CUDA

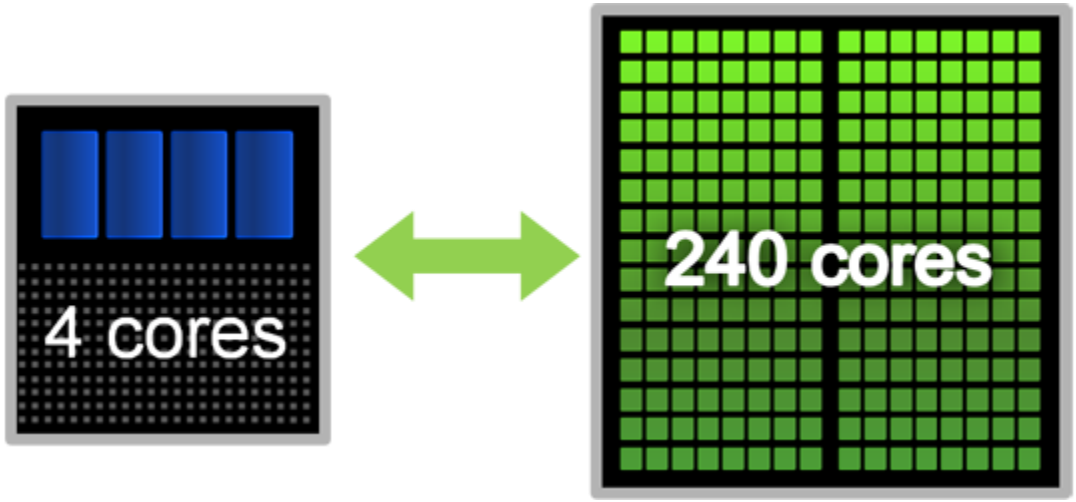
Prepared by
Mallegowda M

GPU vs CPU

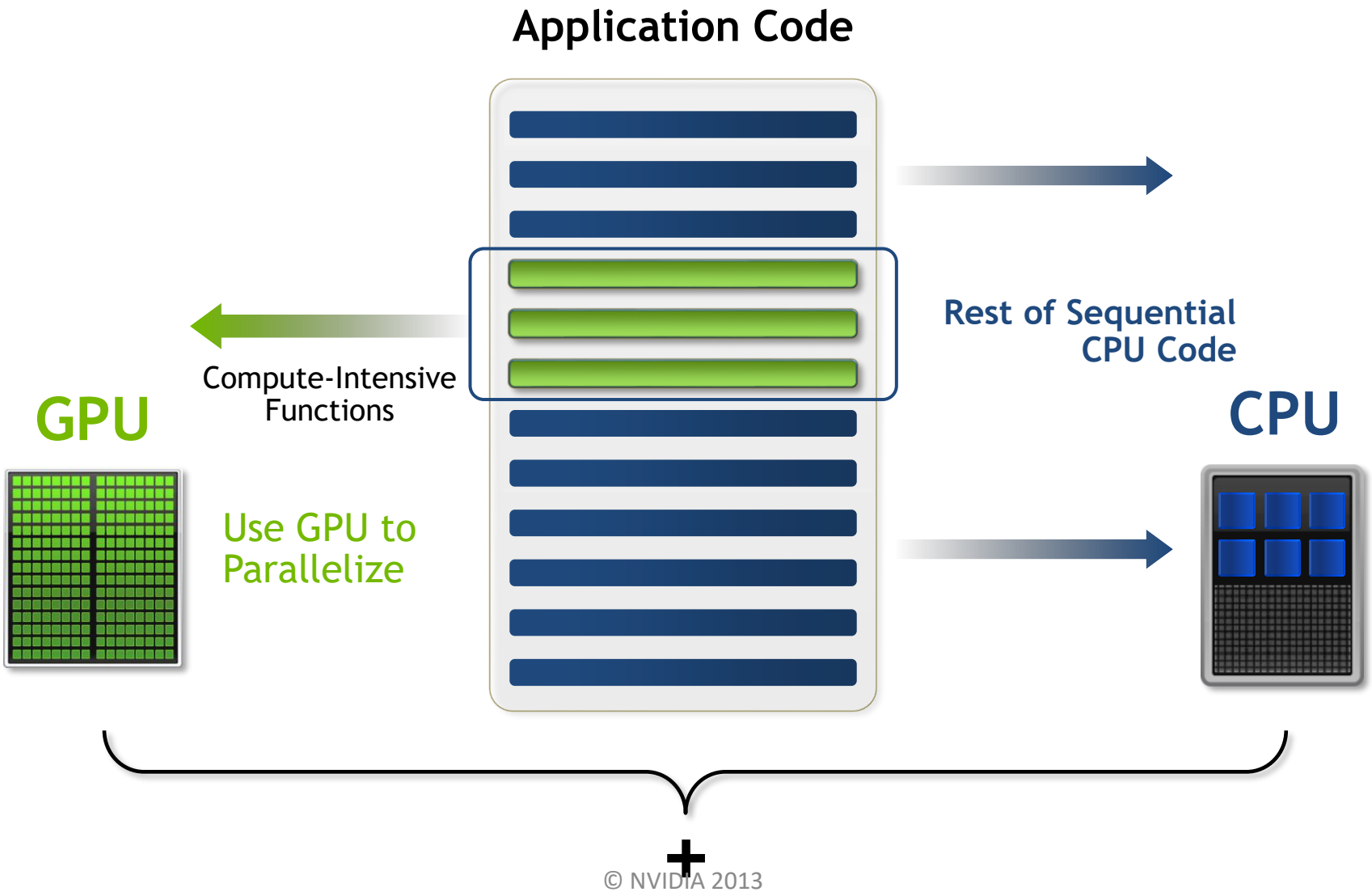
- A GPU is tailored for highly parallel operation while a CPU executes programs serially
- For this reason, GPUs have many parallel execution units and higher transistor counts, while CPUs have few execution units and higher clockspeeds
- A GPU is for the most part deterministic in its operation (though this is quickly changing)
- GPUs have much deeper pipelines (several thousand stages vs 10-20 for CPUs)
- GPUs have significantly faster and more advanced memory interfaces as they need to shift around a lot more data than CPUs

GPU and CPU

- Typically GPU and CPU coexist in a heterogeneous setting
- “Less” computationally intensive part runs on CPU (coarse-grained parallelism), and more intensive parts run on GPU (fine-grained parallelism)
- NVIDIA’s GPU architecture is called CUDA (Compute Unified Device Architecture) architecture, accompanied by CUDA programming model, and CUDA C language



Small Changes, Big Speed-up



CUDA

- "Compute Unified Device Architecture"
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute – graphics-free API
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

Parallel Computing on a GPU

- 8-series GPUs deliver 25 to 200+ GFLOPS on compiled parallel C applications
  - Available in laptops, desktops, and clusters

