INTRODUCTION TO GPU COMPUTING

Marwan Abdellah
Blue Brain Project, EPFL

December - 16, 2012

Agenda

- What (was, is) the GPU?
- Why GPU Computing?
- GPU Vs. CPU
  - Architectures
- GPU APIs
- CUDA Programming Model
- CUDA API Basics
- Vector Addition Example
- GPU (CUDA)-accelerated Libraries

GPU was

- It was a **Graphics (ONLY)** Processing Units
- A Processor added to the computer to accelerate graphics operations.
- It addresses the demands of real-time high-resolution 3D graphics compute-intensive tasks.

Now, GPU is

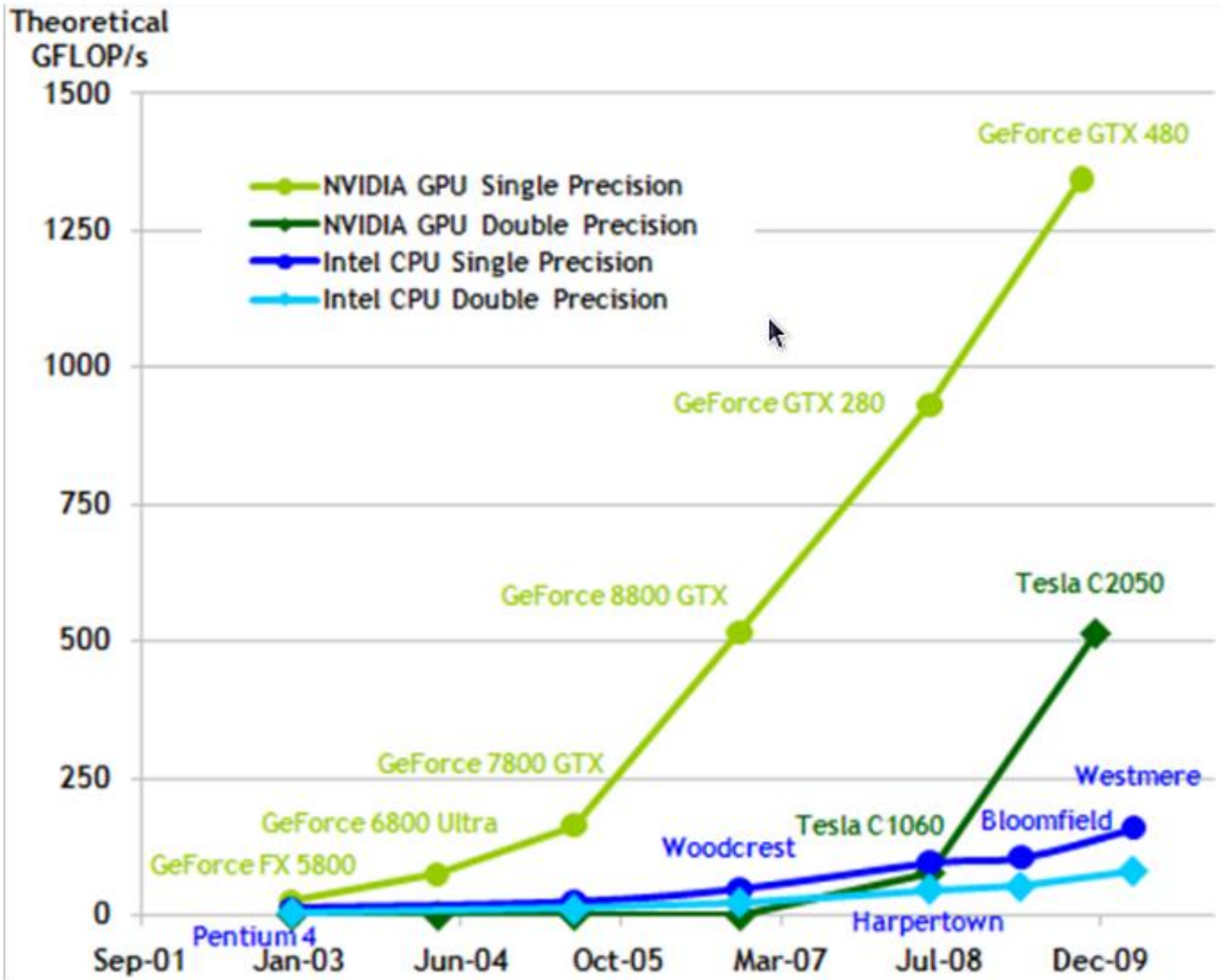
- High performance co-processor linked to the CPU to solve complex computational data-parallel problems.
- Massively Parallel Floating-Point CoProcessors

Why GPU Computing?

- Inexpensive (Compared to a Multi-Core CPU)
- Idle (Unless you are playing Games !!!)
- Designed to well perform for handling floating point arithmetic.
- Outperform the CPU on a per-$ basis.

	Intel Quad Core Xeon	NVIDIA GTX 257
FP Performance	68 GFlops	304 GFlops
Memory Bandwidth	19/GB/s	127 GB/sec
Cost	1500 $	300+ $

