GPU Programming
I – Fundamentals

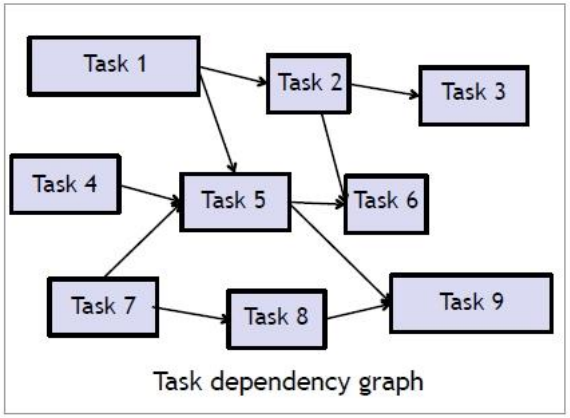
Stefan BORNHOFEN

Parallelism

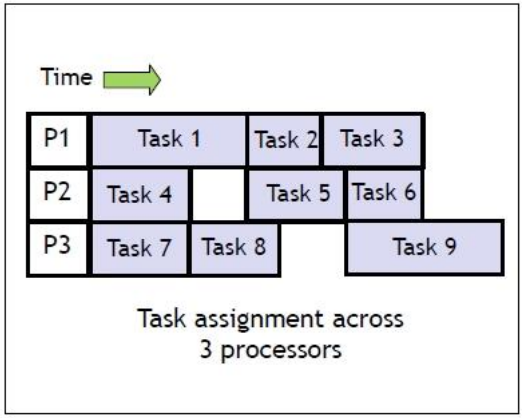
The MHz race has run out of steam.

Task parallelism

- Parallelize different tasks that do not depend on other uncompleted tasks. The tasks being parallelized can be completely different (same or different data)



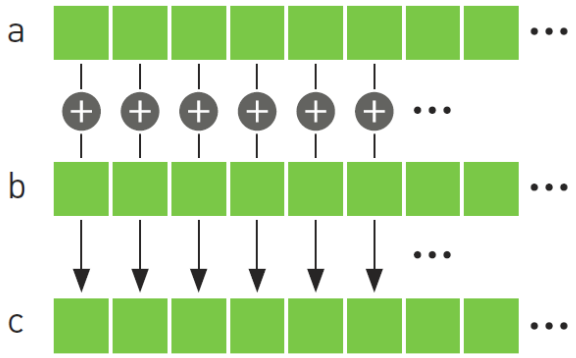
Task dependency graph



Task assignment across 3 processors

Data parallelism

- Perform the same task on multiple pieces of data



Summing two vectors

Parallel Implementations

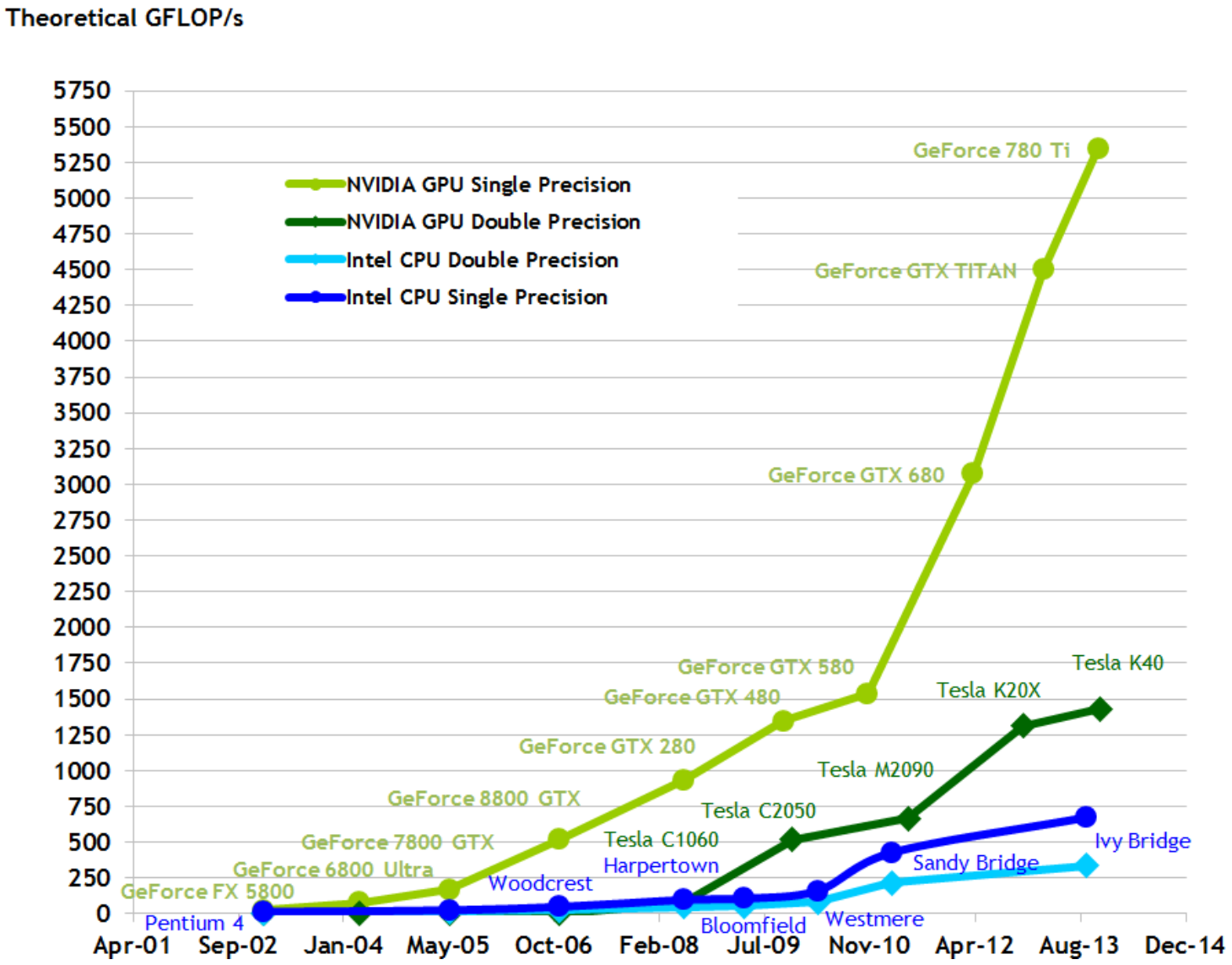
- Multi-threading, multi-process programming
- Shared memory (OpenMP)
- Distributed memory (MPI)
- Grid, Cluster
- GPU programming (CUDA, OpenCL)

Hybrid approaches are common.

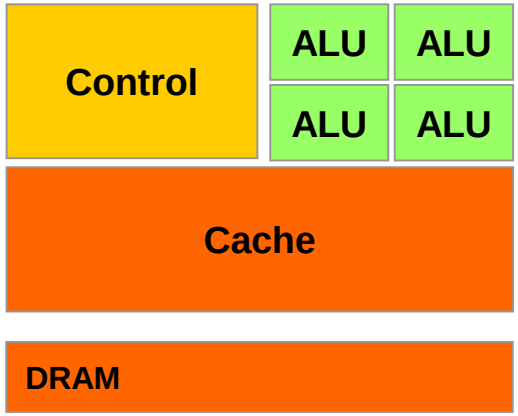
Enter the GPU

- Graphics Processing Unit
- Massively multithreaded chip optimized for 3D graphics
- Development initially driven by the game industry
- Two major vendors: NVIDIA and AMD
- Serves as both
  - a programmable graphics processor and
  - a scalable parallel computing platform.
- **Particularly designed for data parallelism (SIMD)**

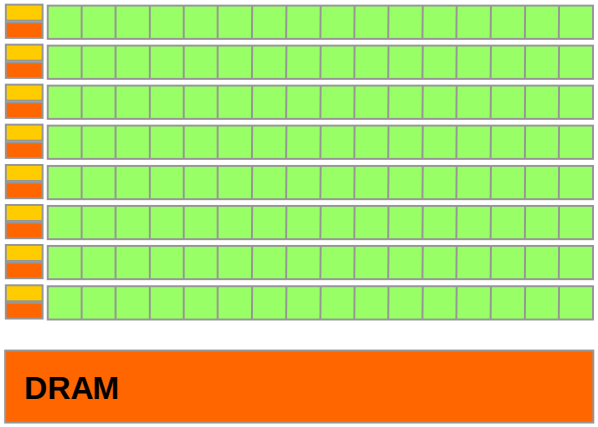
GPU evolution



CPU vs. GPU



CPU



GPU

- General purpose architecture
- Large amount of transistors for non-computational tasks
- Only a few parallel threads possible, thread creation overhead
- Complex memory management (heap, stack)