GeForce 8800

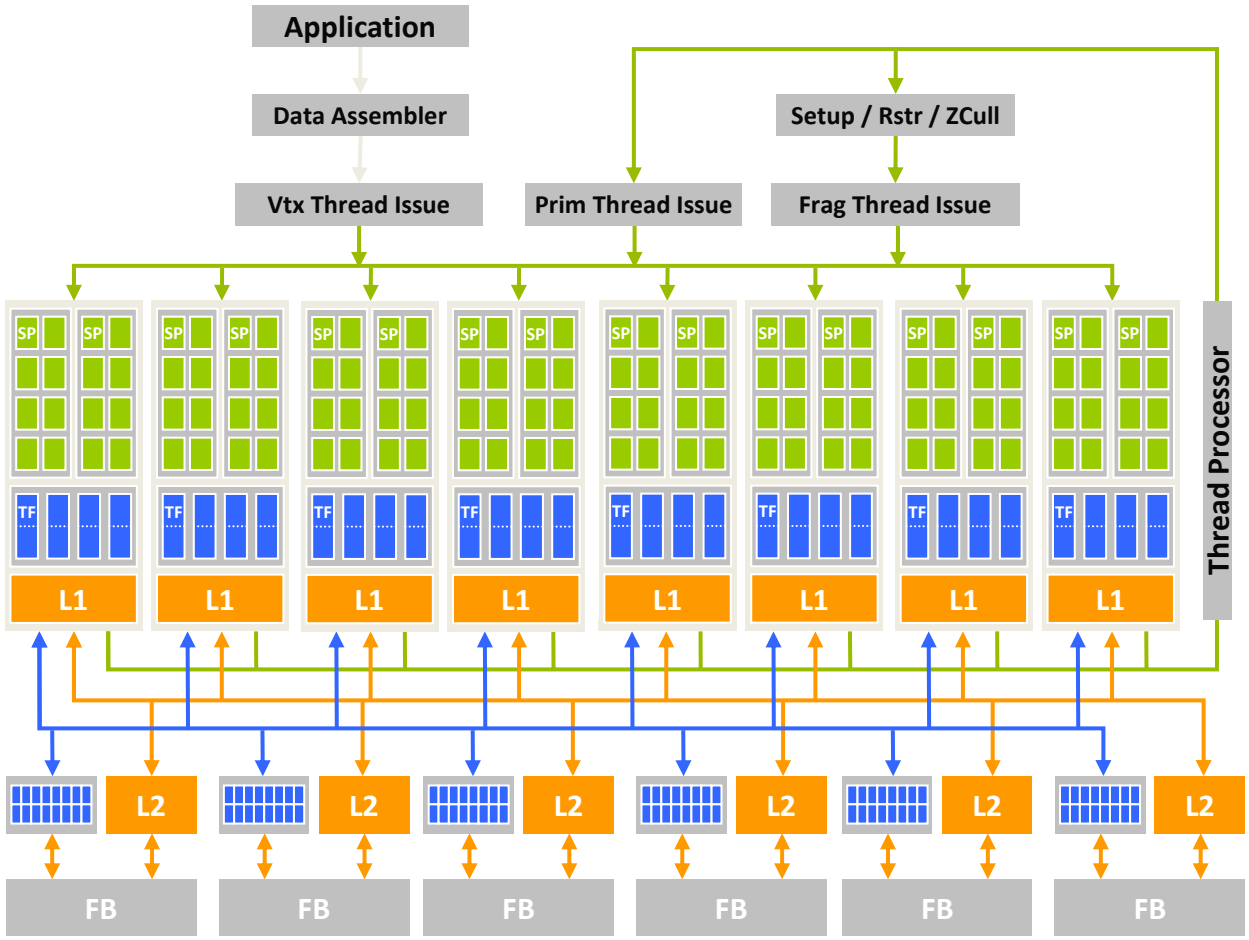
- GPU parallelism is doubling every year
- Programming model scales transparently

Tesla D870

- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism

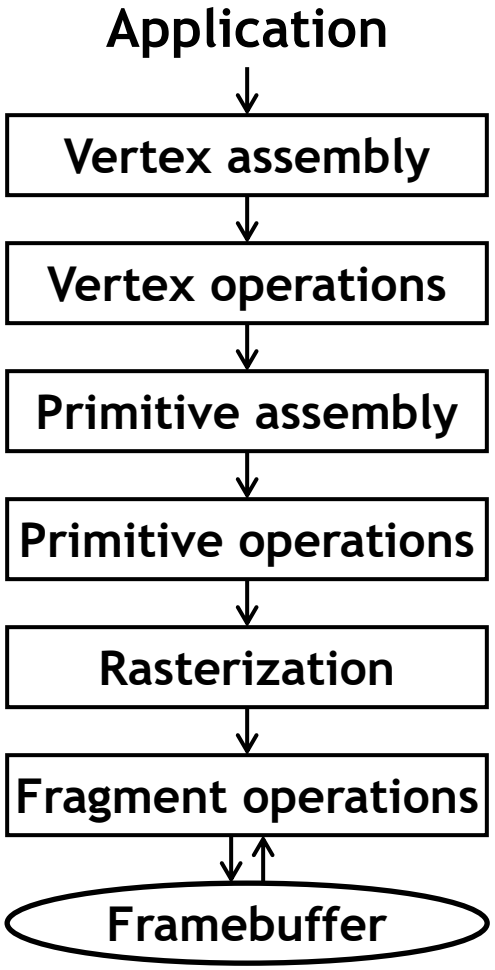
Tesla S870

Implementation = abstraction (from lecture 2)



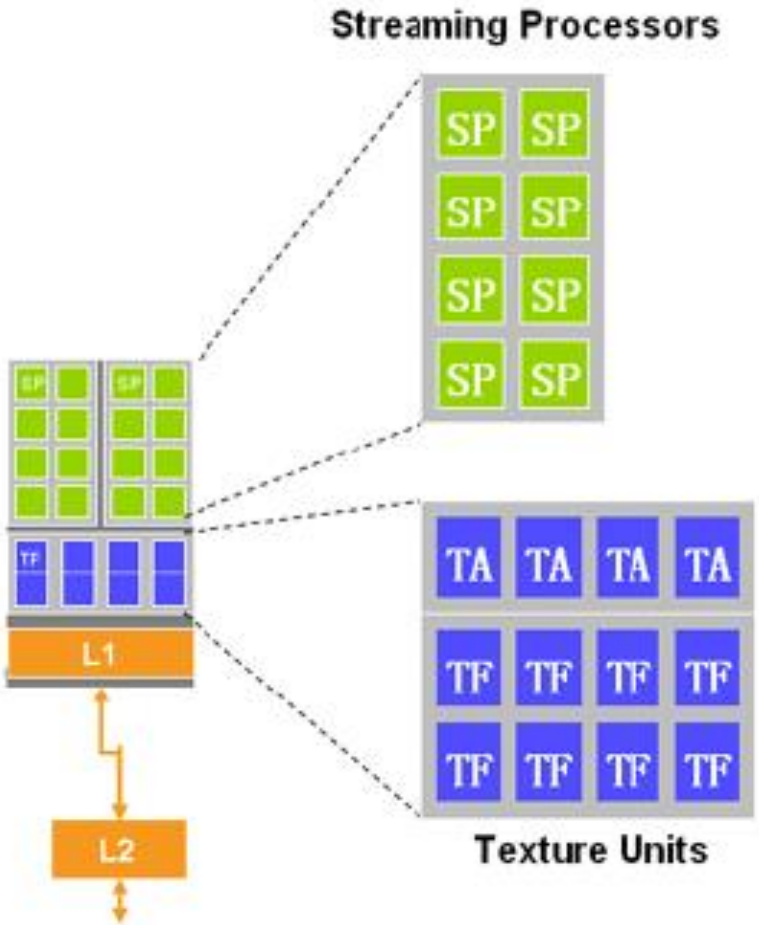
NVIDIA GeForce 8800

Source : NVIDIA



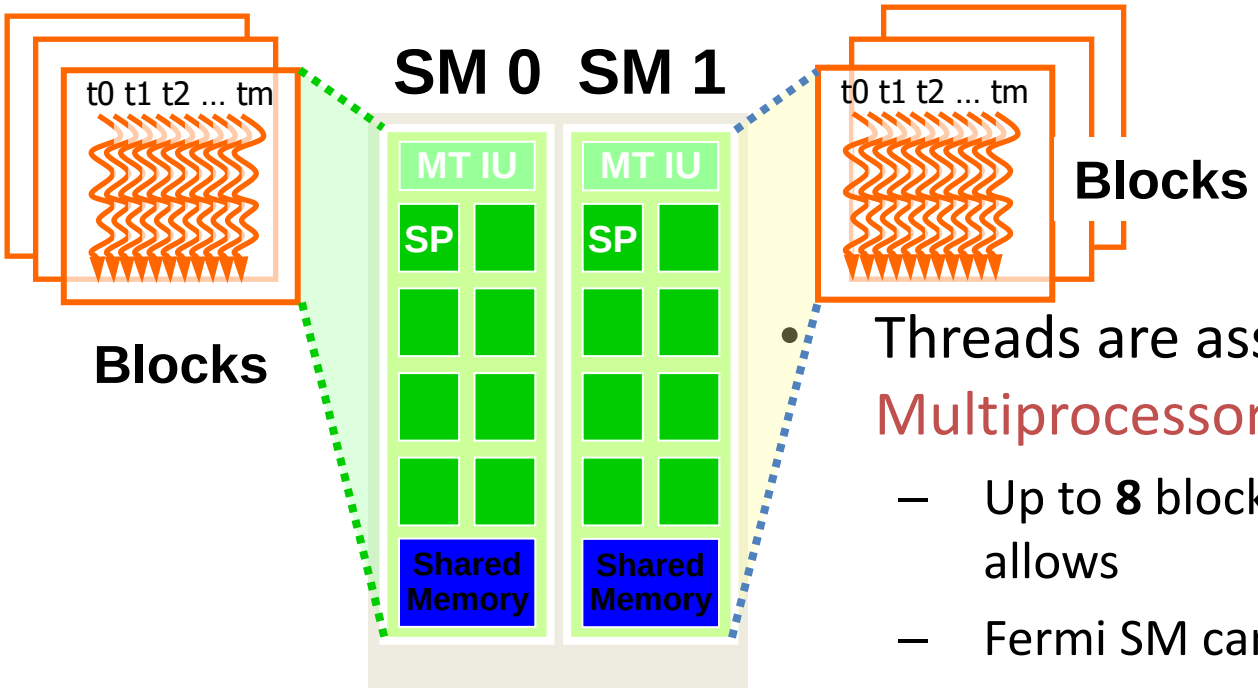
OpenGL Pipeline

Streaming Processors, Texture Units, and On-chip Caches



- SP = Streaming Processors
- TF = Texture Filtering Unit
- TA = Texture Address Unit
- L1/L2 = Caches