Performance: CPU Vs. GPU

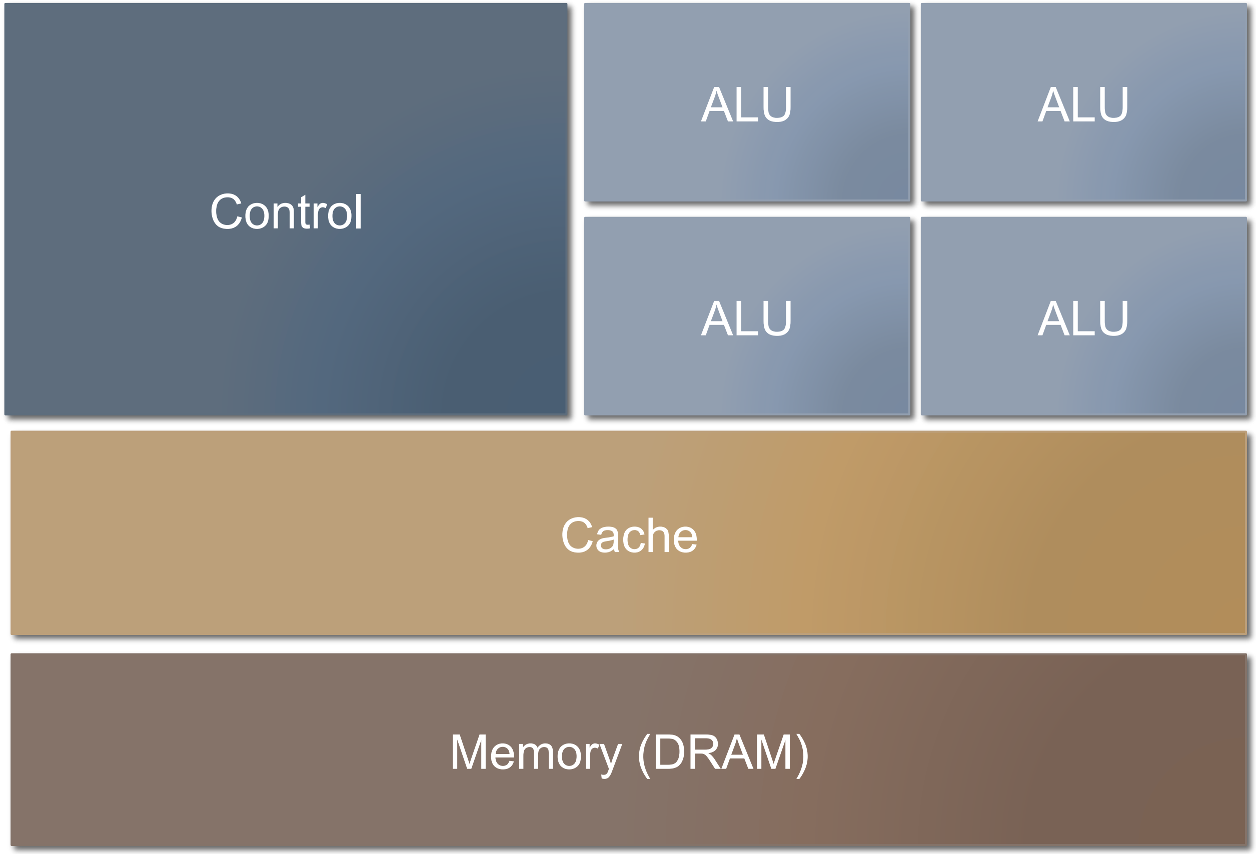


Tesla Arch.