Optimized for fast sequential processing

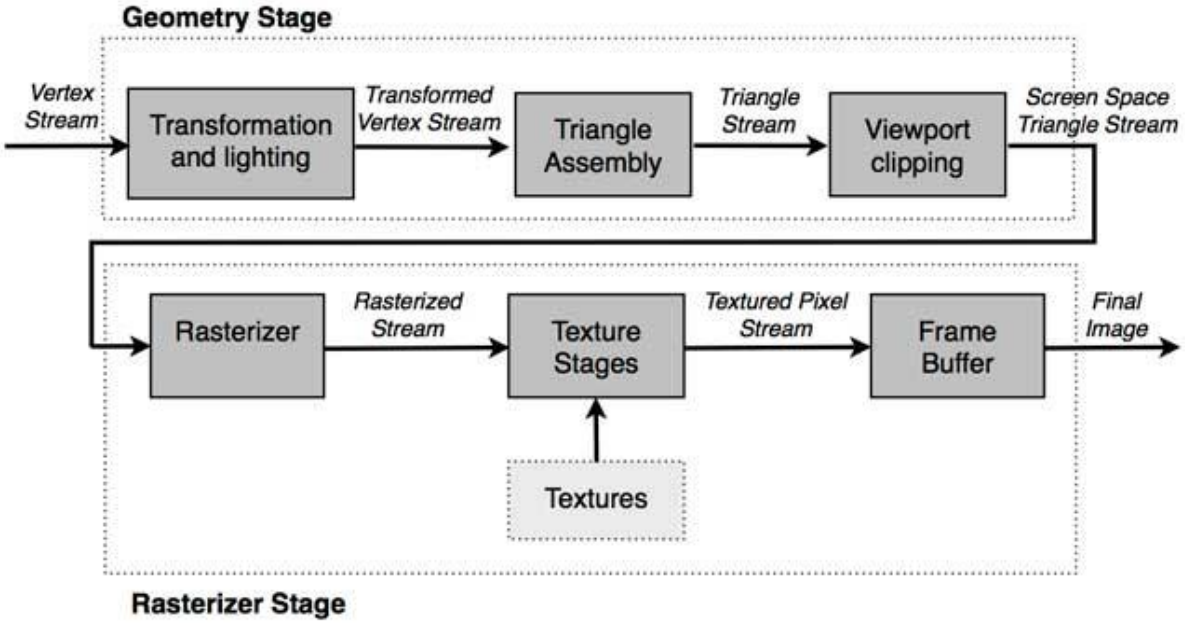
- Many processors dedicated to computations
- Less support for cache and branching
- GPU threads are lightweight, no creation cost, instant switching, thousands of threads possible
- Slow dynamic memory allocation

Perfect for embarrassingly parallel compute intensive problems

History

80s and 90s: Fixed-function graphics pipelines

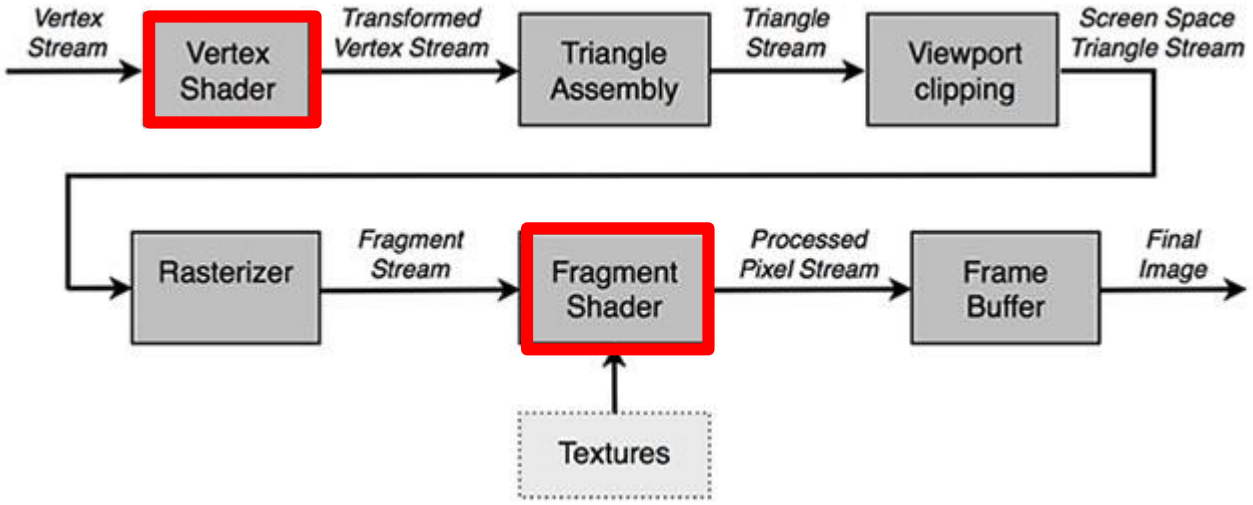
- Hardware configurable, but not programmable
- Receive geometry data as input, process the data through the pipeline blocks, and produce the final output rendered image
- Implementation of graphics APIs (OpenGL, DirectX)



History

2000: Programmable Real-Time Graphics

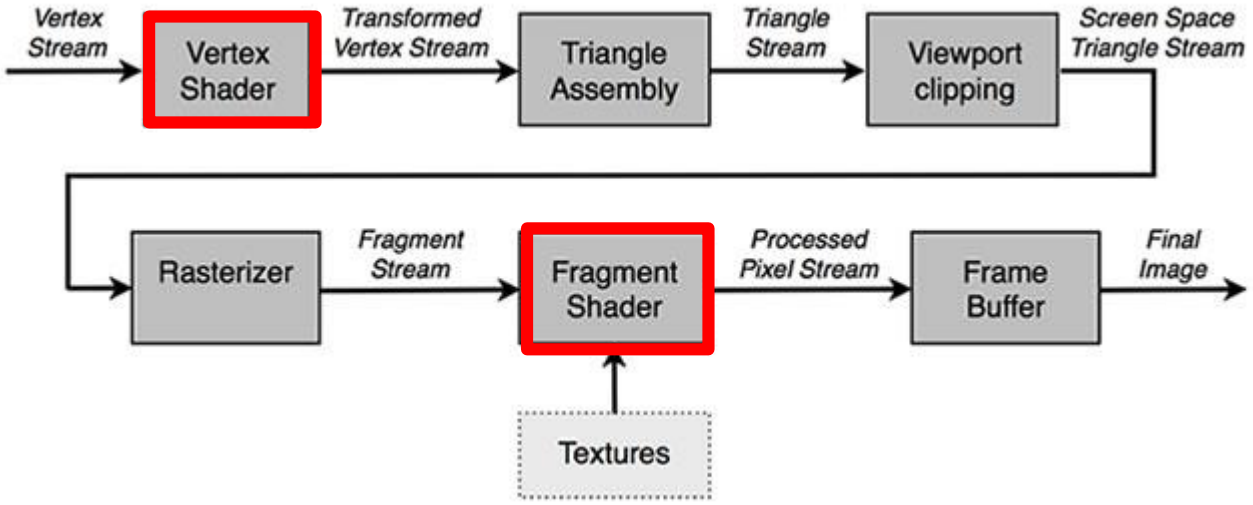
- Shaders: programmable pipeline stages
- Hardware evolves towards massively parallel architectures



History

Early 2000s: GPGPU

- General purpose computing on GPUs
- Non-graphical computations via shader functions
- Driven by performance advantage of GPUs



History

GPUs evolve towards wider-purpose “unified” architectures.