Review: Executing Thread Blocks

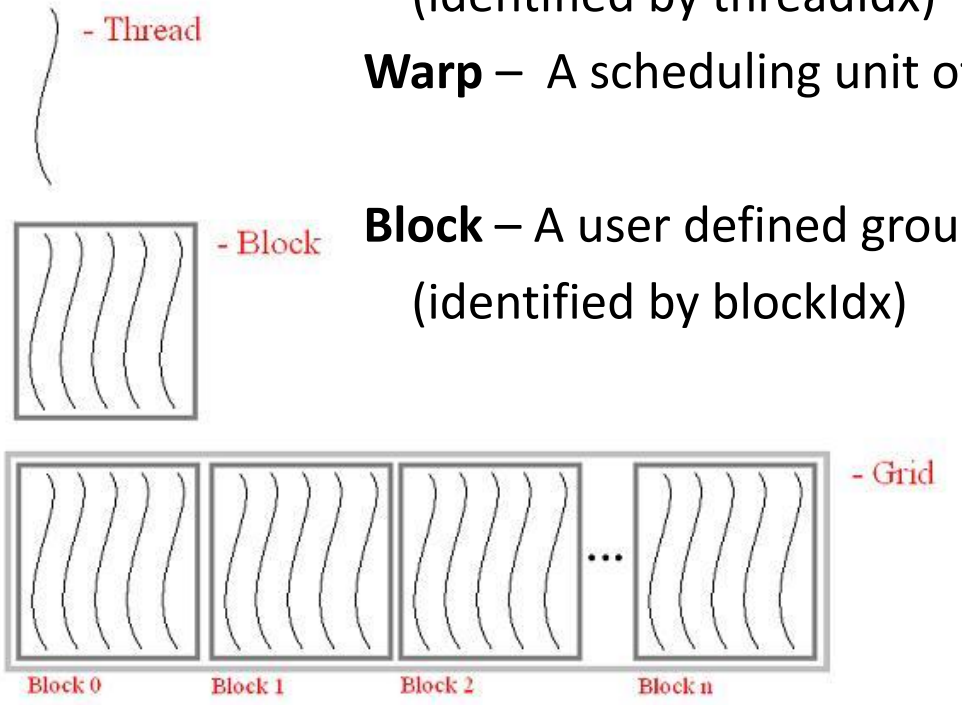


- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to **8** blocks to each SM as resource allows
  - Fermi SM can take up to **1536** threads
    - Could be 256 (threads/block) * 6 blocks
    - Or 512 (threads/block) * 3 blocks, etc.
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread execution

Thread Hierarchy

Thread – Distributed by the CUDA runtime
(identified by threadIdx)

Warp – A scheduling unit of up to 32 threads



Block – A user defined group of 1 to 512 threads.
(identified by blockIdx)

Grid – A group of one or more blocks. A grid is created for each CUDA kernel function

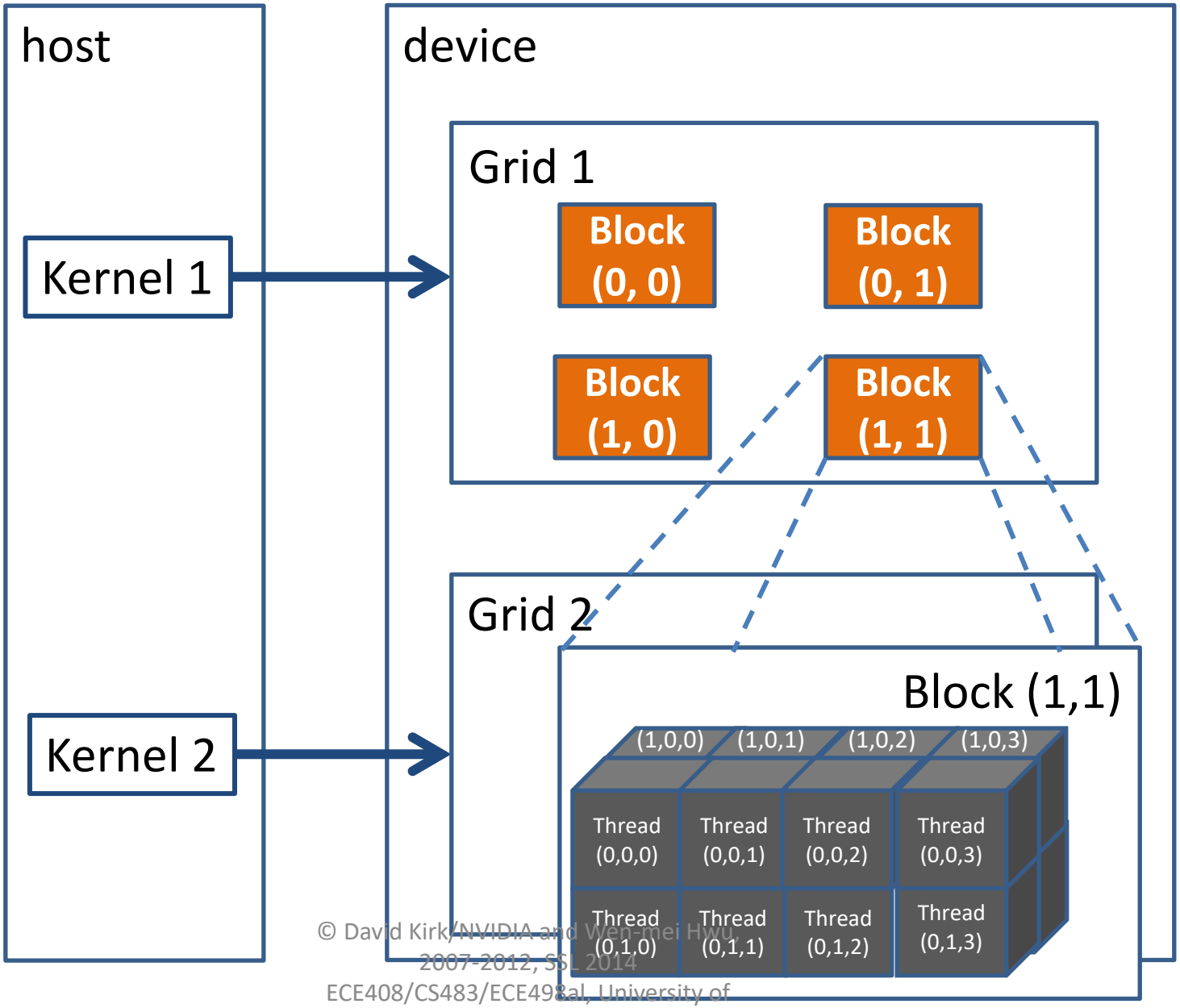
Programming Model

Historically, GPUs designed for creating image data for displays.

That application involves manipulating image pixels (picture elements) and often the same operation each pixel