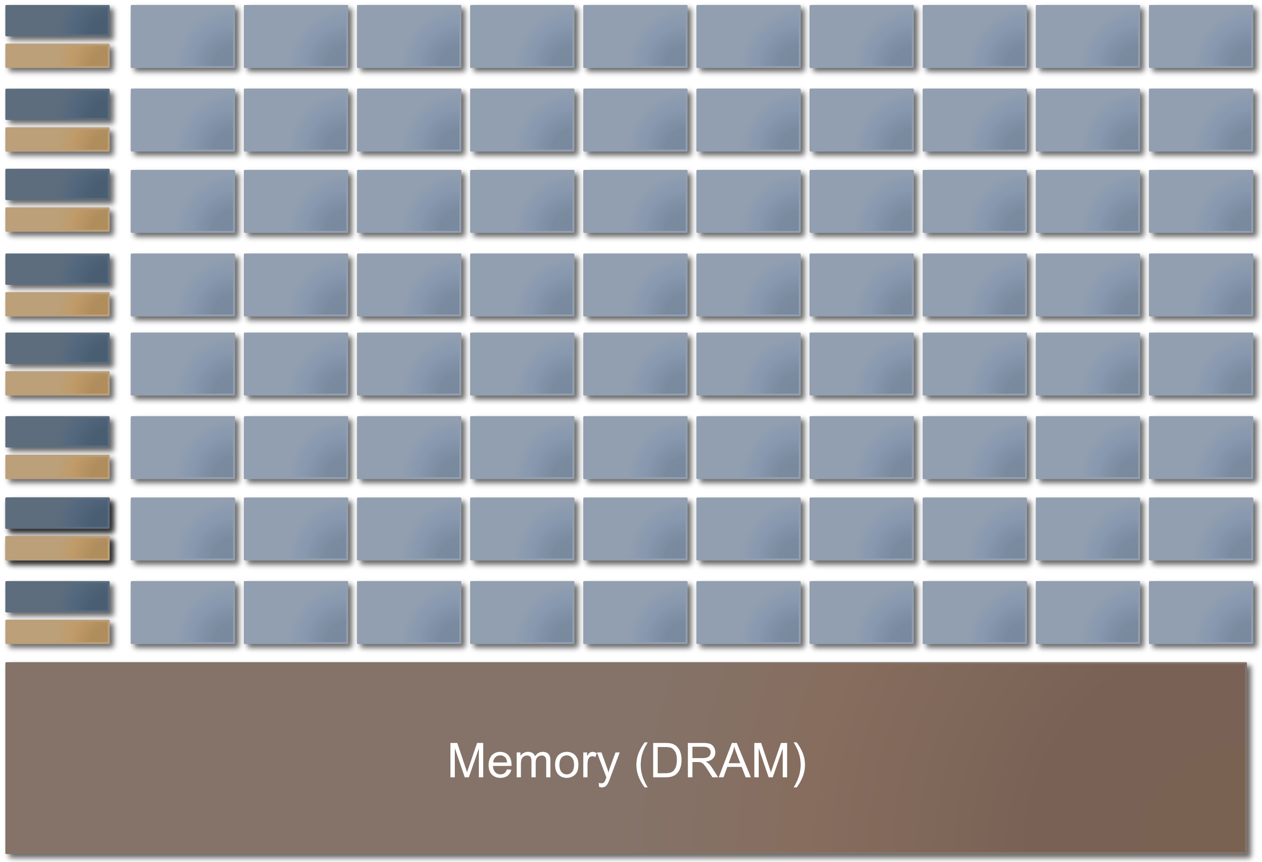
Kepler GTX 680

Multi-GPU System

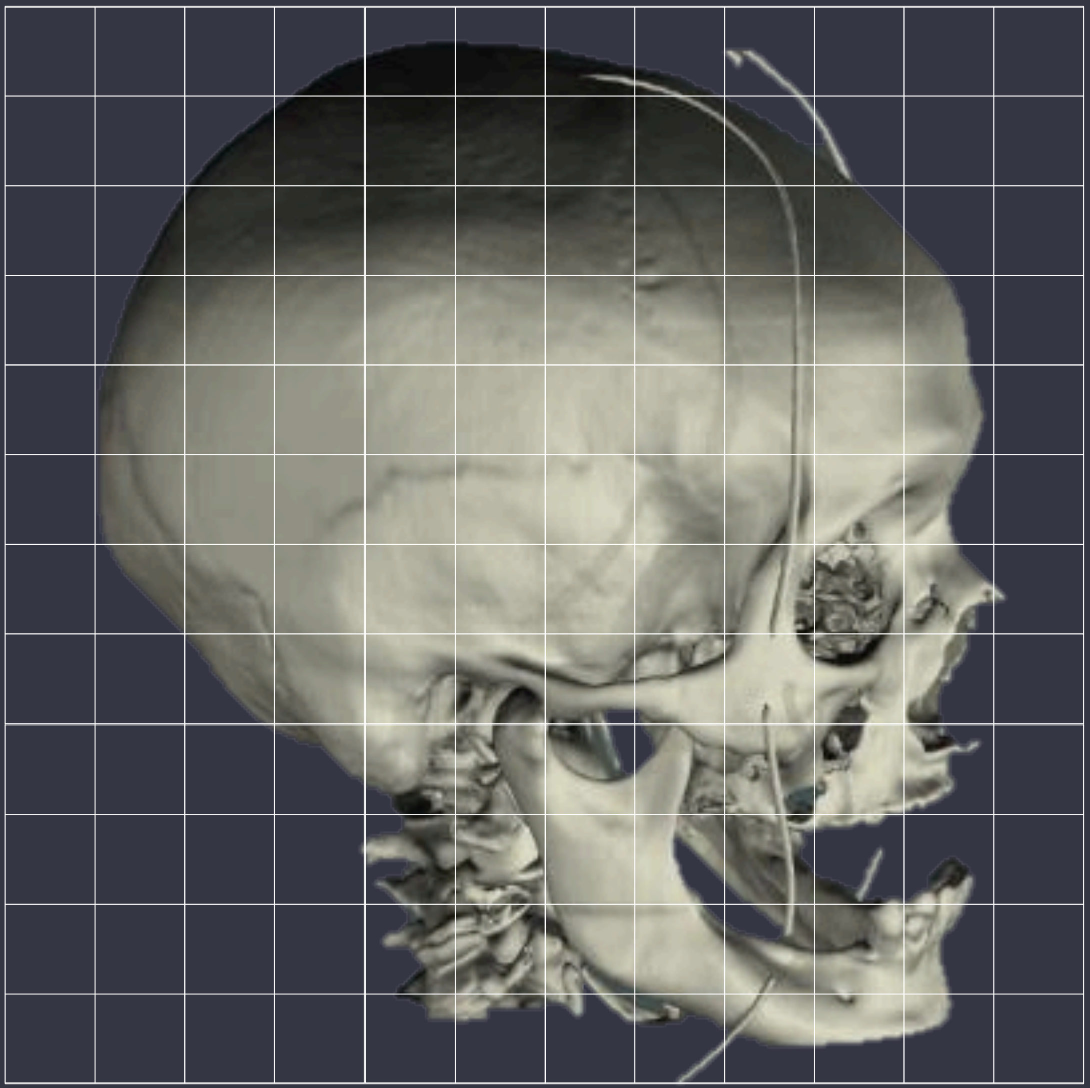
Architectures (CPU)



Architectures (GPU)



Use Case: Image Correction

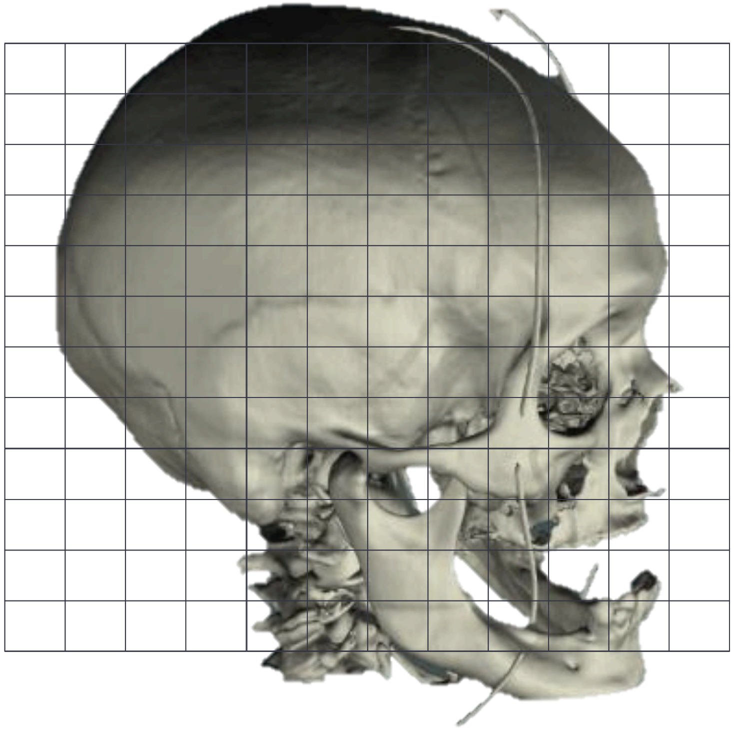


CPU-Approach

This operation could be done in a C programming environment by a simple `for` Loop