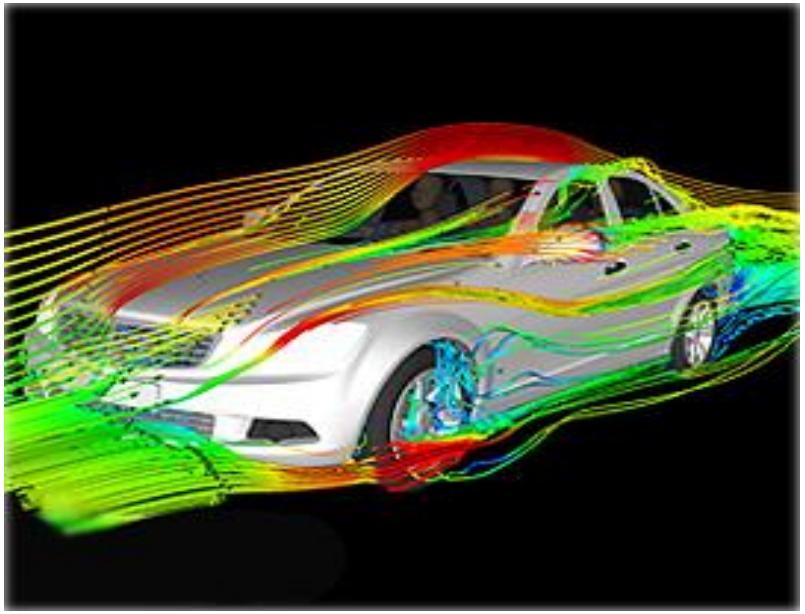
Today

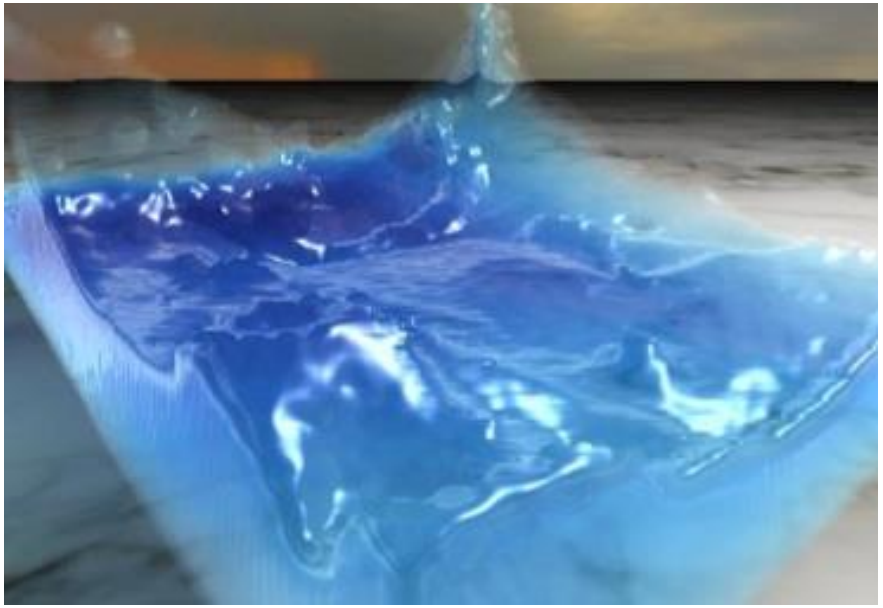
- Thousands of parallel processors
- Graphics pipeline nearly entirely programmable or at least configurable (OpenGL 4)
- Non-graphic API for GPU computing: CUDA, OpenCL
- High-level compiler directives (OpenACC)

Applications

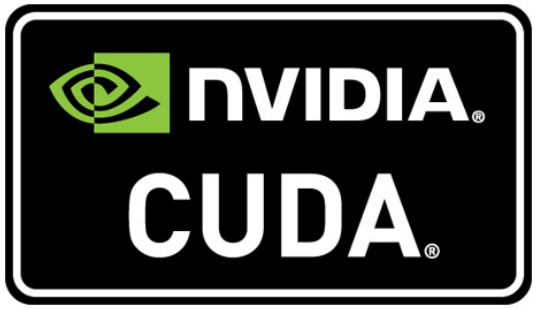
Science & engineering disciplines get 100x or better speedups on GPUs

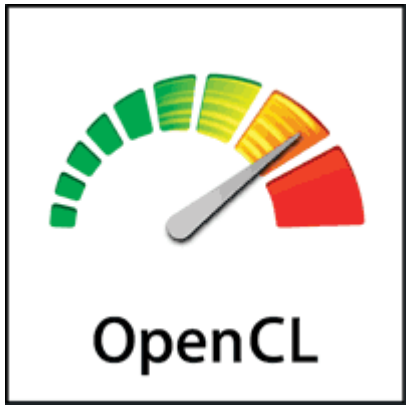
- Game effects (FX), physics, image processing
- Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting
- Artificial Intelligence, Deep Learning





CUDA vs. OpenCL





CUDA	OpenCL
GPU only	Heterogeneous platform CPU fallback
Proprietary (Nvidia) More mature, more libraries Big community	Open and royalty-free (standard formed by Khronos Group)
Develops fast	Slower development
Easier to learn	More difficult to set up

What is CUDA?

- Compute Unified Device Architecture
  - Scalable parallel programming model for unified GPU architecture
  - No graphics context: 1 core = 1 processing unit
  - API (extensions to C/C++)
  - Many libraries (CUBLAS, CUFFT, …)

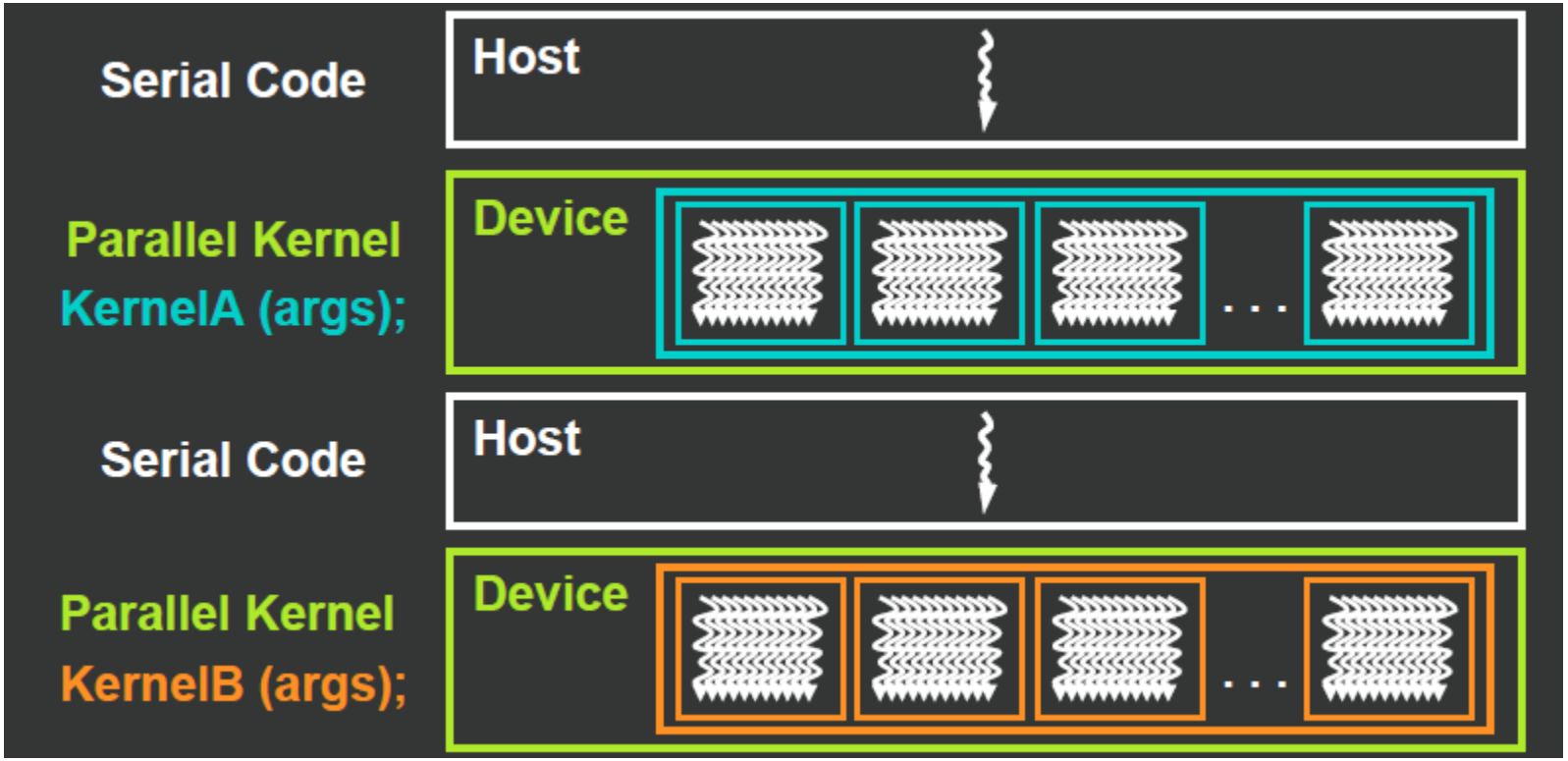
CUDA Programming model

- **Host** – The CPU and its memory (host memory)
- **Device** – The GPU and its memory (device memory)
- **Kernel** – method executed on the Device by many threads in parallel

