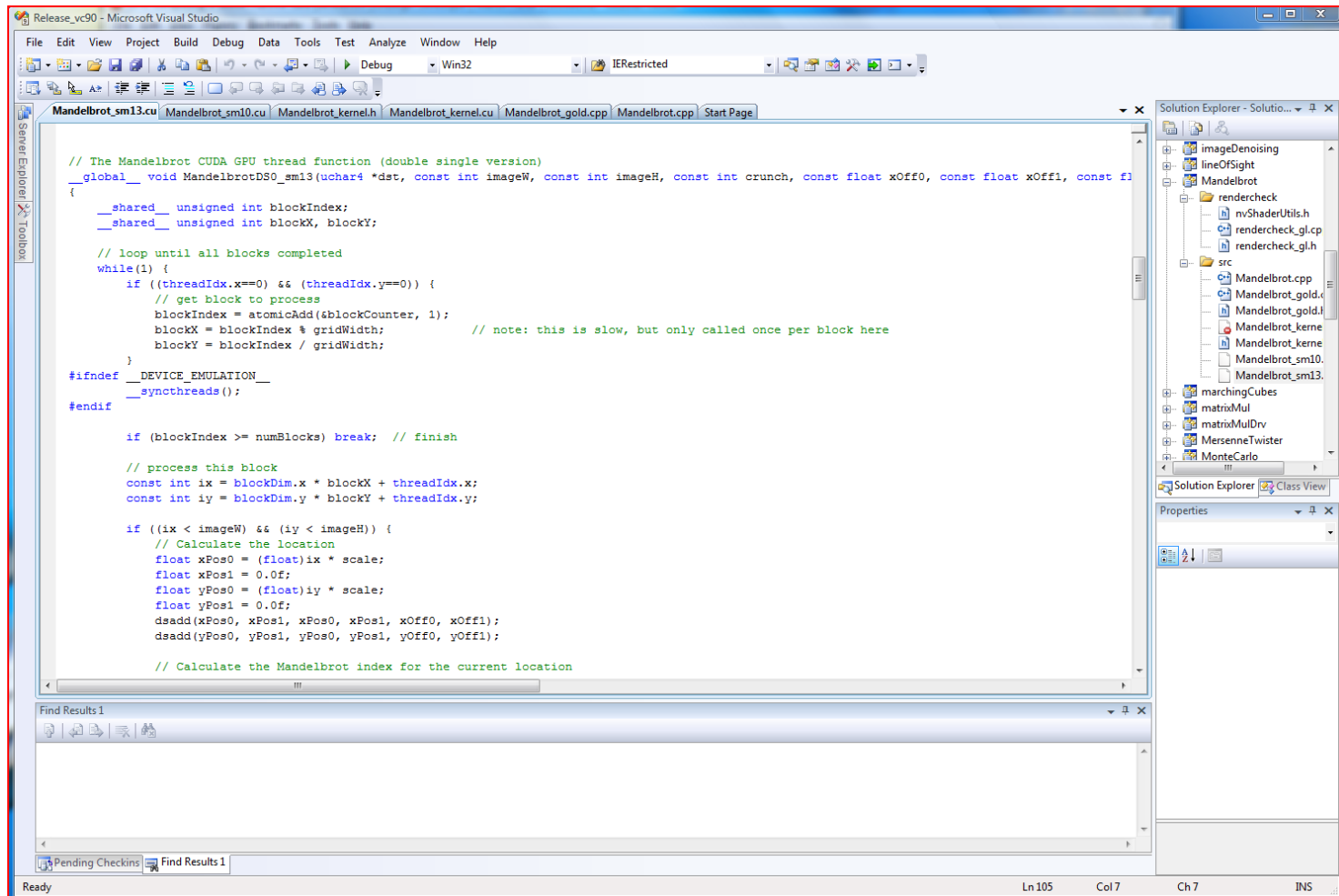
Threads run in parallel



Software Requirements/Tools

- CUDA Driver and Software Development Kit
<https://developer.nvidia.com/cuda-downloads>
 - Driver
 - Compiler
- C program editor
 - vi / nano / Microsoft Visual Studio / Notepad
 - CUDA source file has extension **.cu**