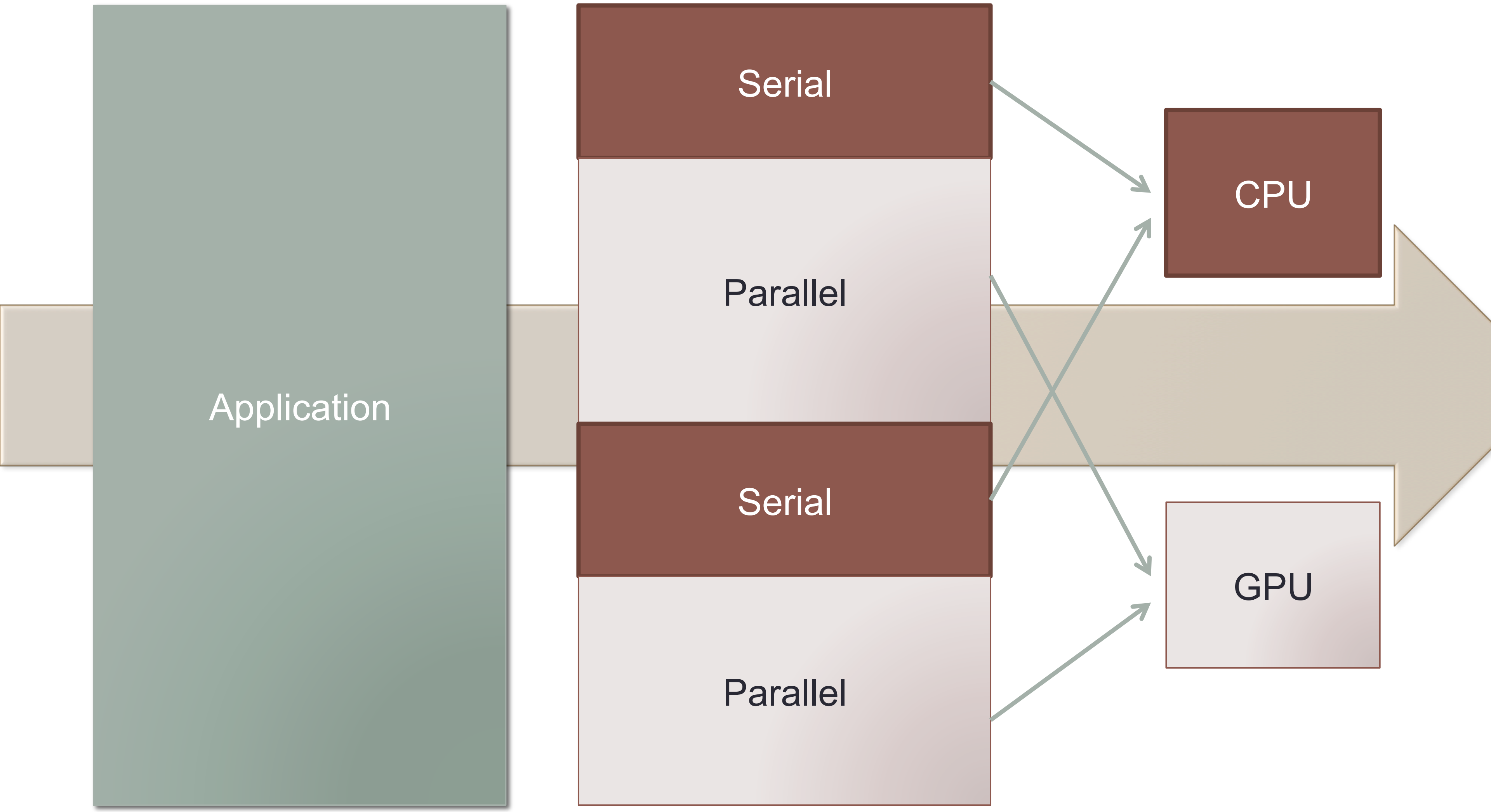
```
for (int i = 0; i < numPixels; i++)
{
    pixel[i] += 10;
}
```

GPU-Approach



Quite Similar !

To Make it Short (Application Flow)



GPU Evolution

- Multi core design
- SIMD core optimized for floating point arithmetic
- Dozens of multi cores per card

- Early GPGPU Efforts
  - Fixed Function Pipelines (Graphics APIs)
    - (OpenGL or DirectX Mapping)
- Programmable Pipelines
  - Shader Mapping
    - GLSL
    - NVIDIA Cg
    - Microsoft HLSL
- **GPU Programming APIs**
  - **Unleash your application**

APIs

- **CUDA (Compute Unified Device Architecture)**
  - Released by NVIDIA
  - Supported on all modern NVIDIA GPUs (notebook GPUs, high-end GPUs, mobile devices)
- OpenCL (Open Computing Language)
  - Released by Apple, and now under the maintenance of Khronous.
  - Open standard, targeting NVIDIA, AMD/ATI GPUs, Cell, multicore x86, ..

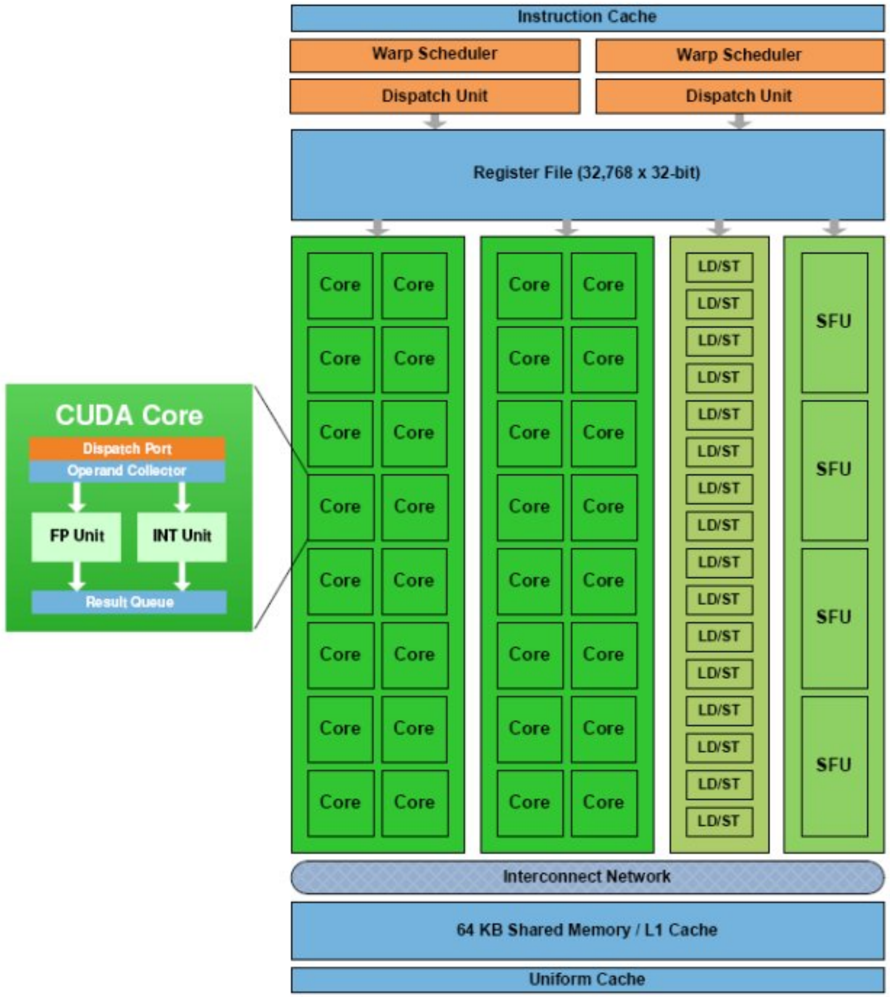
CUDA

- Parallel Computing Platform and programming model released by NVIDIA in 2004.
- Version **5.0** has been released in October 2012.
- The CUDA platform is accessible to software developers through
  - CUDA-accelerated libraries
  - Compiler directives
  - Extensions to industry-standard programming languages, including C, C++ and Fortran.
- CUDA provides both a **low level API** and a **higher level API**.
- Supported GPUs.