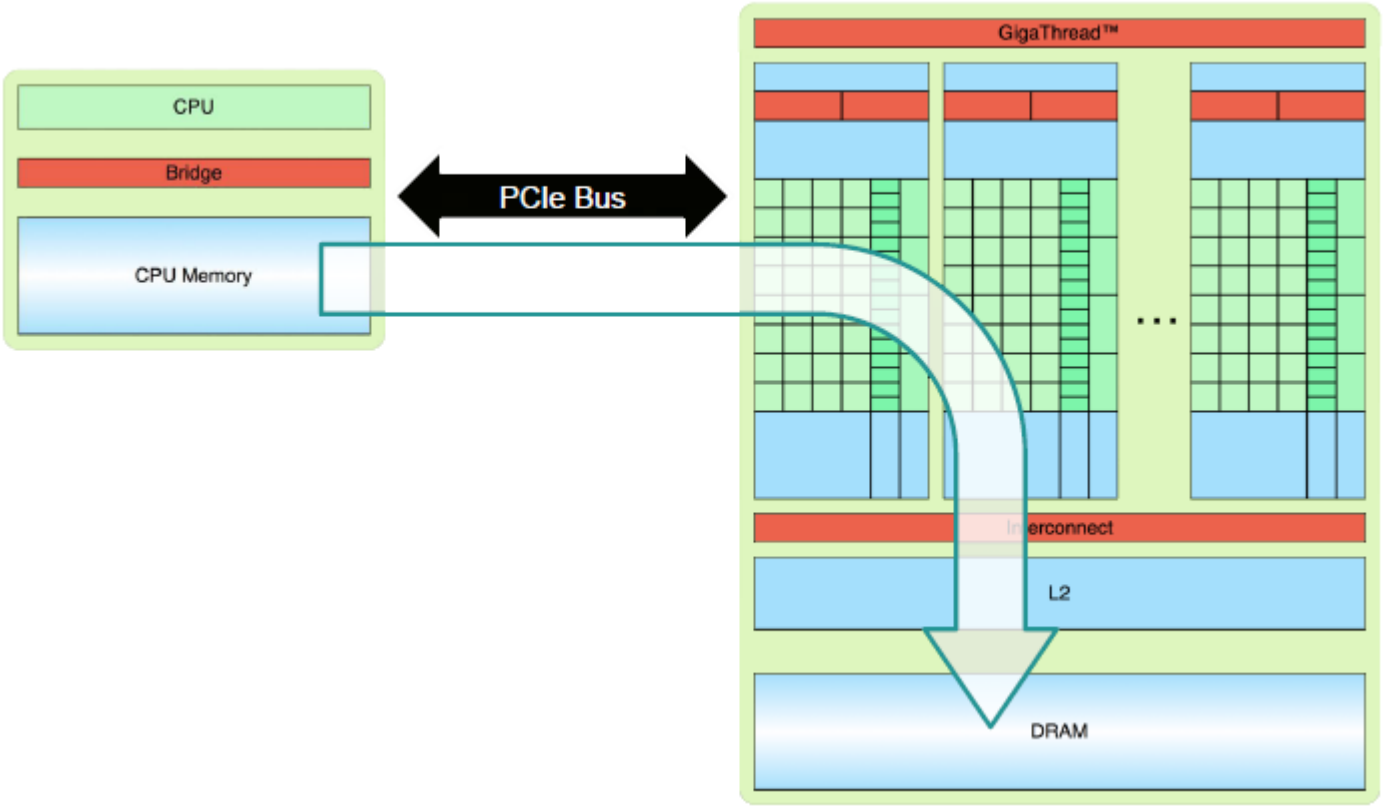
CUDA Programming model

- CPU and GPU coexist in a heterogeneous setting
- GPU considered as a CPU co-processor
- Host code (program control) and device code (GPU) combined in a single C program
  - Host code: can manage memory on device, launch device threads, etc.
  - Device code: consists of massively parallel kernels (SIMD)

CUDA Programming model

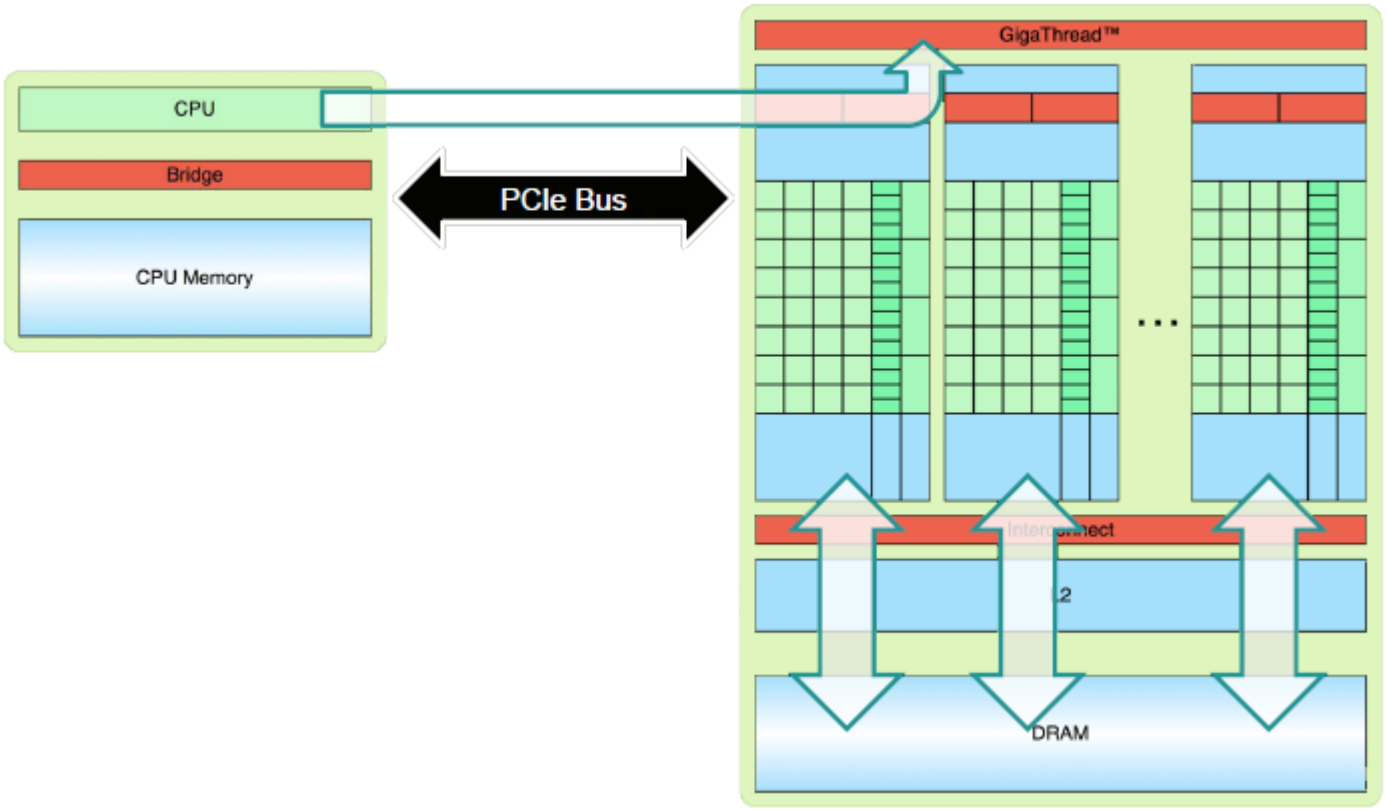


Processing flow



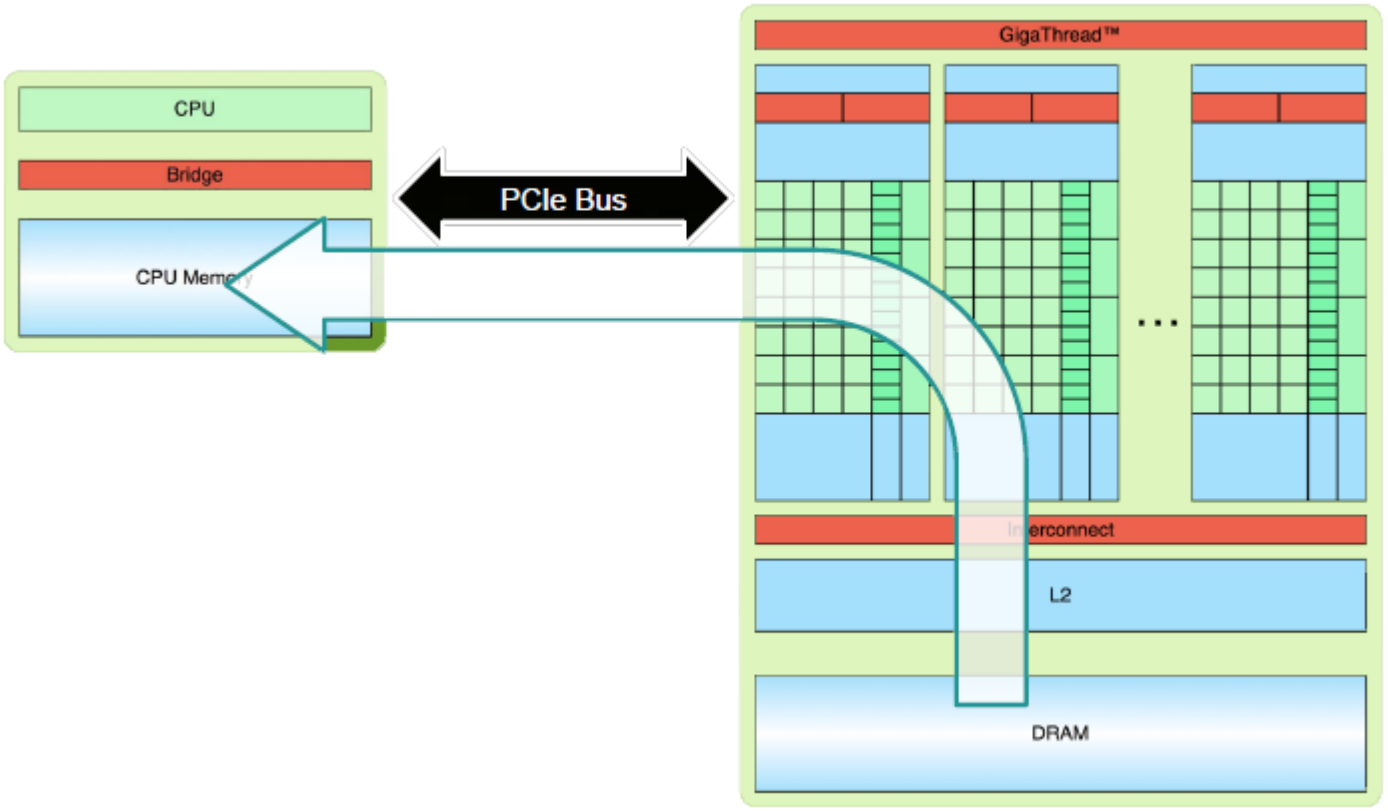
- Data is copied from CPU to GPU.
- Moving data is usually the highest cost of GPU computation.