CUDA Program in Visual Studio



How to Compiling and Run

- To compile (command line) for GPU

`nvcc -o example example.cu`

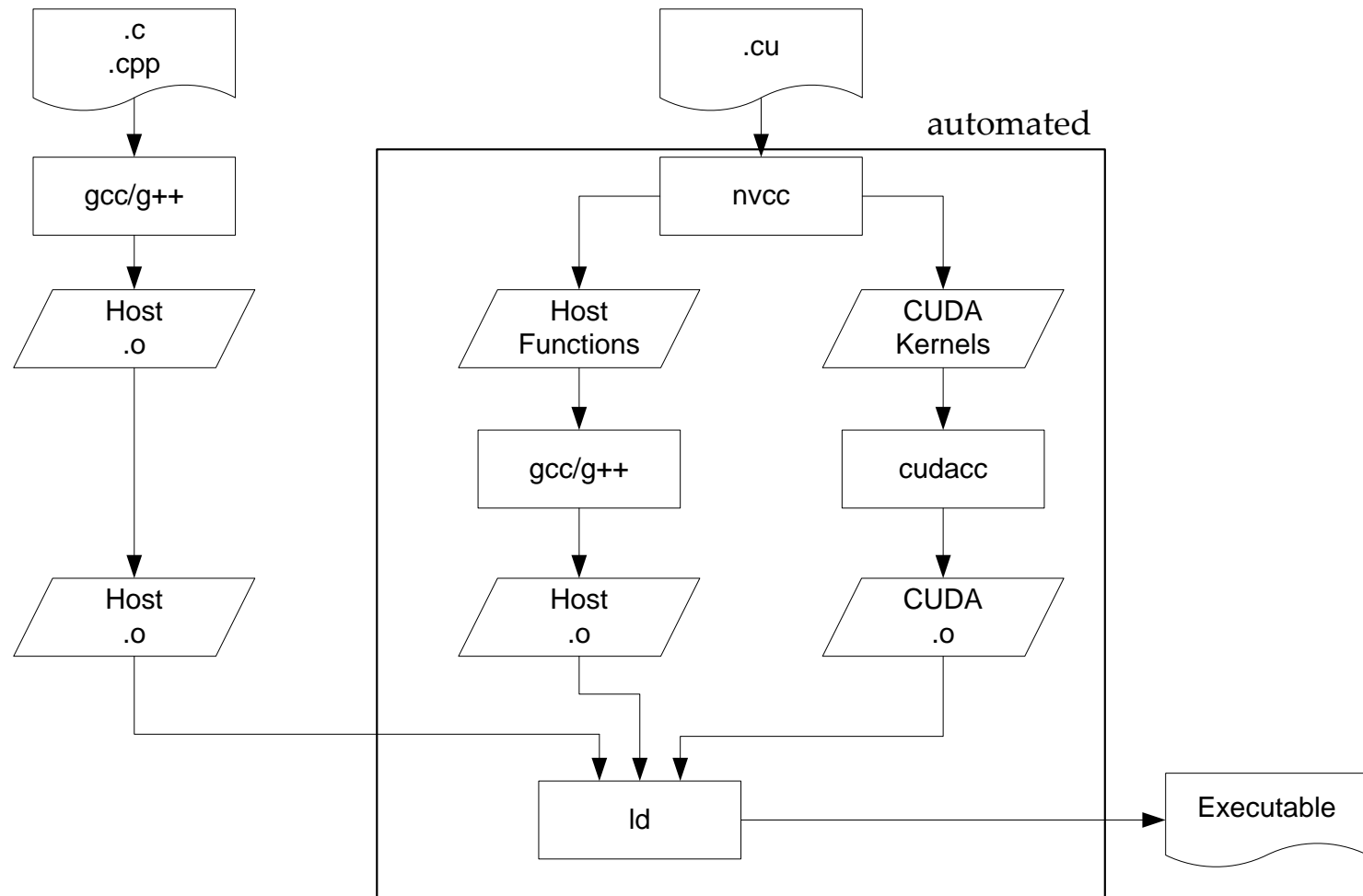
- To compile in **simulation mode**
- (The program will execute on CPU; no GPU)