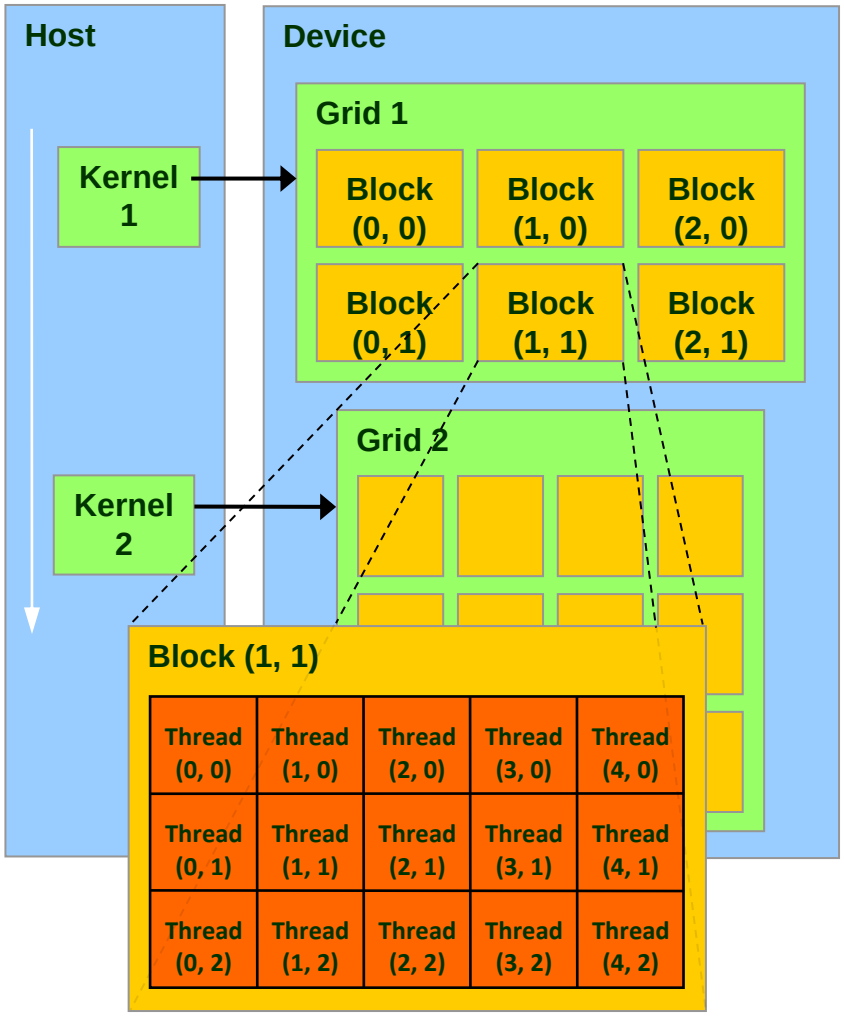
SIMD (single instruction multiple data) model - An efficient mode of operation in which the same operation is done on each data element at the same time

A Multi-Dimensional Grid Example



Thread Batching: Grids and Blocks

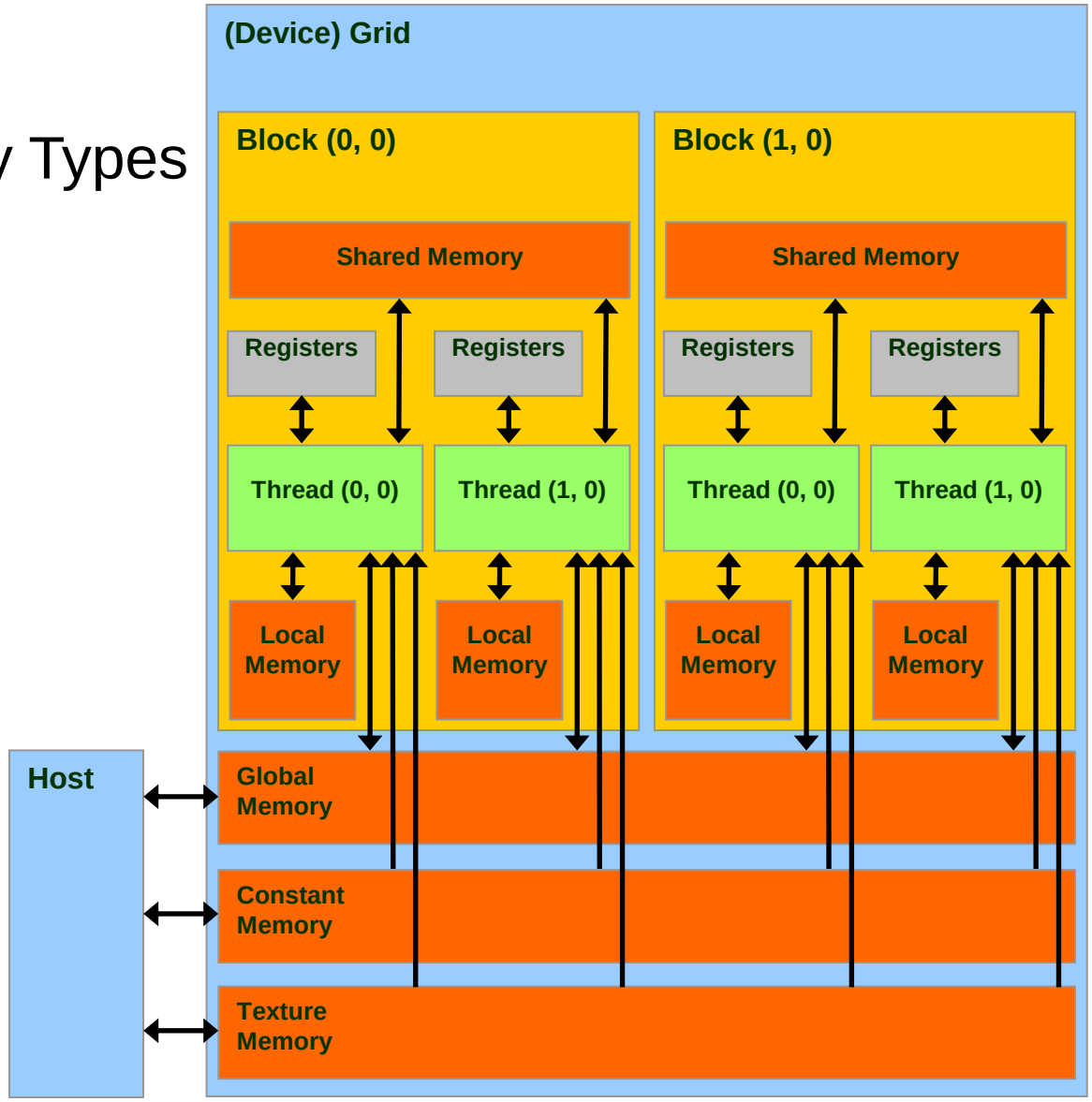
- Kernel executed as a grid of thread blocks
  - All threads share data memory space
- Thread block is a batch of threads, can cooperate with each other by:
  - Synchronizing their execution: For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate
  - (Unless thru slow global memory)
- Threads and blocks have IDs



Courtesy: NDVIA

Memory Model

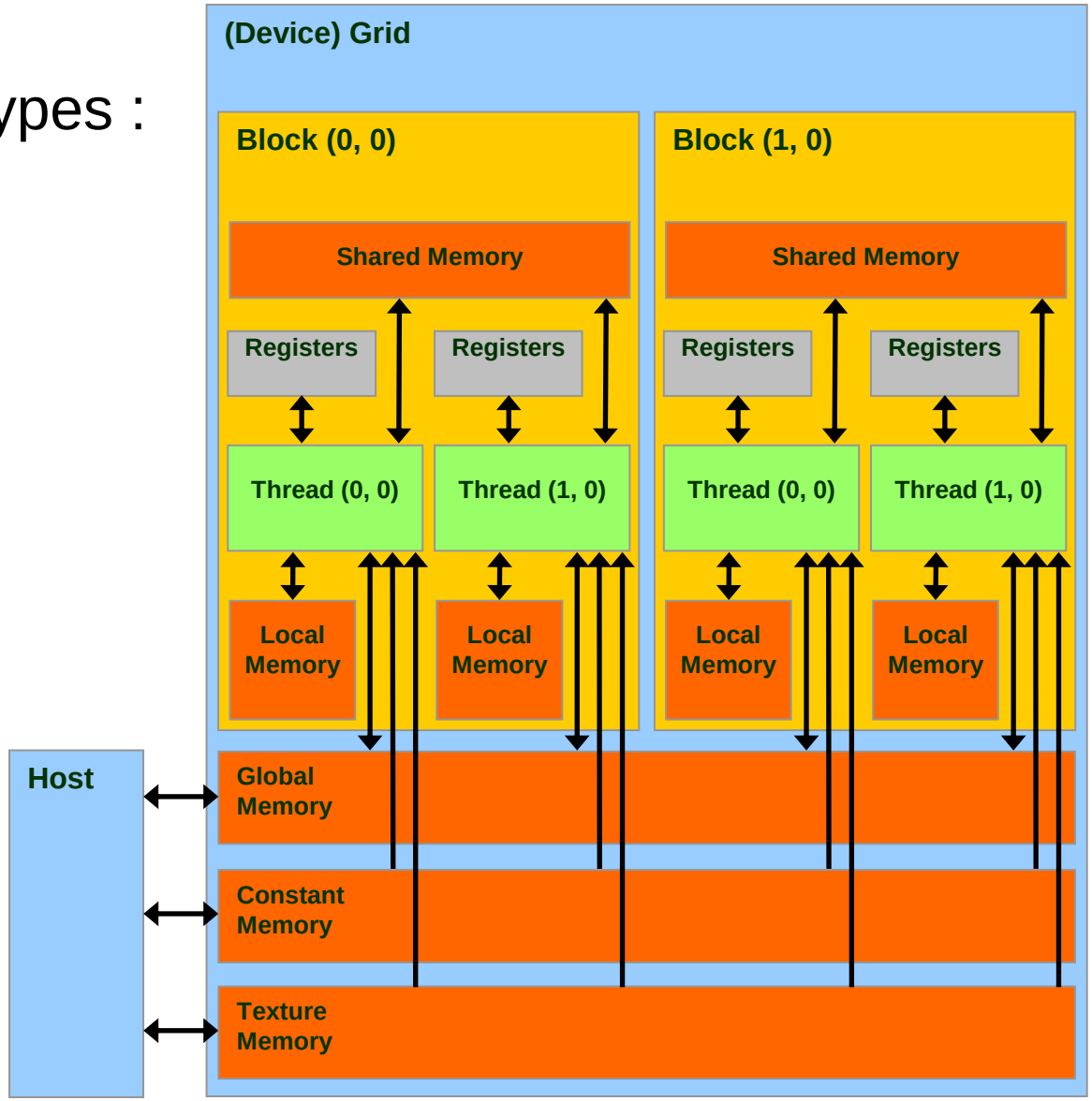
There are 6 Memory Types



Memory Model

There are 6 Memory Types :