Processing flow



- Load and launch the kernel code (one kernel at a time)

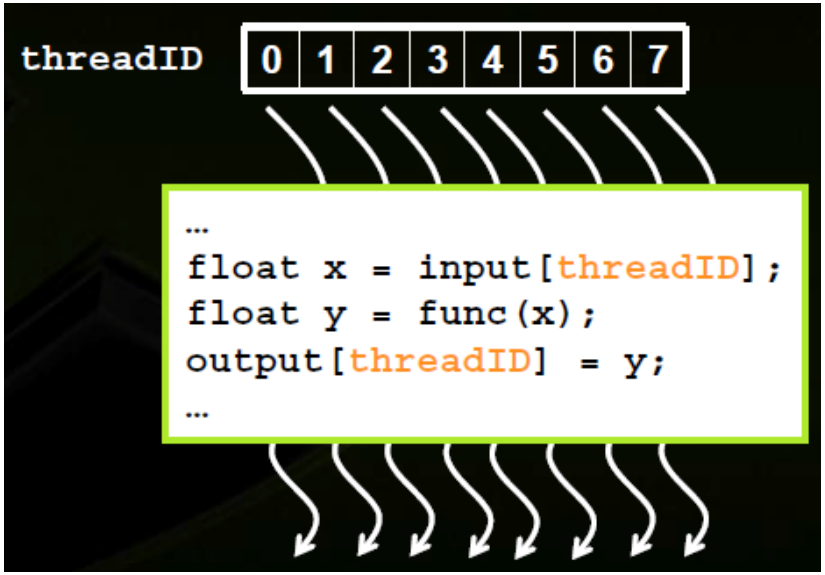
Processing flow



- After GPU finishes computation, the data is copied back to the CPU memory.

Kernel

- One kernel at a time on the device
- Many threads per kernel
- Each thread can execute the same code on different data based on its ID
- Cannot access host memory
- Must have void return type



Grids and Blocks

Blocks:

- Threads are organized in blocks: 1D, 2D or 3D.
- Limited to 1024 threads per block (512 on old cards)
- Threads in one block are always executed in parallel and can use separate shared memory

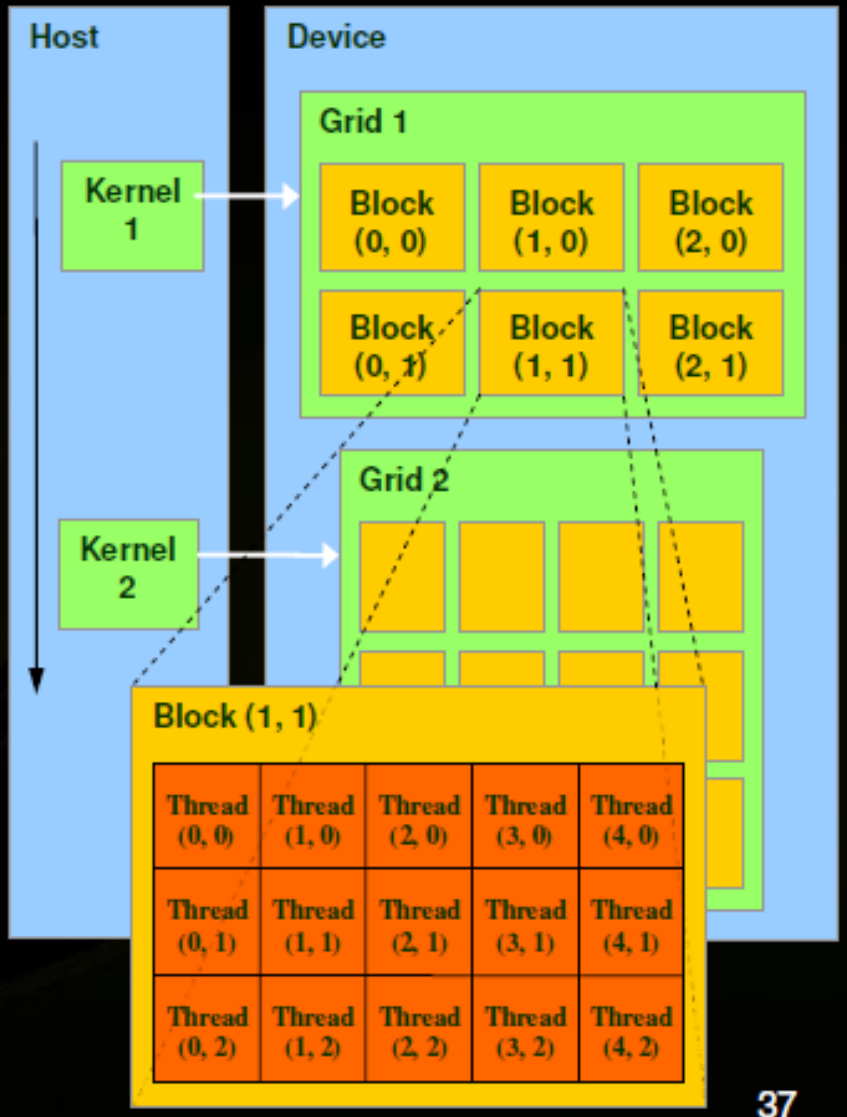
Grids:

- Blocks are organized in a grid: 1D, 2D or 3D.
- Up to 65536 x 65536 x 65536 blocks per grid
  Blocks in a grid may be executed in parallel but not always (in practice, they are scheduled to the available cores)