`nvcc -deviceemu -o example example.cu`

- To run a CUDA program

`./example`

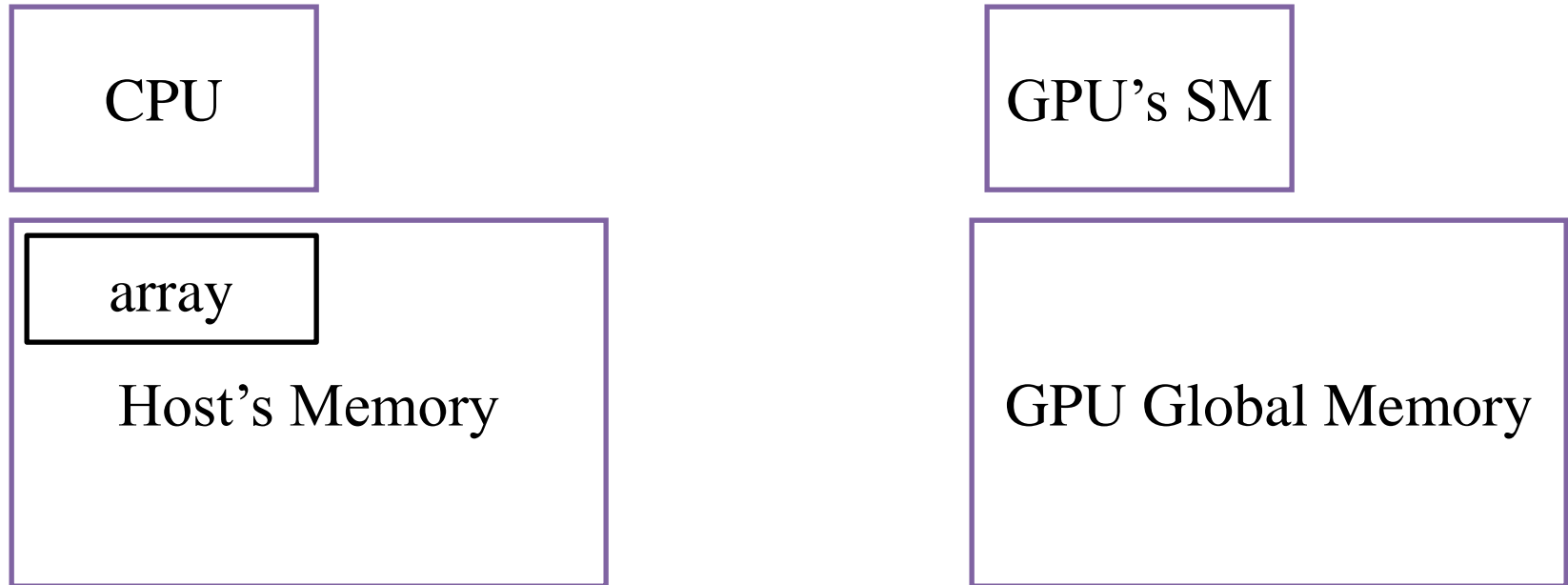
CUDA Compilation with nvcc



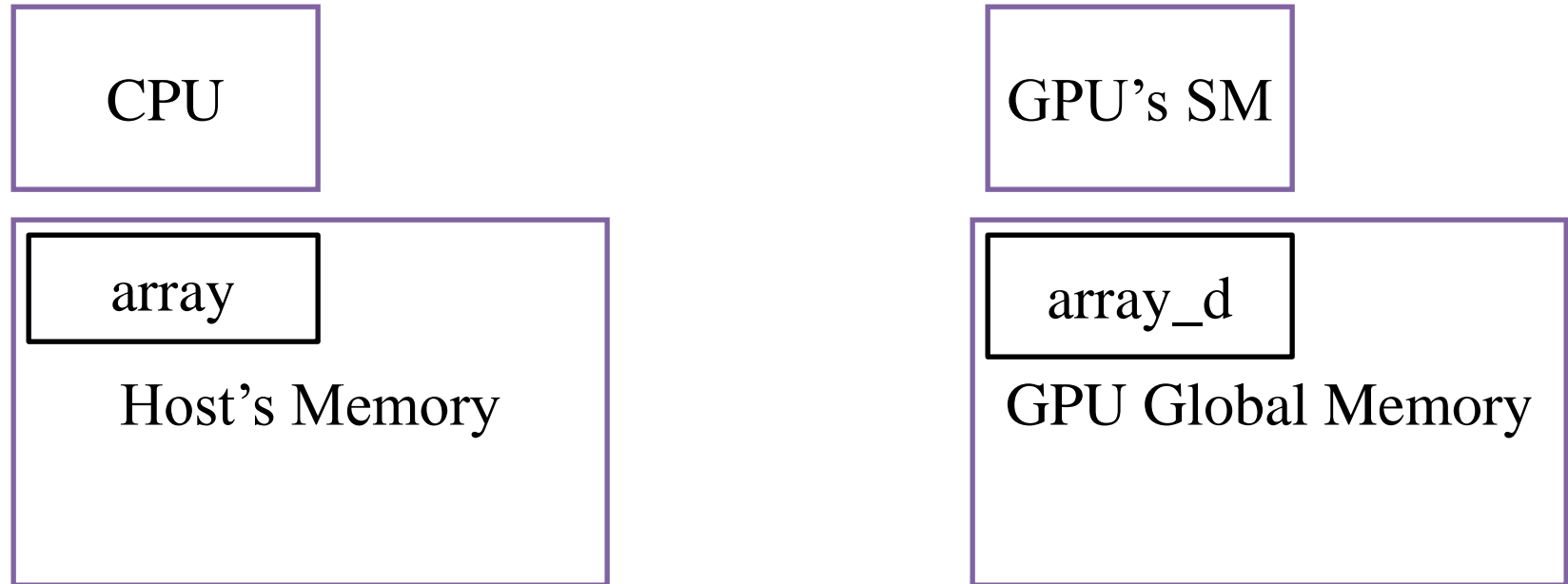
To compute with GPU

1. We need to allocate space in the video card's memory for the variables.