CUDA

- To Start writing Hello World (Vector Addition)on CUDA, we have to understand
  - GPU Architecture
  - Basic Terminology
  - CUDA Programming Model
  - CUDA Threading Hierarchy
  - CUDA Memory Model

CUDA Architecture

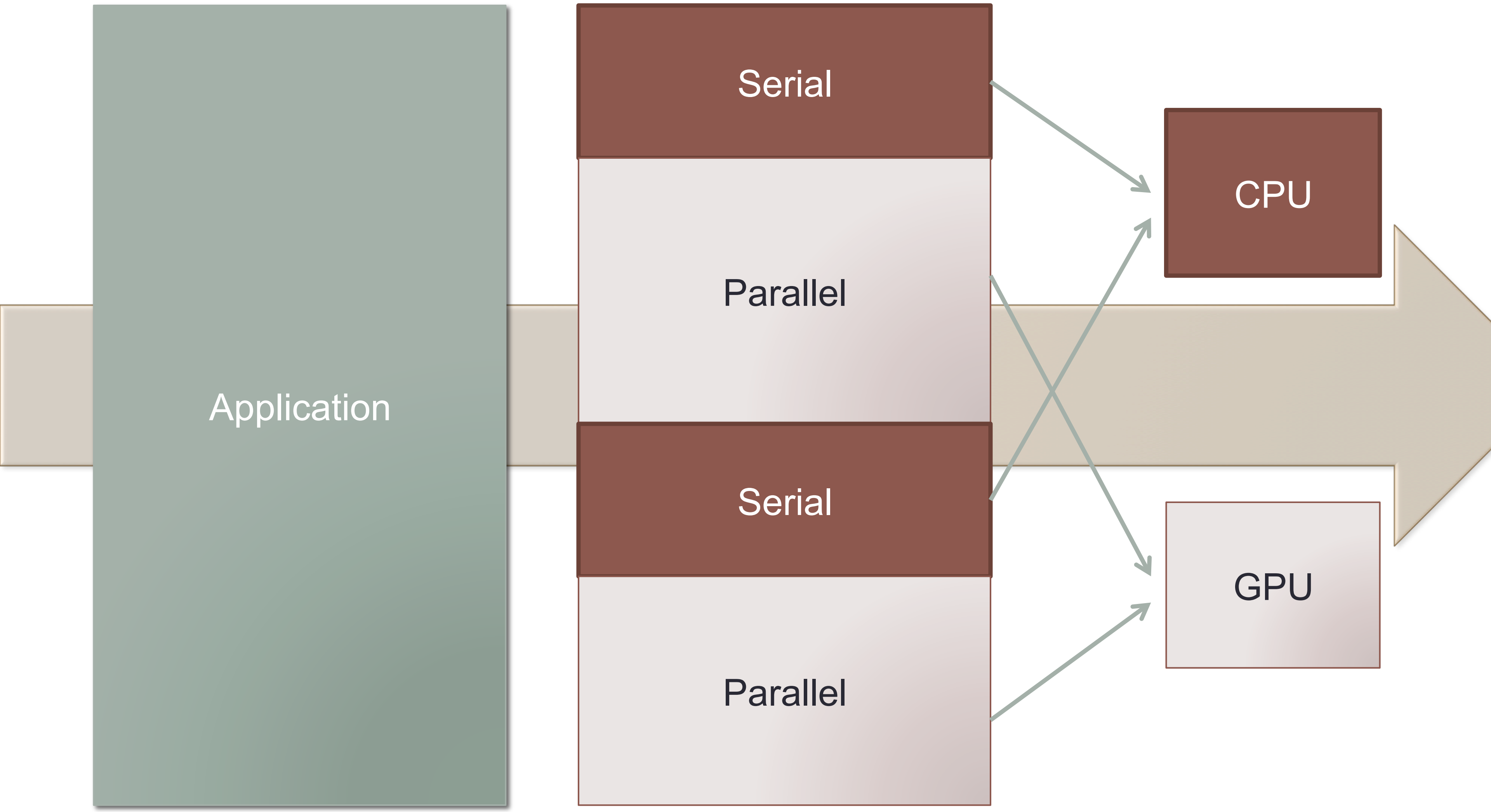


Fermi Streaming Multiprocessor (SM)

GPU Programming Model

- GPU = **Compute Device** (Compute Capability)
- GPU can execute a **portion** of an application that
  - Has to be executed many times (as a `for` loop)
  - Can be isolated as a **function**
  - Works independently on different data
- This function is compiled to be executed on the device as a **kernel**.
- The kernel will be executed on the different data processing thousands of elements of the data in parallel.

Programming Model (Recall)