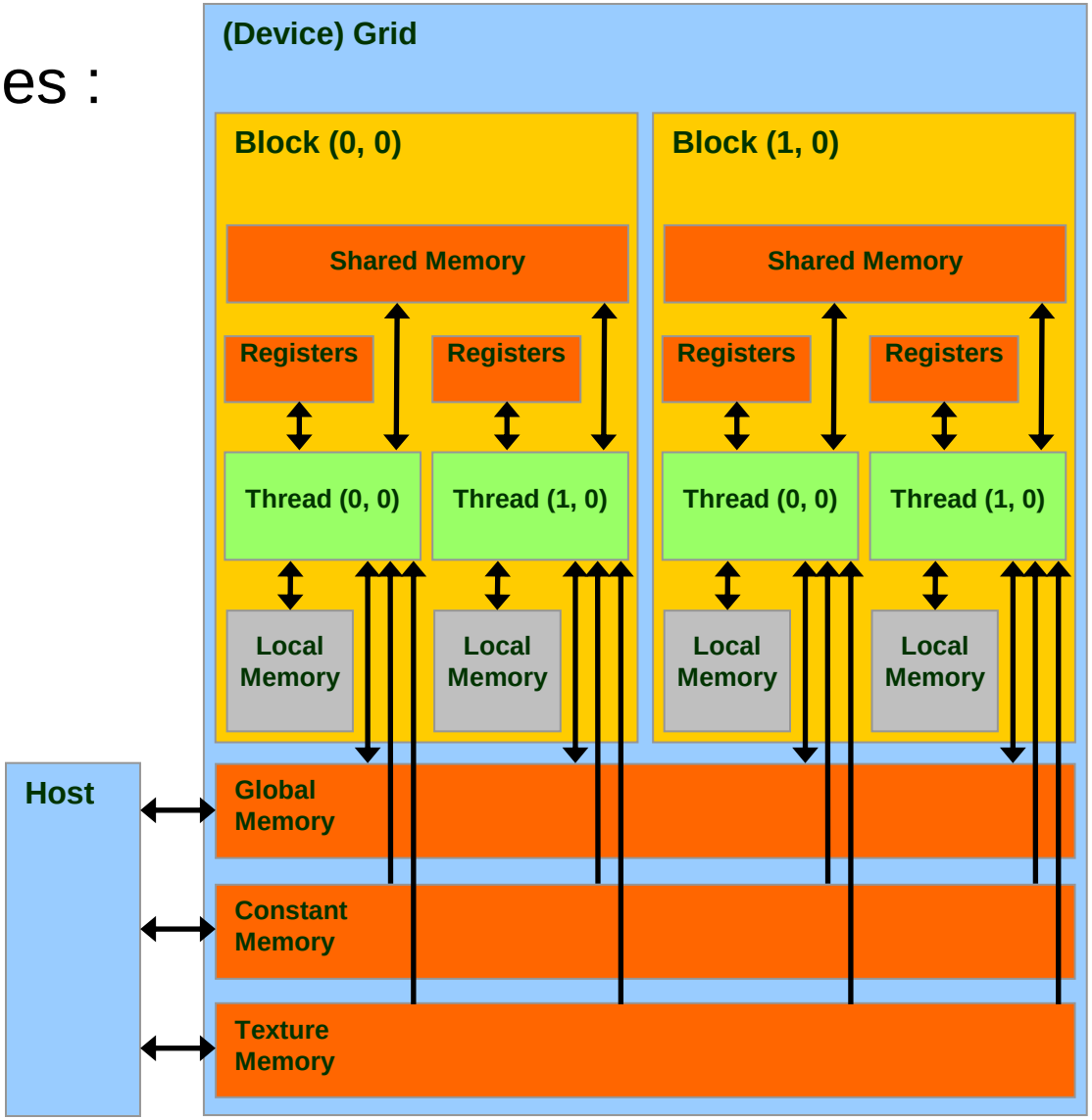
- **Registers**
  - on chip
  - fast access
  - per thread
  - limited amount
  - 32 bit



Memory Model

There are 6 Memory Types :

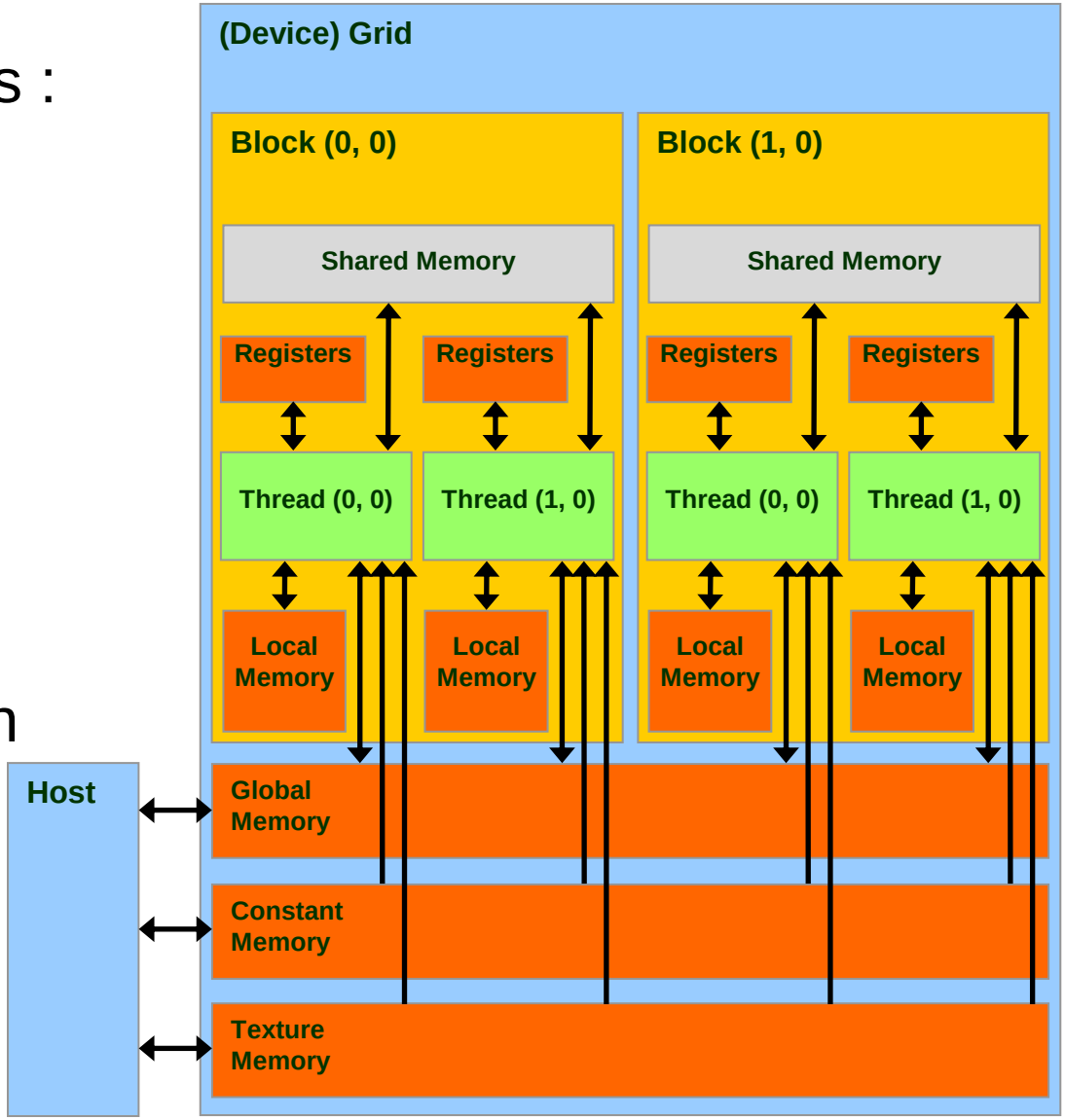
- Registers
- **Local Memory**
  - in DRAM
  - slow
  - non-cached
  - per thread
  - relative large