2. The video card does not have I/O devices, hence we need to copy the input data from the memory in the host computer into the memory in the video card, using the variable allocated in the previous step.
3. We need to specify code (called *kernel*) to execute.
4. Copy the results back to the memory in the host computer.

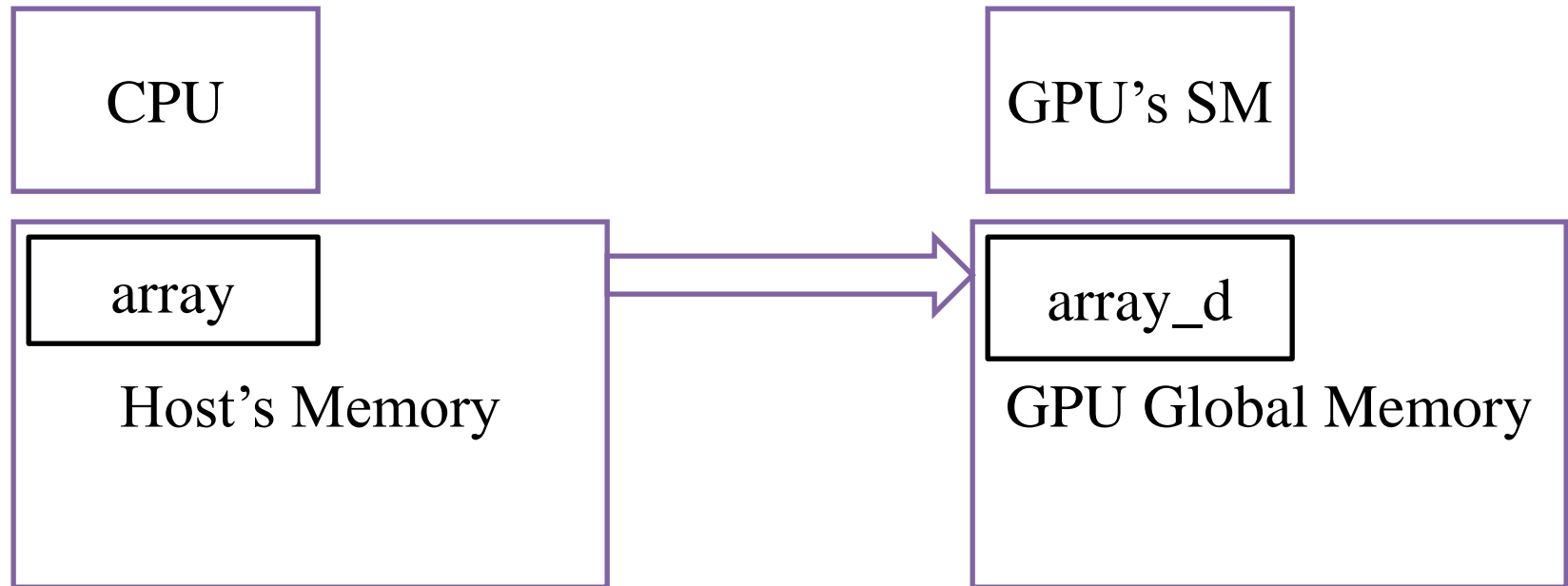
Initially:



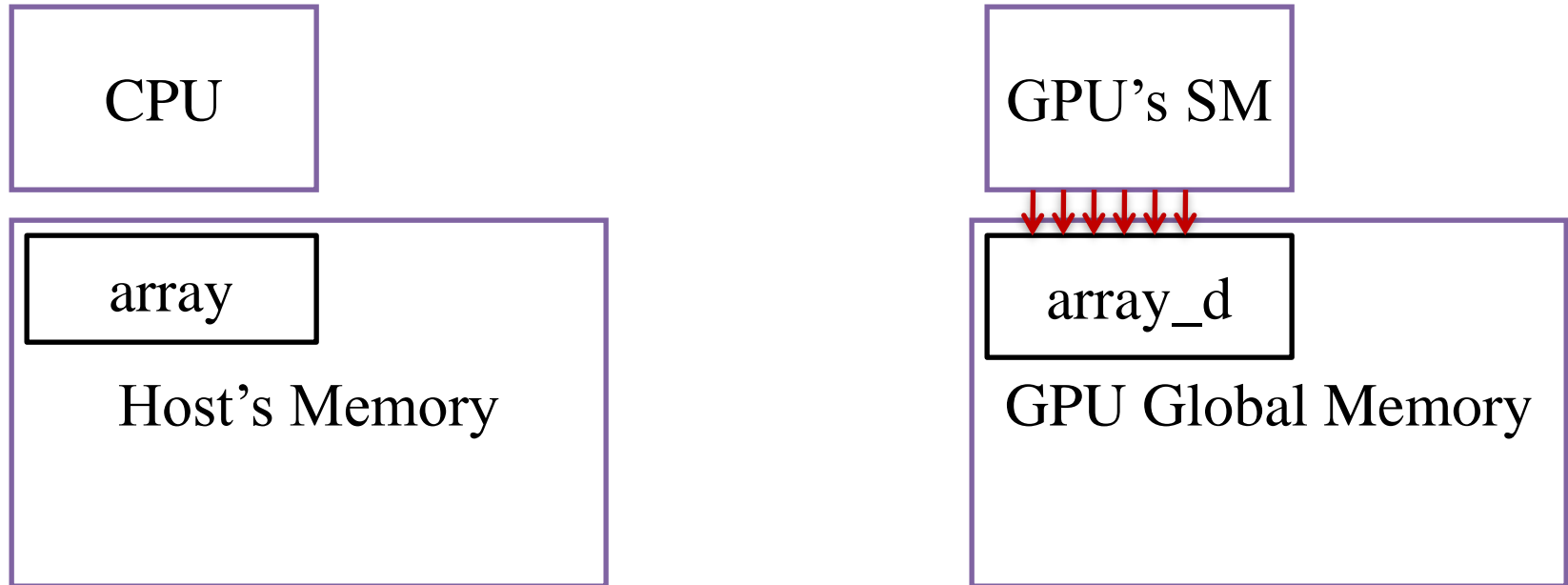
1) Allocate Memory in the GPU card



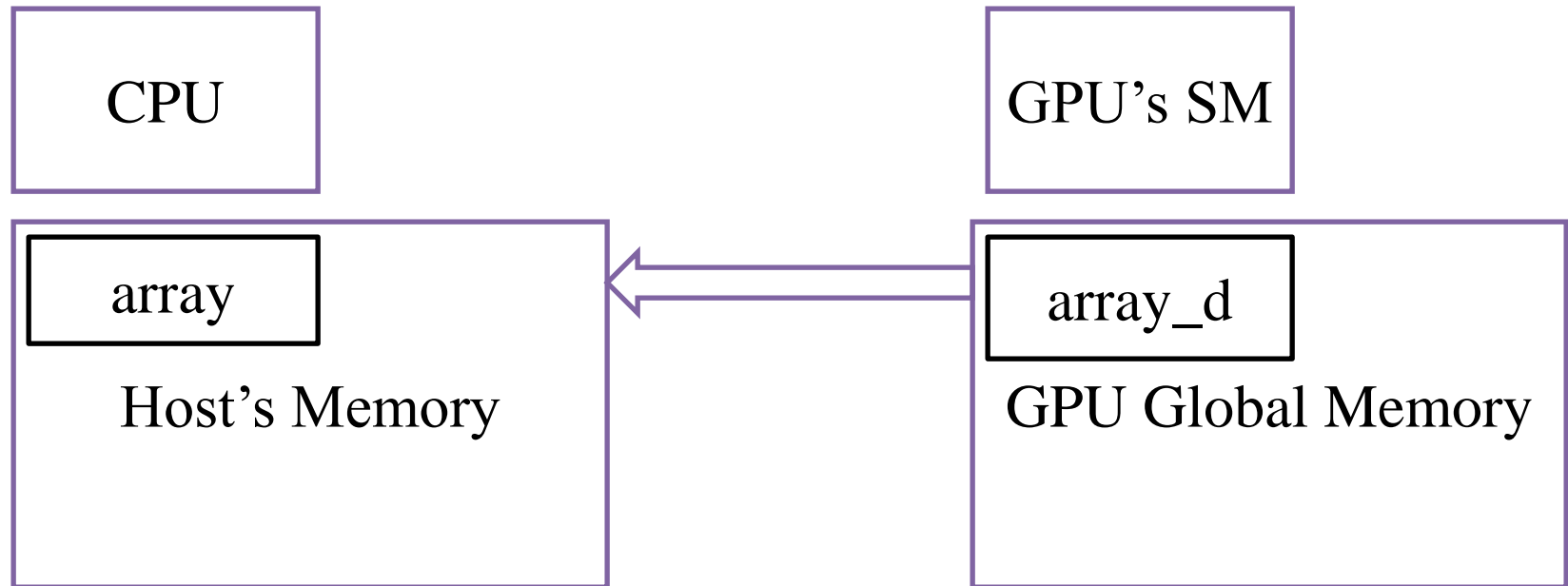
2) Copy content from the host's memory to the GPU card memory



3) Execute code on the GPU



4) Copy results back to the host memory



Example: Vector addition

A	1	3	8	3
	+	+	+	+
B	8	3	1	1
C	9	6	9	4

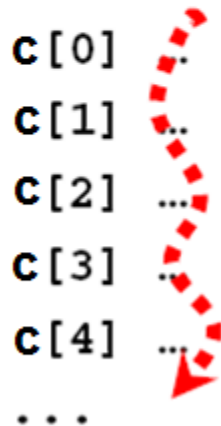
```
#define N 4
void vectAdd(int *A, int *B, int *C)
{
    for (int i = 0; i < N; i++)
    {
        C[i] = A[i] + B[i];
    }
}
```

Parallel Threads

Sequential Code

```
for (i=0; i < N; i++)  
{  
    c[i] = a[i] + b[i];  
}
```

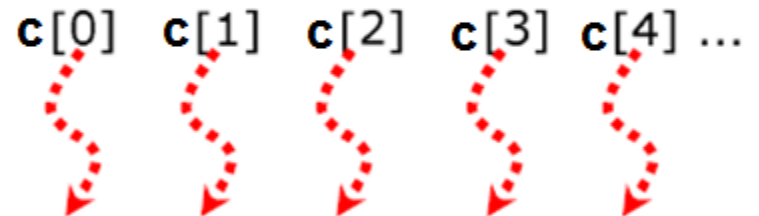
c[0] ..
c[1] ...
c[2]
c[3] ...
c[4] ...
...



Parallel Code

```
/* assuming you have one  
thread per array entry  
*/  
i = threadID;  
c[i] = a[i] + b[i]
```

c[0] c[1] c[2] c[3] c[4] ...



Time

Basic CUDA program structure

```
int main (int argc, char *argv[] ) {
```

1. Allocate memory space in device (GPU) for data

2. Copy data from CPU to GPU

**3. Call “kernel” routine to execute on GPU
(with CUDA syntax that defines no of threads and their
physical structure)**

4. Transfer results from GPU to CPU

5. Free memory space in device (GPU)

```
return;
```

```
}
```

From Host (CPU) Code and Data

```
#define N 256
```

```
...
```

```
int  a[N], b[N], c[N];
```

```
// a[N] is for vector A
```

```
// b[N] is for vector B
```

```
// c[N] is for vector C
```

```
for (int i = 0; i < N; i++)
```

```
{
```

```
    c[i] = a[i] + b[i];
```

```
}
```