Programming Model

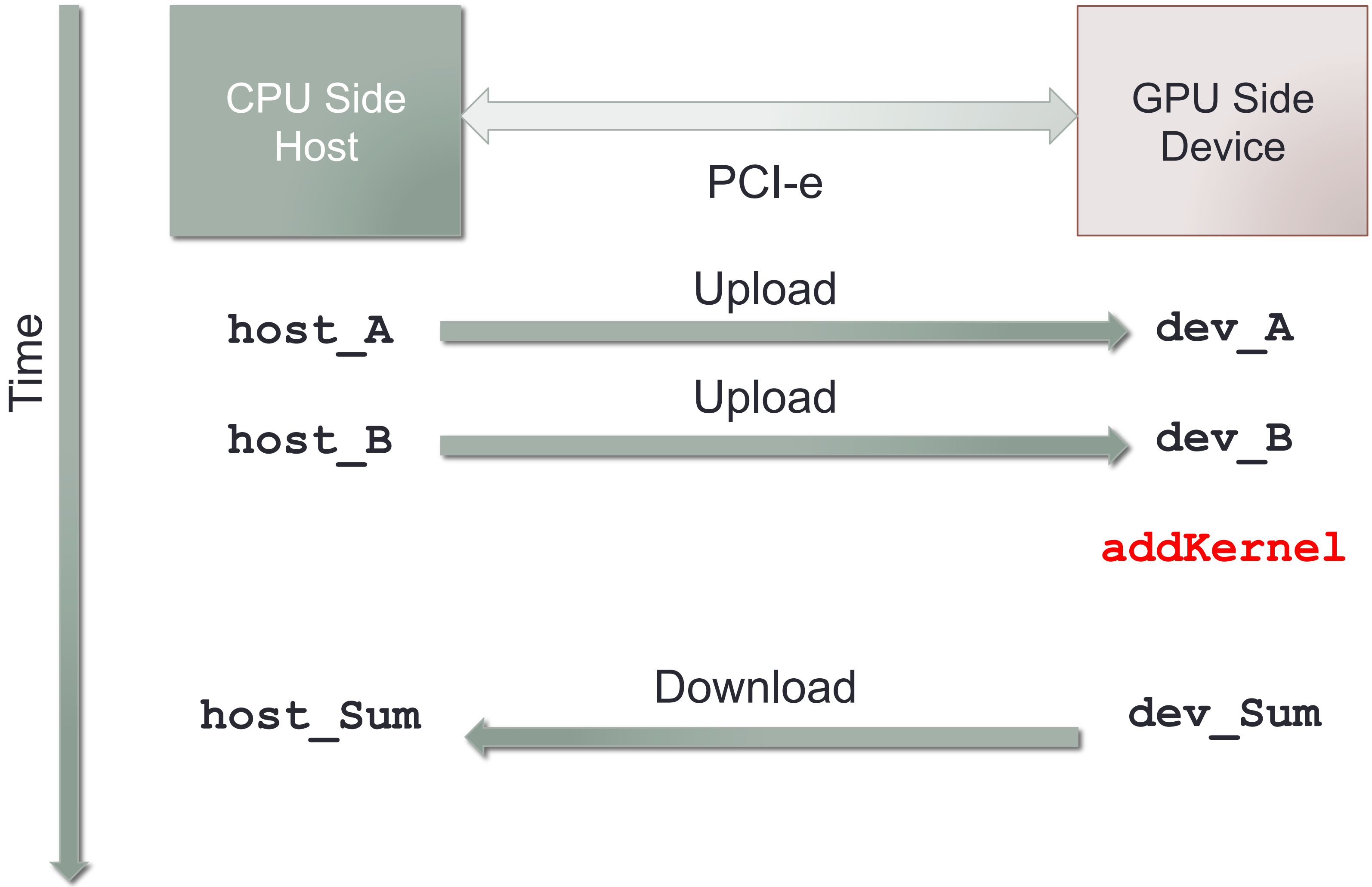
- Assuming a perfect parallel problem (Vector Addition)

```
Sum = A + B
```

- CPU Solution (`for` loop)

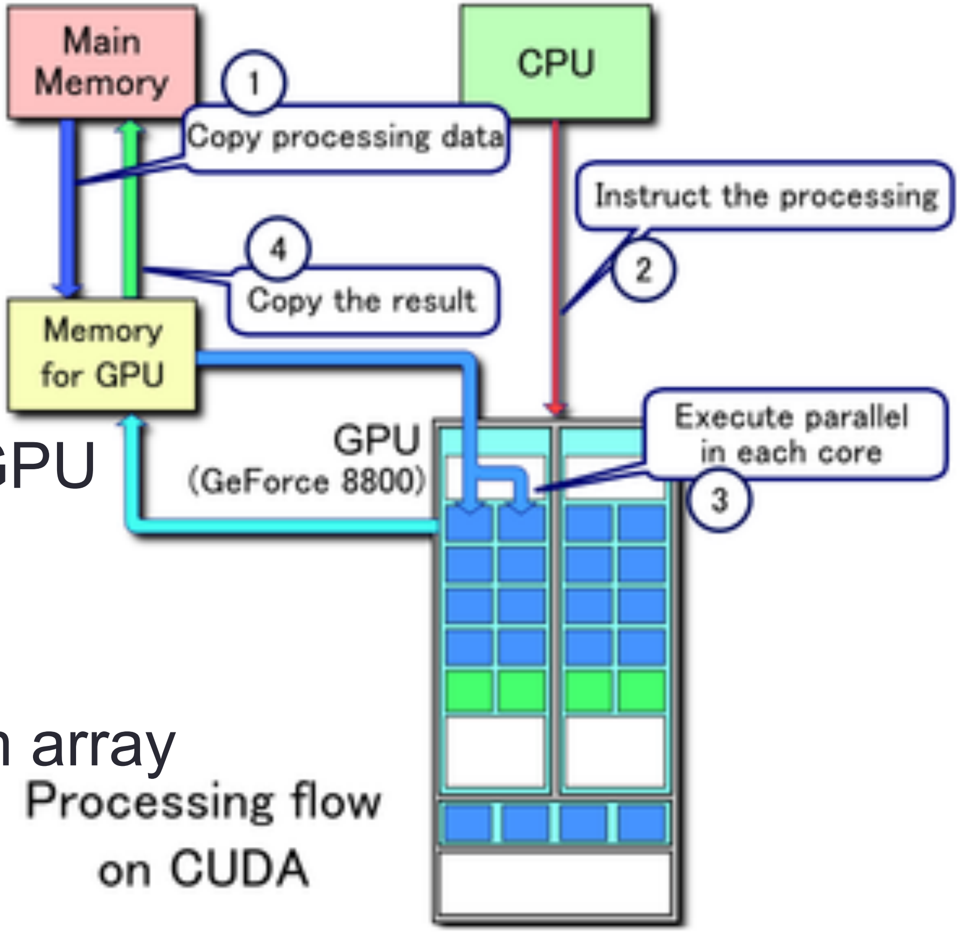
```
for (int i = 0; i< size(Sum); i++)
{
        Sum[i] = A[i] + B[i];
}
```

Programming Model



Sequence

1. Allocate CPU arrays
2. Pack them with Signals
3. Allocate GPU arrays
4. Send (Upload) arrays to GPU
5. Configure GPU
6. Execute addition kernel
7. Send (Download) the Sum array