Memory Model

There are 6 Memory Types :

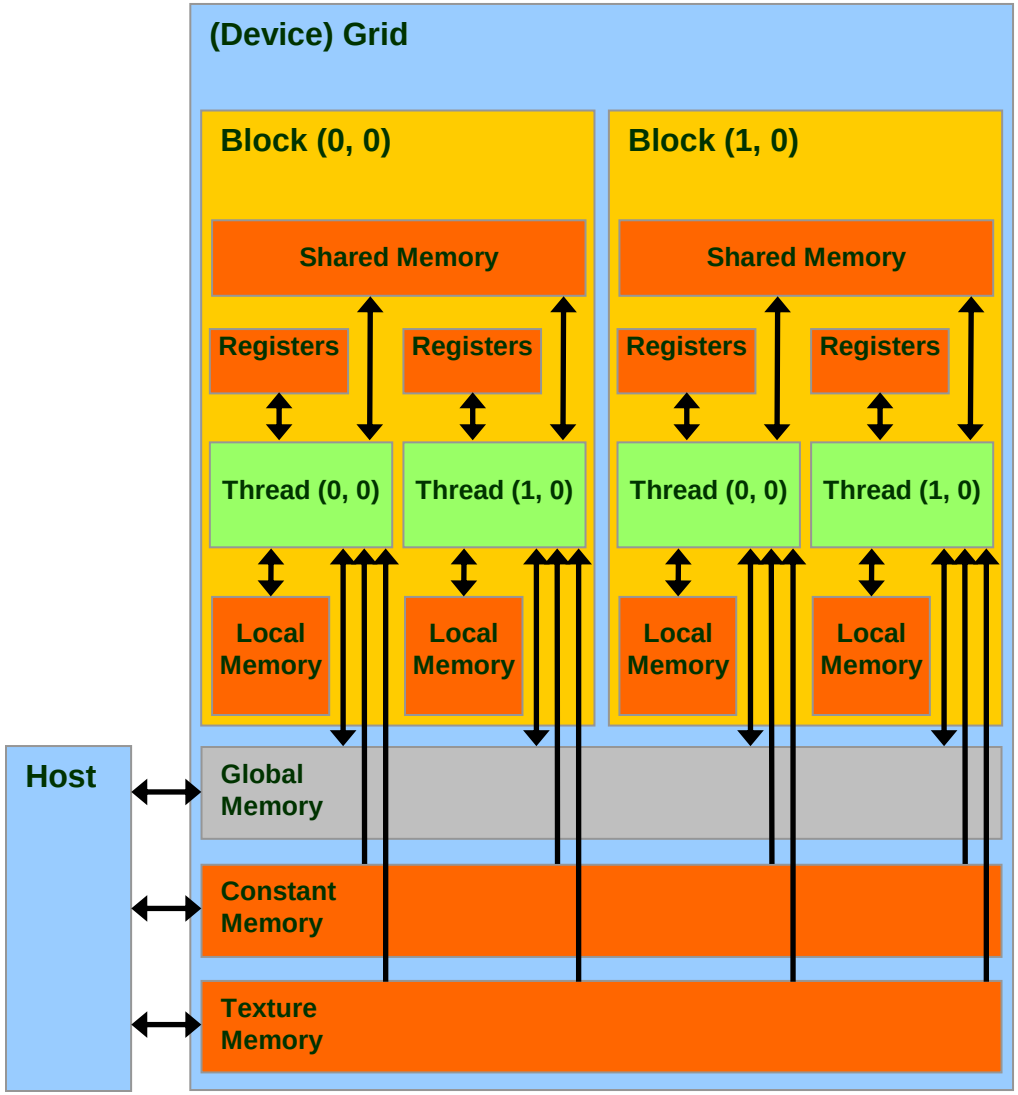
- Registers
- Local Memory
- **Shared Memory**
  - on chip
  - fast access
  - per block
  - 16 KByte
  - synchronize between threads



Memory Model

There are 6 Memory Types :

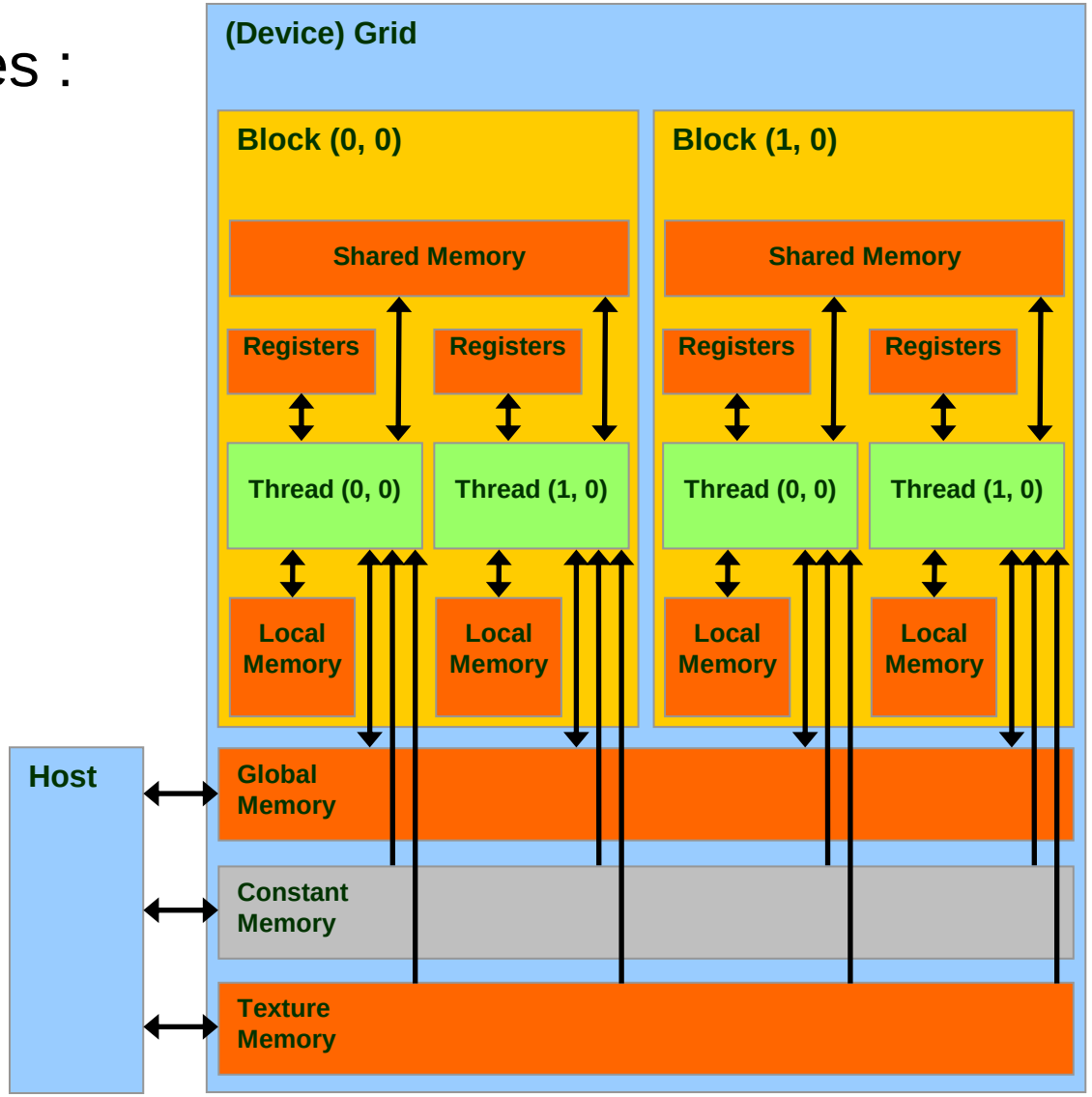
- Registers
- Local Memory
- Shared Memory
- **Global Memory**
  - in DRAM
  - slow
  - non-cached
  - per grid
  - communicate between grids



Memory Model

There are 6 Memory Types :