1. Allocating global memory space in “device” (GPU) for data

Use CUDA malloc routines:

```
int size = N * sizeof( int);    // space for N integers
```

```
int *devA, *devB, *devC;    // devA, devB, devC ptrs
```

```
cudaMalloc( (void**)&devA, size );
```

```
cudaMalloc( (void**)&devB, size );
```

```
cudaMalloc( (void**)&devC, size );
```

2. Copy data from host (CPU) to device (GPU)

Use CUDA routine `cudaMemcpy`

```
cudaMemcpy( devA, a, size, cudaMemcpyHostToDevice);  
cudaMemcpy( devB, b, size, cudaMemcpyHostToDevice);
```

where `devA` and `devB` are pointers to destination in device
and `a` and `b` are pointers to host data

3. Call “kernel” routine to execute on device (GPU)

CUDA introduces a syntax addition to C:

Triple angle brackets mark call from host code to device code. Contains organization and number of threads in two parameters:

```
myKernel<<< n, m >>>(arg1, ... );
```

n and **m** will define organization of thread blocks and threads in a block.

For now, we will set **n = 1**, which say one block and **m = N**, which says N threads in this block.

arg1, ... , -- arguments to routine **myKernel** typically pointers to device memory obtained previously from **cudaMalloc**.

The CUDA Kernel and Thread

- A piece of code executed on GPU
- A kernel is executed by parallel threads on processors in the GPU card
- All threads run the same code
- Each thread has a unique ID
- Each thread uses ID to know which piece of to work on

Declaring a Kernel Routine

A kernel defined using CUDA specifier `__global__`

Example – Adding to vectors A and B

```
#define N 256
```

```
__global__ void vecAdd(int *A, int *B, int *C) { // Kernel definition
```

```
    int i = threadIdx.x; ← thread ID
```

```
    C[i] = A[i] + B[i];
```

```
}
```

Each of the N threads performs one pair-wise addition:

Thread 0: $C[0] = A[0] + B[0];$

Thread 1: $C[1] = A[1] + B[1];$

...

```
int main() {
```

```
    // allocate device memory &
```

```
    // copy data to device
```

```
    // device mem. ptrs devA,devB,devC
```

```
    vecAdd<<<1, N>>>(devA,devB,devC);
```

```
    ...
```

```
}
```

Grid of one block having N threads

4. Transferring data from device (GPU) to host (CPU)

Use CUDA routine `cudaMemcpy`

```
cudaMemcpy( c, devC, size, cudaMemcpyDeviceToHost);
```

where **devC** is a pointer in device and **c** is a pointer in host.

5. Free memory space in “device” (GPU)

Use CUDA `cudaFree` routine:

```
cudaFree( devA);  
cudaFree( devB);  
cudaFree( devC);
```