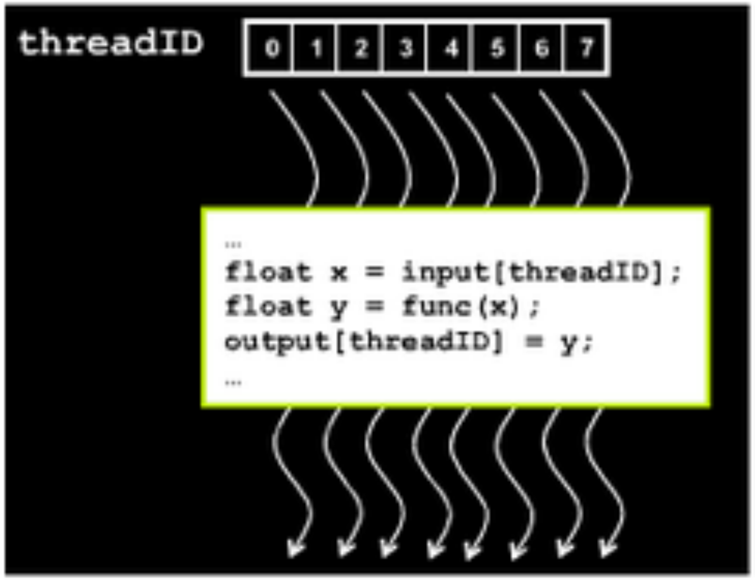


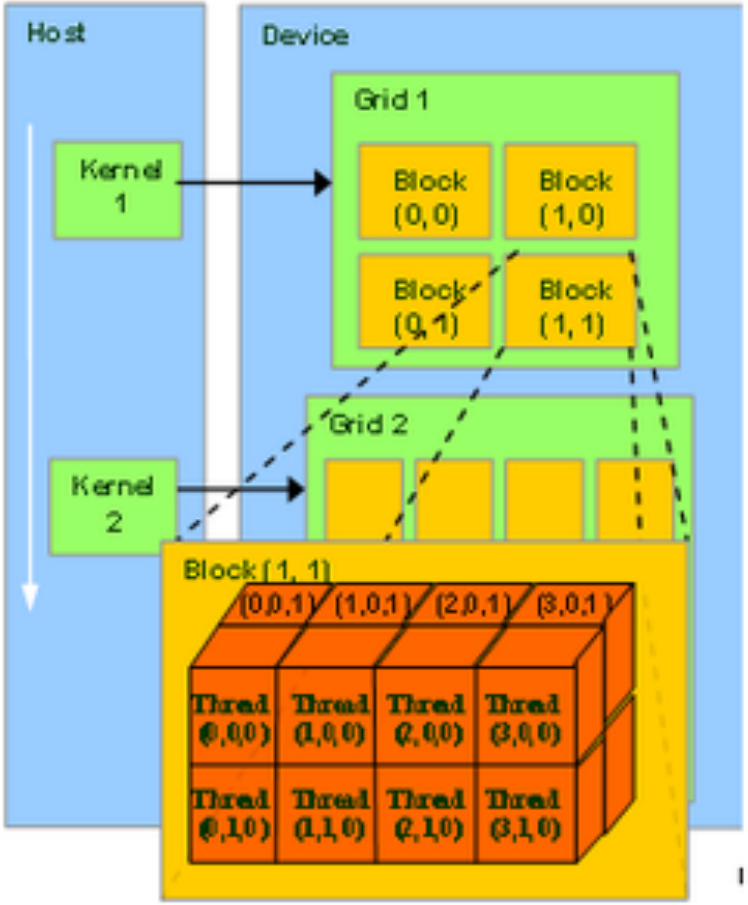
Processing flow on CUDA

Threading Hierarchy (Transparent)

- GPU Cores (Resources) are Configurable in
  - Grids
    - Blocks
      - Threads
- Kernel lunches a gird that contains several blocks where every block wraps a group of threads.
- This process is kernel-dependent and configurable for every **kernel**

Threading Hierarchy (Transparent)





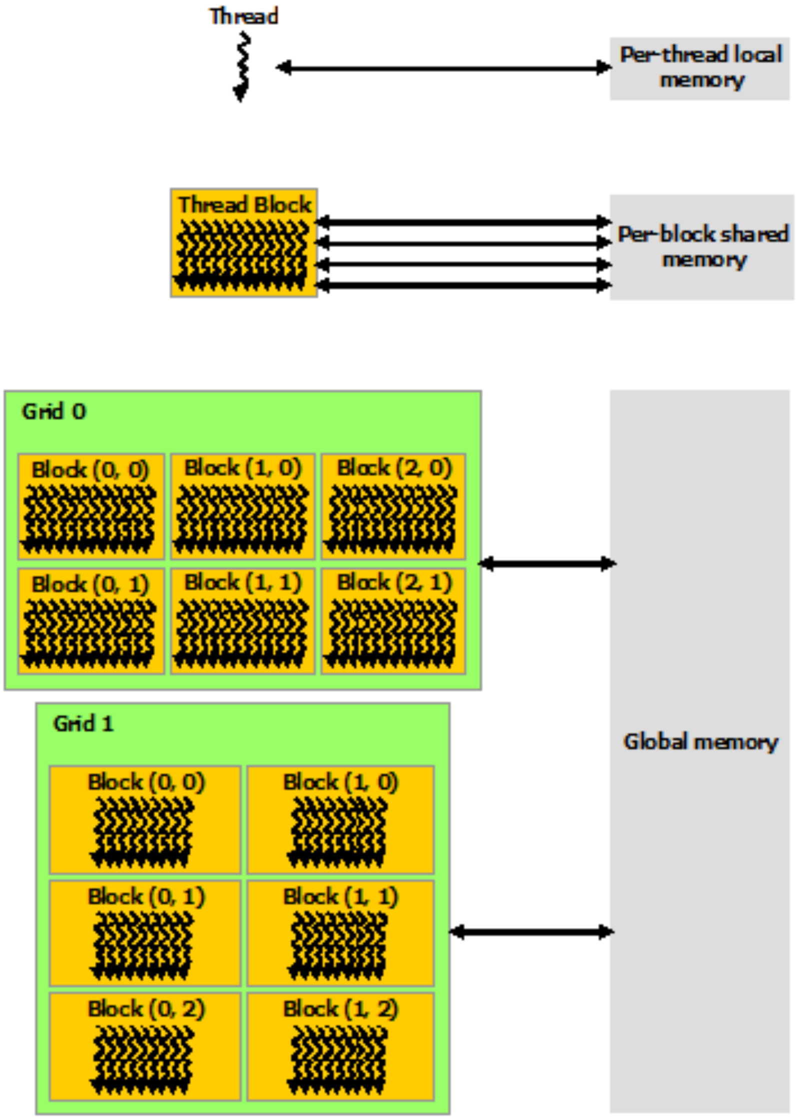
Memory Model

- Memory is divided into
  - Host Memory (CPU)
  - Device Memory (GPU)
    - Global Memory
    - Shared Memory
    - Texture Memory
    - Constant Memory
    - Local Memory
    - Register