CUDA Programming Model

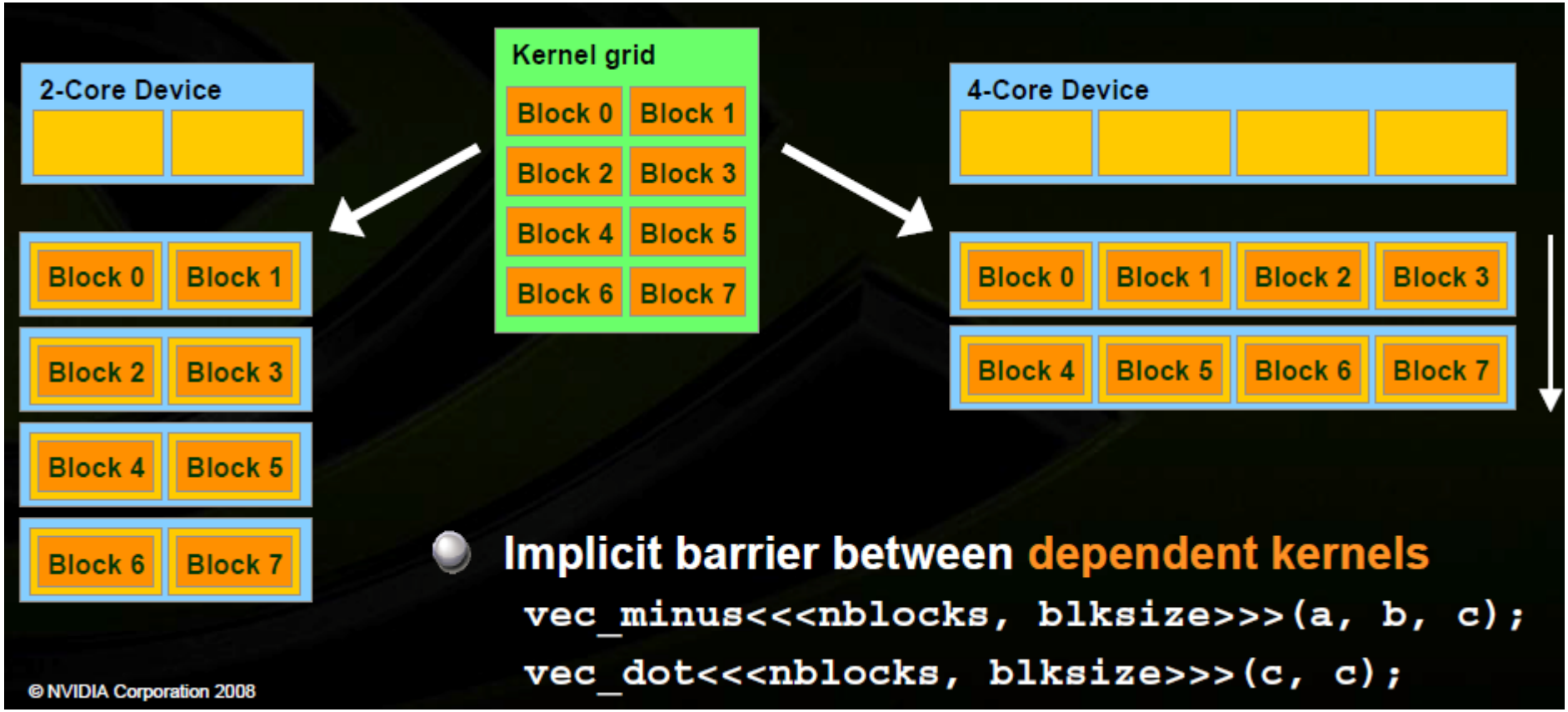
A kernel is executed by a grid of thread blocks

- A thread block is a batch of threads that can cooperate with each other by:
  - Sharing data through shared memory
  - Synchronizing their execution
- Threads from different blocks cannot cooperate

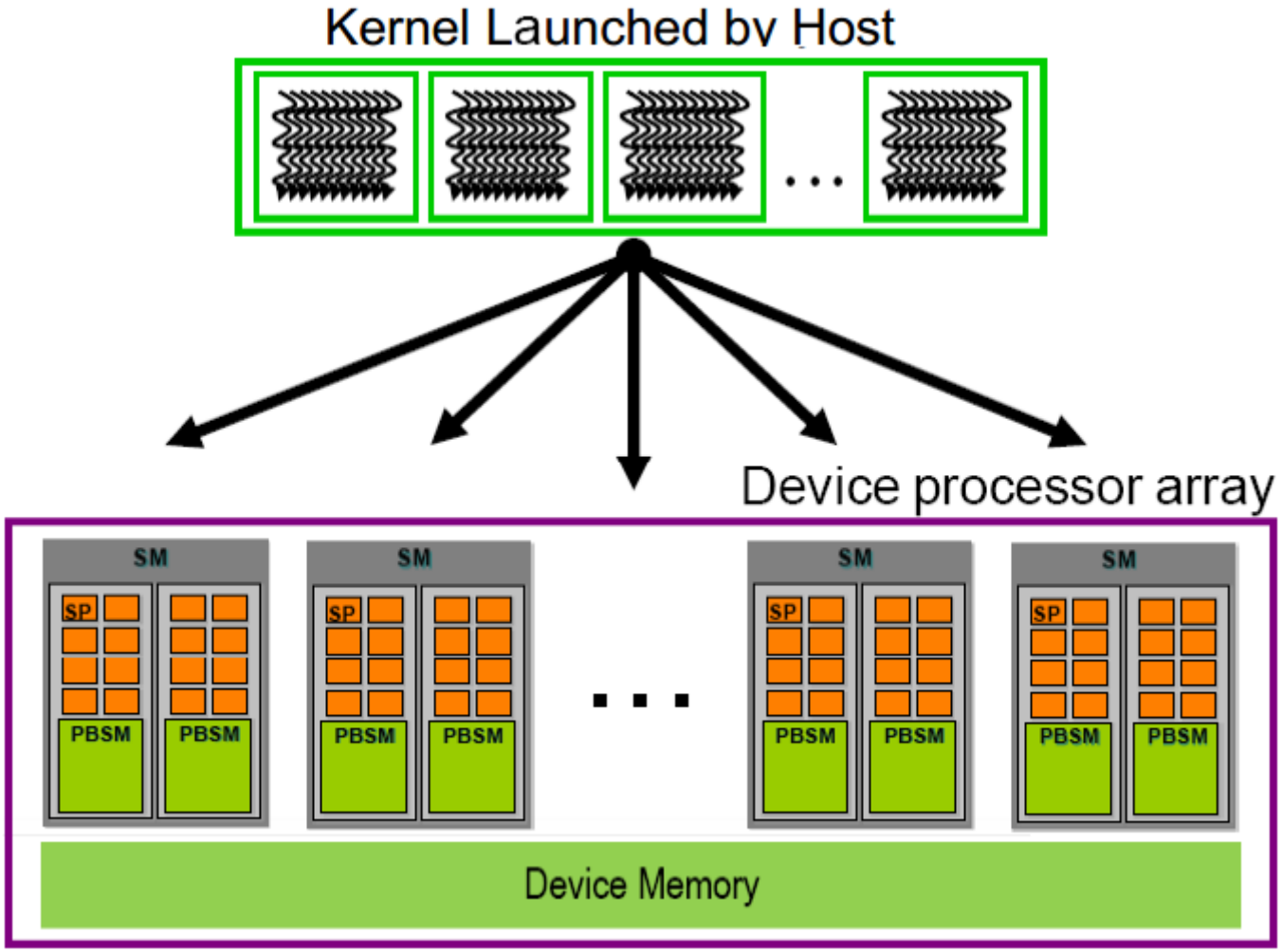


Scalability

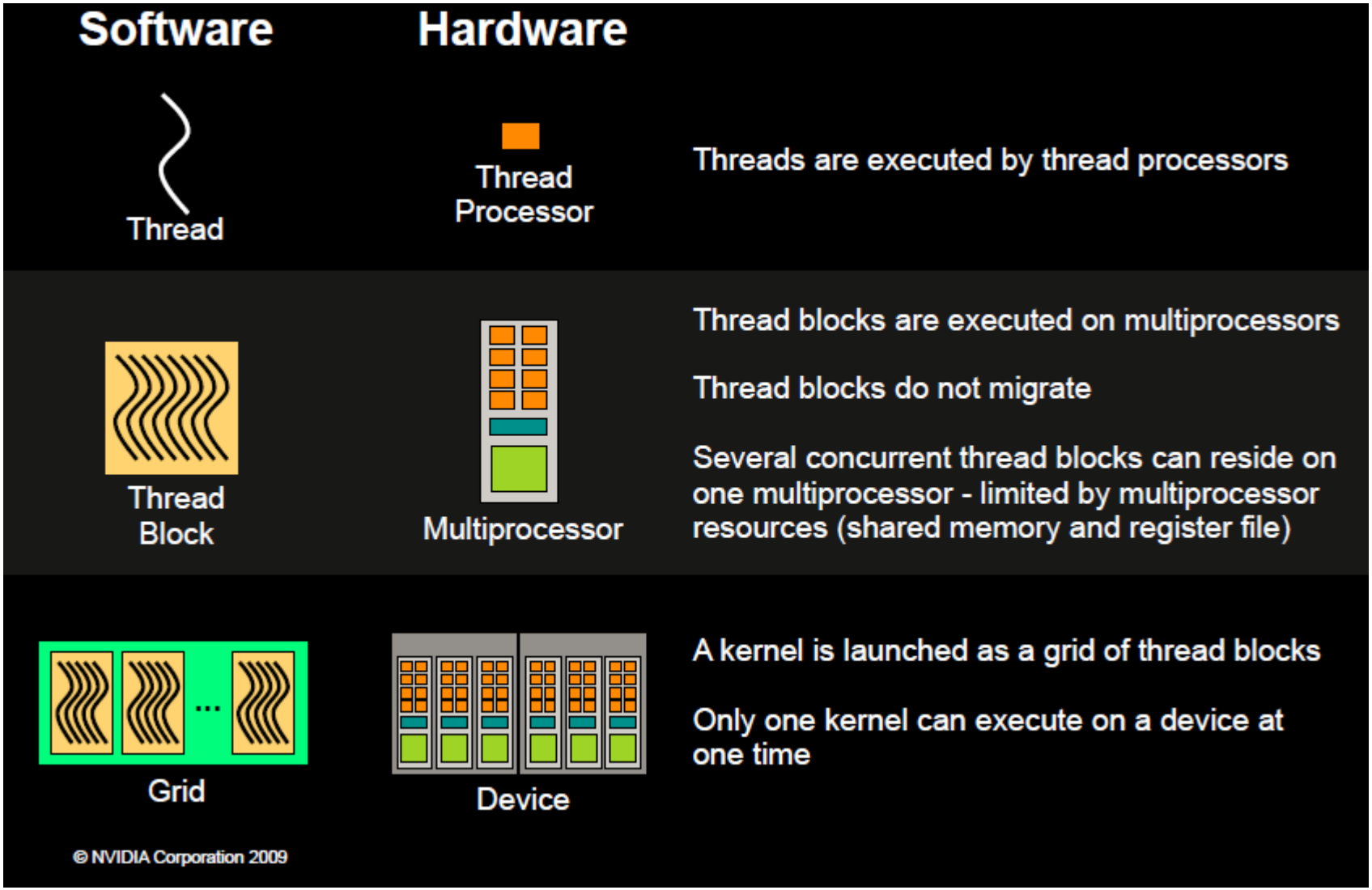
- Thread blocks cannot synchronize
- They can run in any order (concurrently or sequentially)
- This independence gives scalability: hardware is free to schedule thread blocks on any processor