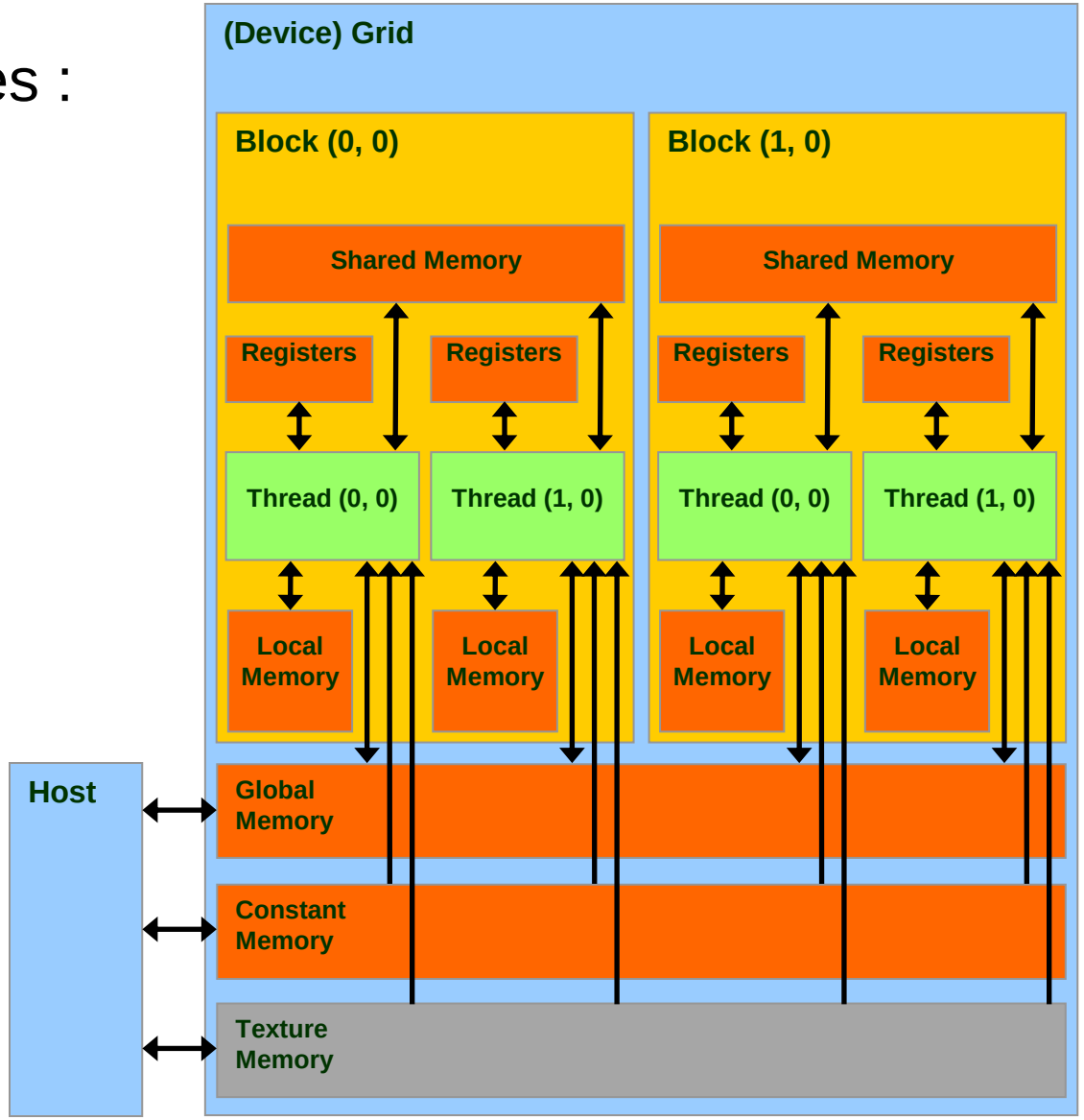
- Registers
- Local Memory
- Shared Memory
- Global Memory
- **Constant Memory**
  - in DRAM
  - cached
  - per grid
  - read-only



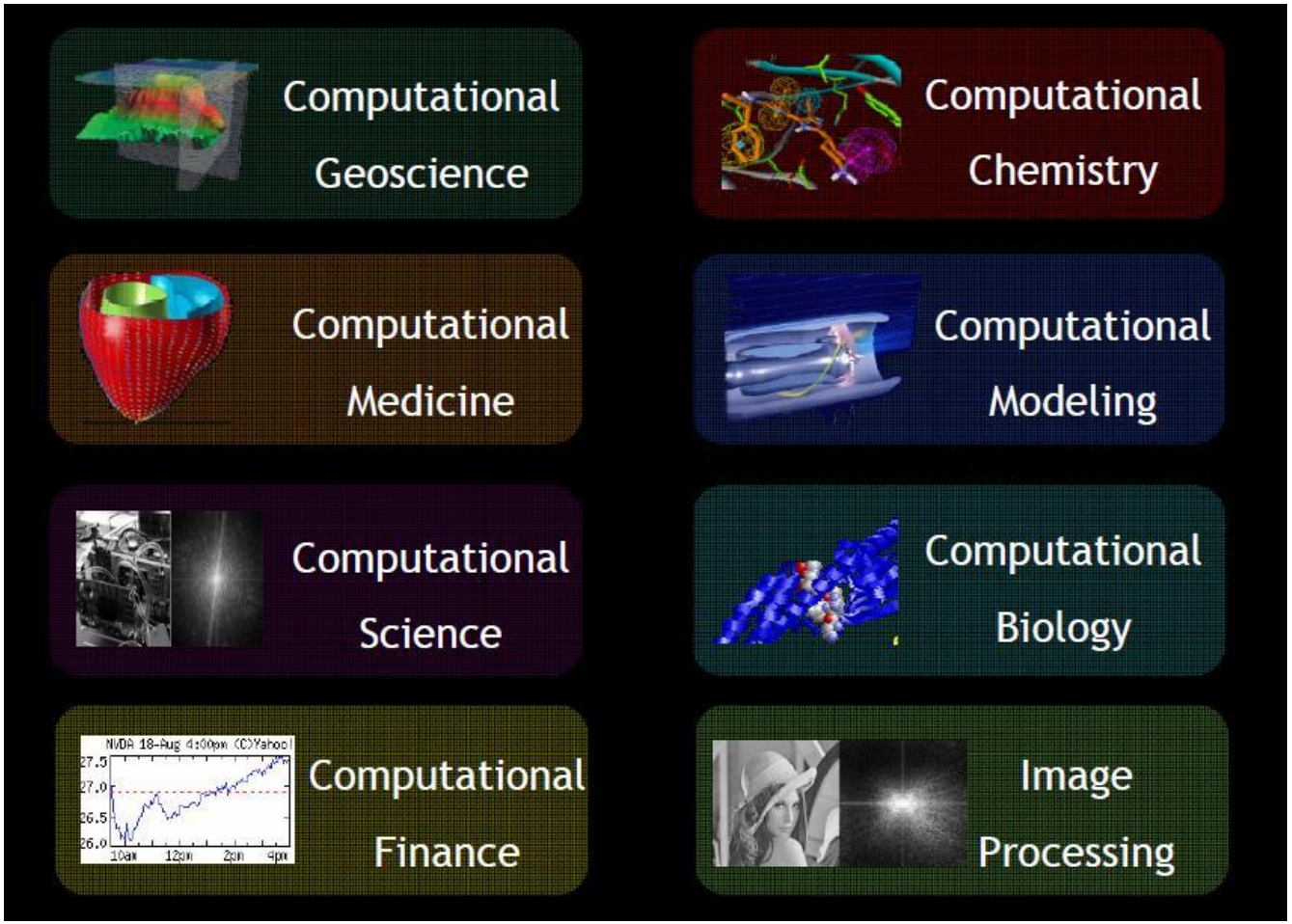
Memory Model

There are 6 Memory Types :

- Registers
- Local Memory
- Shared Memory
- Global Memory
- Constant Memory
- **Texture Memory**
  - in DRAM
  - cached
  - per grid
  - read-only