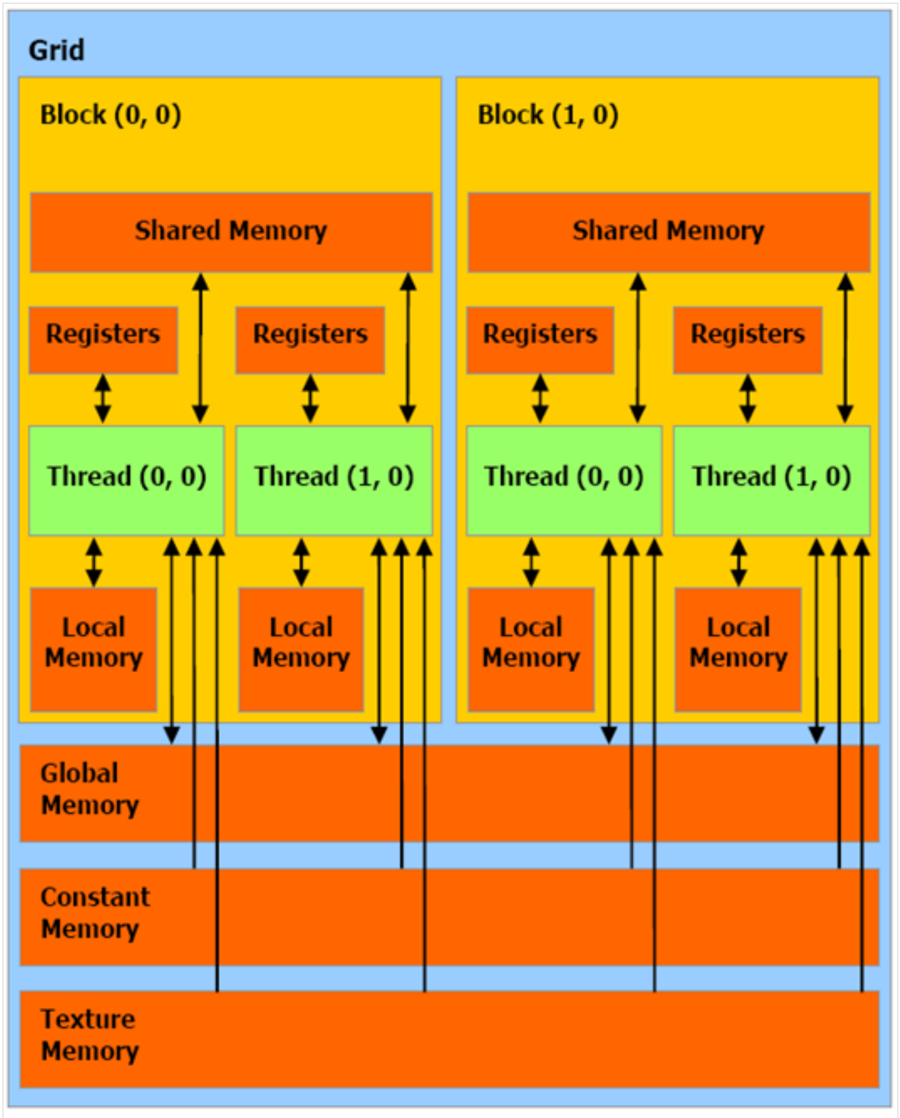
Latency Increasing

Size Decreasing

Memory Model



Memory Model



CUDA API Basics

- Function Qualifiers
  - __global__
    - Executed on the device
    - Callable from the host only
  - __device__
    - Executed on the device
    - Callable from the device only
  - __host__
    - Executed on the host
    - Callable from the host only

CUDA API Basics

- Execution Configuration or Grid Configuration
  - Must be specified for any call to a `__global__` function.
  - Defines the dimension of the grid and blocks.
    - grid_Dim = dim3()
    - block_Dim = dim3()
  - The function

```
__global__ addKernel(float A, float B, float Sum)
```

is callable from the host side via the invocation

```
addKernel  <<< grid_Dim, bloack_dim, NumElements>>>
                (float A, float B, float Sum);
```

Example (Vector Addition)

- Device Kernel Function **`addKernel`**

```
// Addition kernel that executes on the devices
__global__ void addKernel(float *A, float *B, float* Sum)
{
    // Calculate array index from the built-in variables
    int idx = blockIdx.x * blockDim.x + threadIdx.x;

    // Execute the addition operation
    Sum[idx] = A[idx] + B[idx];
}
```

Example (Vector Addition)

- Main Function – Memory Allocation

```
// Number of elements in arrays
const int N = 10000;

// Pointer to host & device arrays
float *host_A, *host_B, host_Sum;
float *dev_A, *dev_B, dev_Sum;

// Array size in bytes
size_t sizeBytes = N * sizeof(float);

// Host allocation
hotst_A = (float *) malloc(sizeBytes);
hotst_B = (float *) malloc(sizeBytes);
hotst_Sum = (float *) malloc(sizeBytes);

// Device allocation
cudaMalloc((void **) &dev_A, sizeBytes);
cudaMalloc((void **) &dev_B, sizeBytes);
cudaMalloc((void **) &dev_Sum, sizeBytes);
```

Example (Vector Addition)

- Main Function – Memory Upload

```
// Upload the arrays to the device
cudaMemcpy(dev_A, host_A, sizeBytes, cudaMemcpyHostToDevice);
cudaMemcpy(dev_B, host_B, sizeBytes, cudaMemcpyHostToDevice);
```

Example (Vector Addition)

- Main Function – Kernel Configuration & Execution

```
// Grid Configuration
int blockSize = 4;
int numBlocks = N/block_size + (N % block_size == 0 ? 0:1);

// Execute kernel on the device
addKernel <<< numBlocks, blockSize >>> (dev_A, dev_B, dev_Sum);
```

Example (Vector Addition)

- Main Function – Download Results and Printing …

```
// Download the result to the CPU
cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

// Print results
for (int i=0; i<N; i++)
    printf("%d %f\n", i, host_Sum[i]);
```

Example (Vector Addition)

- Main Function – Cleaning & Freeing Memory

```
// Cleanup and release memory on host side
free(host_A);
free(host_B);
free(host_Sum);

// Release device memory
cudaFree(dev_A);
cudaFree(dev_B);
cudaFree(dev_Sum);
```

GPU Accelerated Libraries

- cuFFT (10X)
  - MATLAB (fft, fft2, fft3), FFTW, FFTW++
- cuBLAS (6X ~ 17X)
  - Standard BLAS
- cuSPARSE (8X)
  - Collection of basic linear algebra subroutines for sparse matrices
- cuRAND
- AccelErEyes ArrayFire
  - Image processing and signal processing
- NPP (NVIDIA PEFROAMNCE PRIMITIVES)
- Thrust
- **cuYURI** (To be released)

Take Home Message

- If you have a code that contains a core `for` loop, then it is time to learn

How To Drive a GPU

Thanks for Paying Attention