vecAdd.cu: A Complete CUDA Program

```
#include <stdio.h>

#define N 256

__global__ void vecAdd(int *A, int *B, int *C) {
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main (int argc, char *argv[] ) {

    int i;
    int size = N *sizeof( int);
    int  a[N], b[N], c[N], *devA, *devB, *devC;

    for (i=0; i < N; i++) {
        a[i] = 1; b[i] = 2;    // initialize a[] and b[]
    }

    cudaMalloc( (void**)&devA, size);
    cudaMalloc( (void**)&devB, size);
    cudaMalloc( (void**)&devC, size);

    cudaMemcpy( devA, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy( devB, b, size, cudaMemcpyHostToDevice);
```



```
vecAdd<<<1, N>>>(devA, devB, devC);

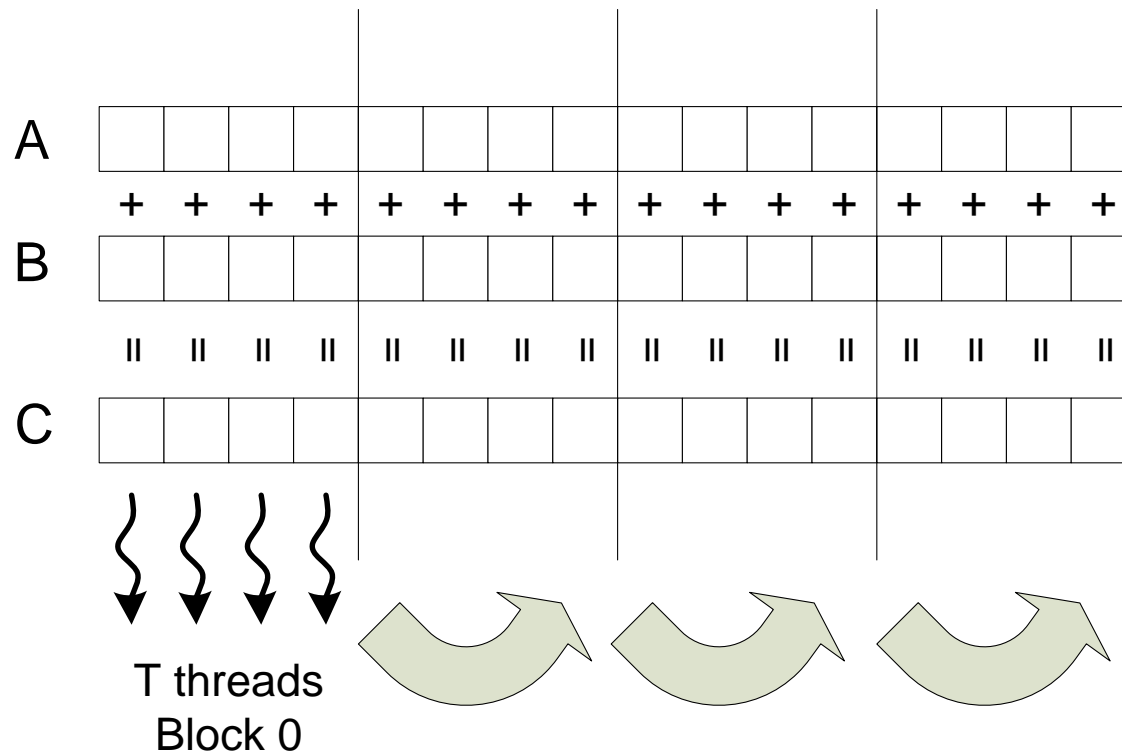
cudaMemcpy( c, devC, size, cudaMemcpyDeviceToHost);
cudaFree( devA);
cudaFree( devB);
cudaFree( devC);

for (i=0; i < N; i++) {
    printf("%d ",c[i]);
}
printf("\n");
}
```

Vector addition when data is large ($n > T$)

n = data size

T = number of threads



Note: A better version will be studied in the next lecture

vecAdd2.cu

```
#include <stdio.h>

#define n 1024
#define T 256

__global__ void vecAdd(int *A, int *B, int *C) {
    int i;

    for (i = threadIdx.x; i < n; i = i + T) {
        C[i] = A[i] + B[i];
    }
}

int main (int argc, char *argv[] ) {

    int i;
    int size = n * sizeof( int);
    int  a[n], b[n], c[n], *devA, *devB, *devC;

    for (i=0; i < n; i++) {
        a[i] = 1; b[i] = 2;
    }
}
```

```
cudaMalloc( (void**) &devA, size);
cudaMalloc( (void**) &devB, size);
cudaMalloc( (void**) &devC, size);

cudaMemcpy( devA, a, size, cudaMemcpyHostToDevice);
cudaMemcpy( devB, b, size, cudaMemcpyHostToDevice);

vecAdd<<<1, T>>>(devA, devB, devC);

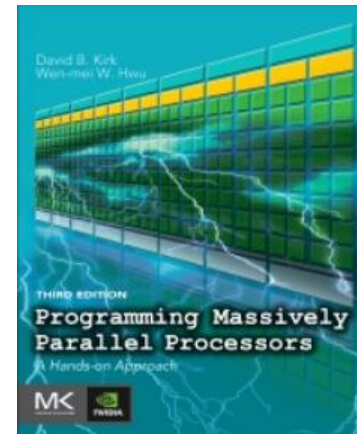
cudaMemcpy( c, devC, size, cudaMemcpyDeviceToHost);
cudaFree( devA);
cudaFree( devB);
cudaFree( devC);

for (i=0; i < n; i++) {
    printf("%d ", c[i]);
}
printf("\n");


}
```

References

- Kirk and Hwu, Programming Massively Parallel Processors: A Hands-on Approach 3rd, Morgan Kaufmann, 2016.
 - Chapter 1 - 4



- Draft of the first edition available for download

Two vertical lines, one blue and one orange, are positioned on the left side of the slide.

Q & A