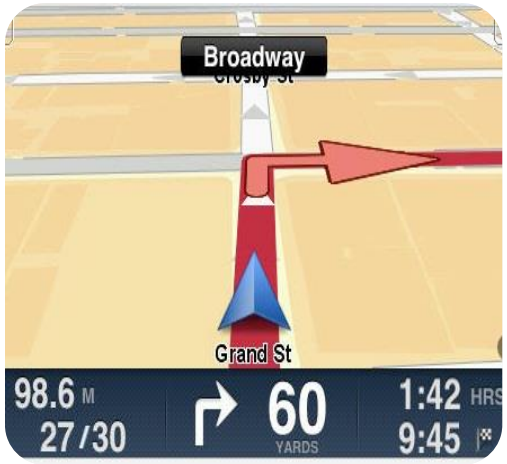
Different Types of CUDA Applications



Directives: Easy & Powerful

Real-Time Object Detection

Global Manufacturer of Navigation Systems

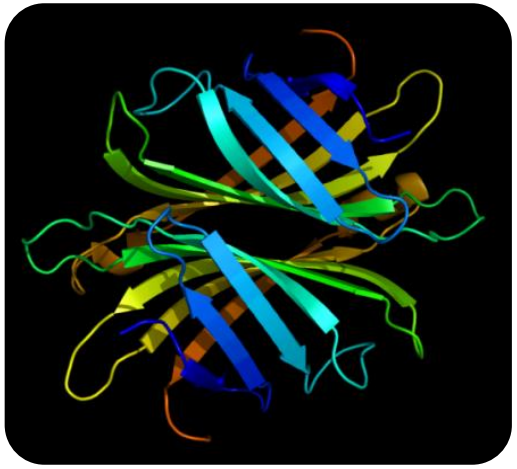


Valuation of Stock Portfolios using Monte Carlo

Global Technology Consulting Company

Interaction of Solvents and Biomolecules

University of Texas at San Antonio

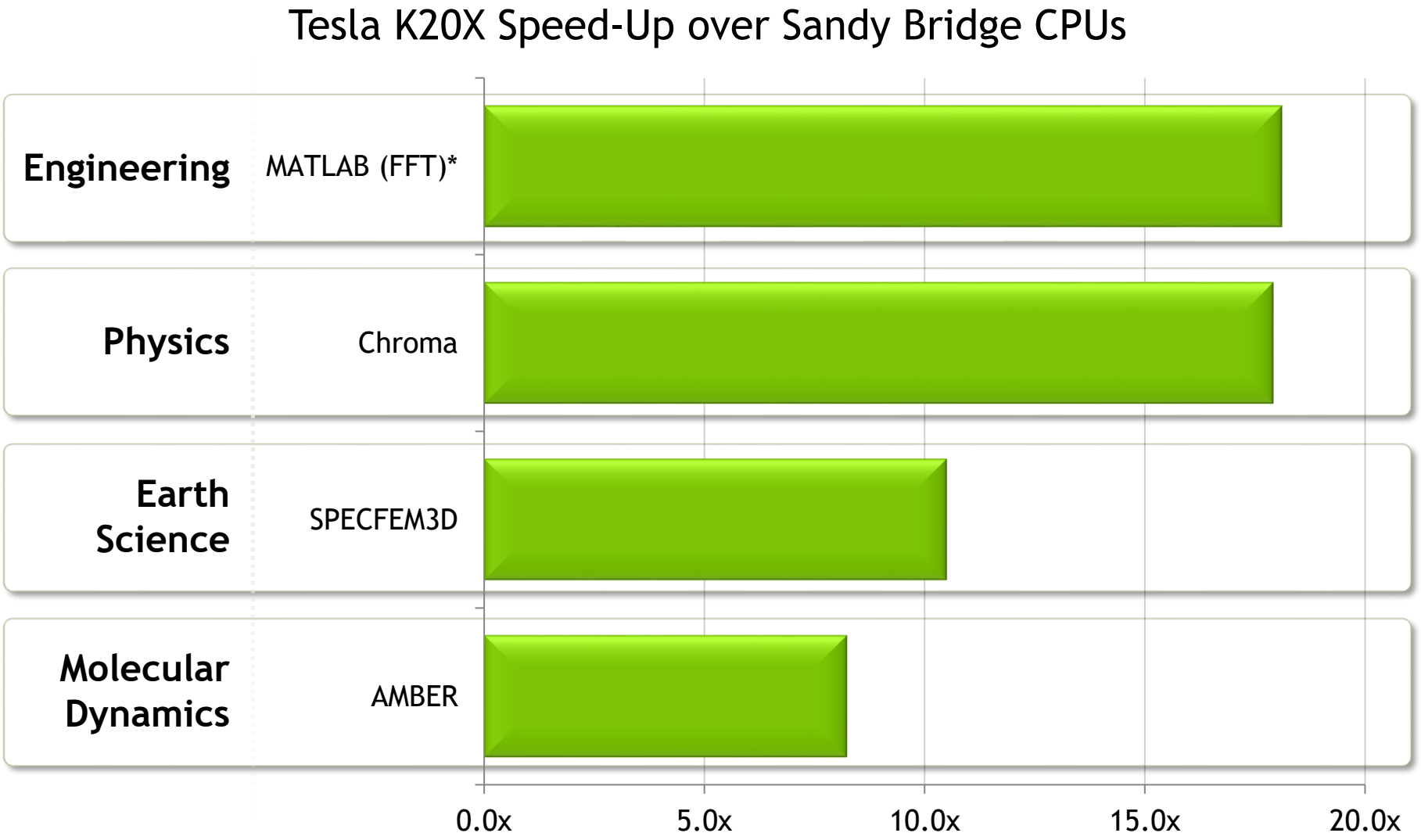


5x in 40 Hours **2x in 4 Hours** **5x in 8 Hours**

"Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications."

-- Developer at the Global Manufacturer of Navigation Systems

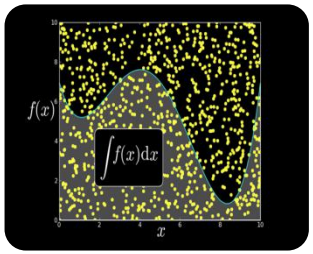
Fastest Performance on Scientific Applications



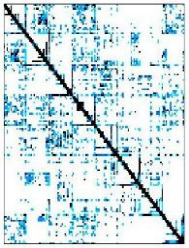
CPU results: Dual socket E5-2687w, 3.10 GHz, GPU results: Dual socket E5-2687w + 2 Tesla K20X GPUs
*MATLAB results comparing one i7-2600K CPU vs with Tesla K20 GPU
Disclaimer: Non-NVIDIA implementations may not have been fully optimized

Some GPU-accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

CULA tools

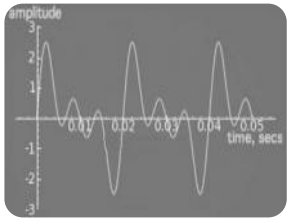
GPU Accelerated Linear Algebra

MAGMA

ICL UT

Matrix Algebra on GPU and Multicore

open source initiative



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

CUSP

Sparse Linear Algebra

open source initiative

C++ STL Features for CUDA

open source initiative

CUDA Programming

Basic CUDA program structure

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

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

        3. Copy data to GPU

        4. Call "kernel" routine to execute on GPU
        (with CUDA syntax that defines no of threads and their physical structure)

        5. Transfer results from GPU to CPU

        6. Free memory space in device (GPU)
        7. Free memory space in host (CPU)

        return;
}
```

1. Allocating 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 );
```


Derived from Jason Sanders, "Introduction to CUDA C" GPU technology conference, Sept. 20, 2010.

2. Allocating memory space in “host” (CPU) for data

Use regular C malloc routines:

```
int *a, *b, *c;
…
a = (int*)malloc(size);
b = (int*)malloc(size);
c = (int*)malloc(size);
```

or statically declare variables:

```
#define N 256
…
int  a[N], b[N], c[N];
```

3. Transferring 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

a and **b** are pointers to host data

4. Declaring “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**.

Declaring a Kernel Routine

A kernel defined using CUDA specifier **__global__**

Two underscores each side

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

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, N threads in block
  …
}
```

CUDA structure that provides thread ID in block

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

```
Thread 0:    devC[0] = devA[0] + devB[0];
Thread 1:    devC[1] = devA[1] + devB[1];

Thread N-1: devC[N-1] = devA[N-1]+devB[N-1];
```


Loosely derived from CUDA C programming guide, v 3.2 , 2010, NVIDIA

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

Use CUDA routine cudaMemcpy

Destination Source

cudaMemcpy(c, devC, size, cudaMemcpyDeviceToHost);

where:

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

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

Use CUDA cudaFree routine:

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

7. Free memory space in (CPU) host

(if CPU memory allocated with malloc)

Use regular C free routine to deallocate memory if previously allocated with malloc:

```
free( a );
free( b );
free( c );
```

Complete CUDA program

Adding two vectors, A and B

N elements in A and B, and

N threads

(without code to load arrays with data)

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

    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);
```

Other examples

Addition on the Device: `add()`

- Returning to our `add()` kernel

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

- Let's take a look at main()…

Addition on the Device: `main()`

```
int main(void) {
        int a, b, c;                // host copies of a, b, c
        int *d_a, *d_b, *d_c;       // device copies 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
        a = 2;
        b = 7;
```

Addition on the Device: `main()`

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

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

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

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