Hardware mapping



Hardware mapping



Memory Management

- Host and device memory are distinct entities and have separate memory spaces.
- Host manages memory on device.
  - Device pointers point to GPU memory: may not be dereferenced from host code
  - Host pointers point to CPU memory: May not be dereferenced from device code

Basic CUDA API for dealing with device memory

- cudaMalloc(), cudaFree(), cudaMemcpy()
- Similar to C equivalents: malloc(), free(), memcpy()

Memory Management

- Users have control over many different GPU memory types:
  - Register
  - Shared Memory
  - Local Memory
  - Global Memory
  - Constant Memory
  - Texture Memory

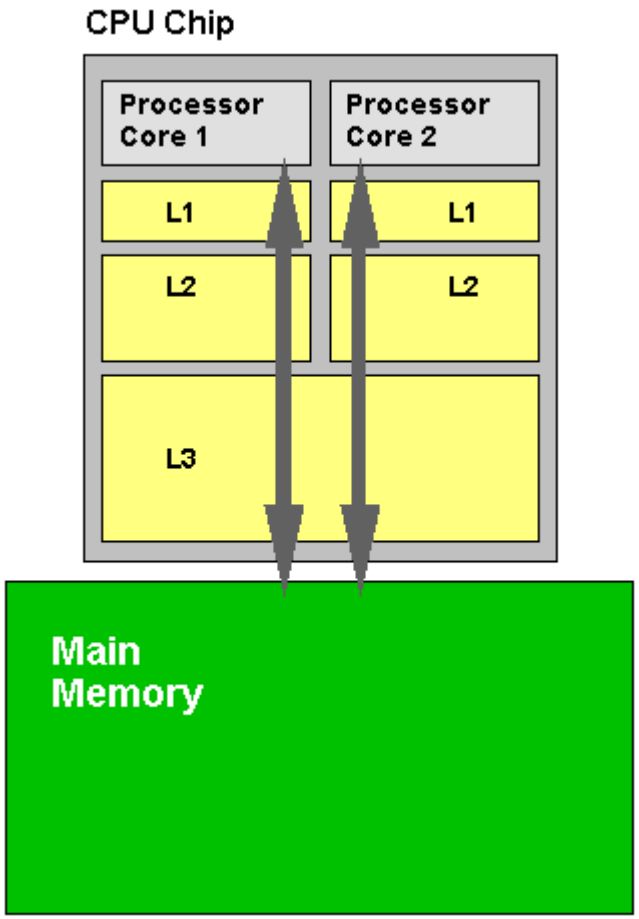
But… there is no traditional cache hierarchy

Cache is expensive

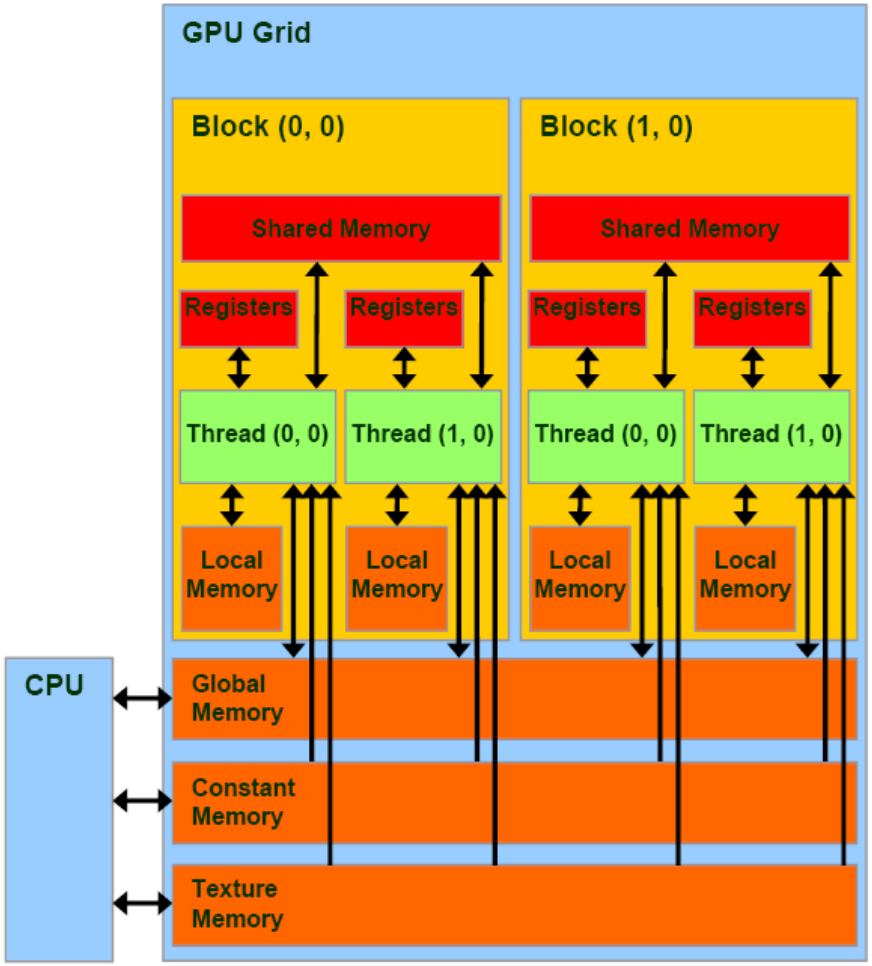
- By running thousands of fast-switching threads, memory latency can be hidden
- Context switching of threads is handled by CUDA
  - Users have little control, only synchronization

Memory Management

CPU memory architecture



GPU memory architecture



CUDA C

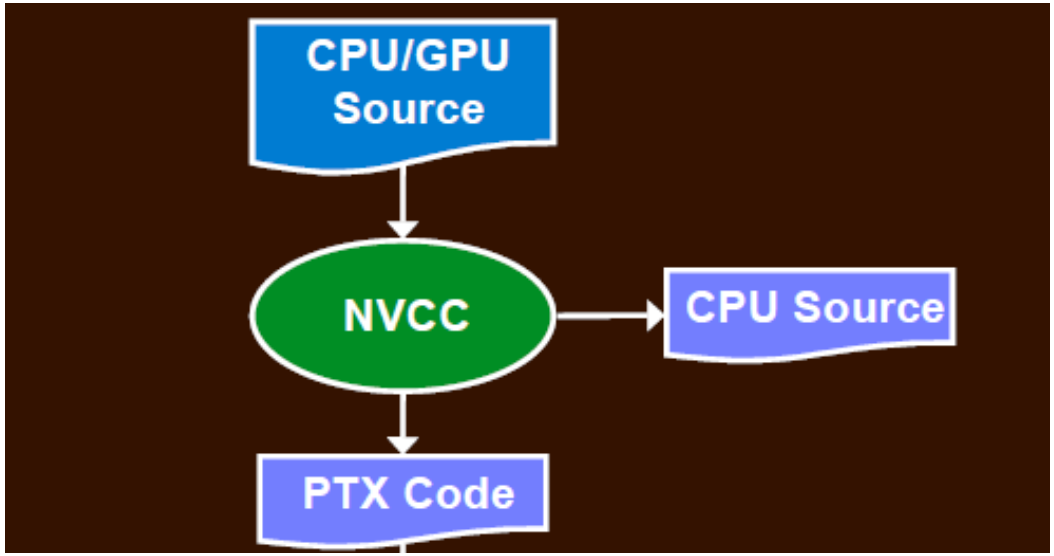
- Programming language for threaded parallelism for GPUs
- Extension of C mixing serial (CPU) code and parallel (GPU) code
  - A serial program that calls parallel kernels
  - Serial code executes on CPU
  - Parallel kernels executed across a set of parallel threads on the GPU
  - Programmer organizes threads into a hierarchy of thread blocks and grids

nvcc

Compiler driver

nvcc splits the CPU and GPU parts of the code

- The CPU parts are compiled with gcc / Microsoft C
- The GPU parts are compiled with ptxas (NV assembler)
- The parts are stitched back together into one big object or executable file.



Environment

- CUDA Toolkit (Windows & Linux)
  - Driver API (cu* functions) for complex low-level programming
  - Runtime API (cuda* functions) simpler and more convenient
  - Compiler: nvcc
  - Libraries (cuFFT, cuBLAS, ...)
  - Tools (debugger, profiler, …)
  - Code samples

Device Management & Inspection

CPU can query and select GPU devices

- `cudaGetDeviceCount (int* count)`
- `cudaSetDevice (int device) // 0 by default`
- `cudaGetDevice (int *current_device)`
- `cudaGetDeviceProperties (cudaDeviceProp* prop, int device)`
- `cudaChooseDevice (int *device, cudaDeviceProp* prop)`
- `...`

Exercice

Install the CUDA samples
https://github.com/NVIDIA/cuda-samples
and run some examples.

In particular, run the device query.
What card do you have, what are its properties?

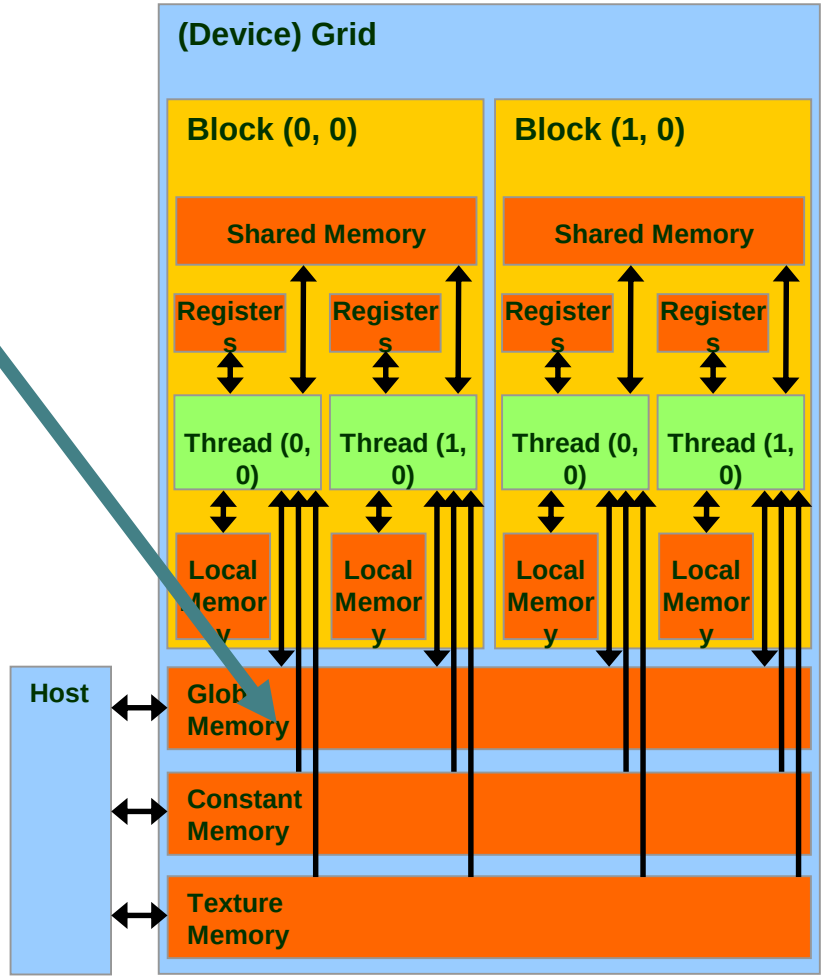
```
// CUDA hello world
#include "cuda_runtime.h";
#include "device_launch_parameters.h";
#include <stdio.h>
#include <stdlib.h>
__global__ void add(float* a, float* b, float* c) {
          c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
int main(void) {
          float *a, *b, *c, *d_a, *d_b, *d_c;
          int size = 1 * sizeof(float);
          a = (float*)malloc(sizeof(float));
          b = (float*)malloc(sizeof(float));
          c = (float*)malloc(sizeof(float));
          *a = 1.0f;
          *b = 1.0f;
          cudaMalloc((void**)&d_a, size);
          cudaMalloc((void**)&d_b, size);
          cudaMalloc((void**)&d_c, size);
          cudaMemcpy(d_a, a, sizeof(float), cudaMemcpyHostToDevice);
          cudaMemcpy(d_b, b, sizeof(float), cudaMemcpyHostToDevice);
          add <<< 1, 1 >>> (d_a, d_b, d_c);
          cudaMemcpy(c, d_c, sizeof(float), cudaMemcpyDeviceToHost);
          printf("%f\n", *c);
          free(a); free(b); free(c);
          cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
          return 0;
}
```

Exercise

Implement and run our « Hello World ».

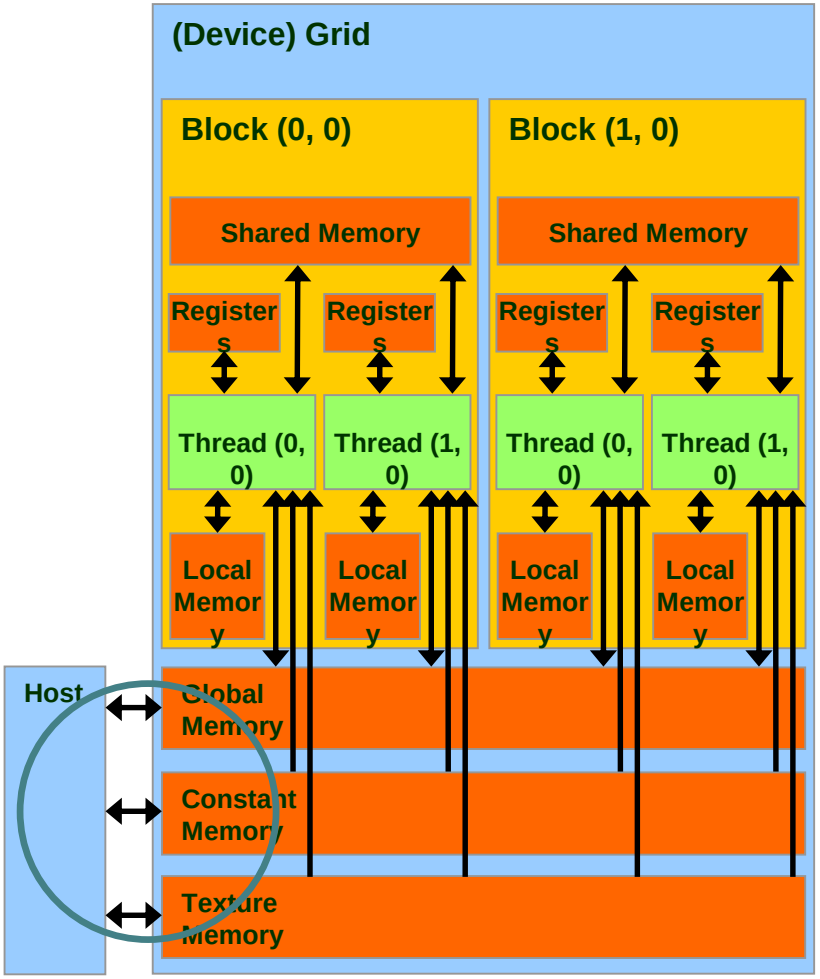
CUDA Memory allocation

- cudaMalloc()
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointe**r to the allocated object
    - **Size** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - Pointer to freed object



CUDA Data Transfer

- cudaMemcpy()
  - Memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
- Blocks the CPU until the copy is complete
- Copy begins when all preceding CUDA calls have completed



Kernel launch

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

```
mykernel<<<dim3 nB, dim3 nT>>>(…)
```

invokes a parallel kernel function on a grid of nB blocks where each block runs nT concurrent threads.

- nT - dimension and size of blocks in threads:
  - Three-dimensional: x, y, z
  - Threads per block: dB.x*dB.y*dB.z
  - Max: 512 x 512 x 64
  - Limited to 1024 parallel threads (old cards: 512)
- nB - dimension and size of grid in blocks
  - Two-dimensional: x and y
  - Blocks launched in the grid: dG.x*dG.y
  - Max: 65535 x 65535 (new cards: Max: 65535 x 65535 x 65535)
- Unspecified dim3 fields initialize to 1

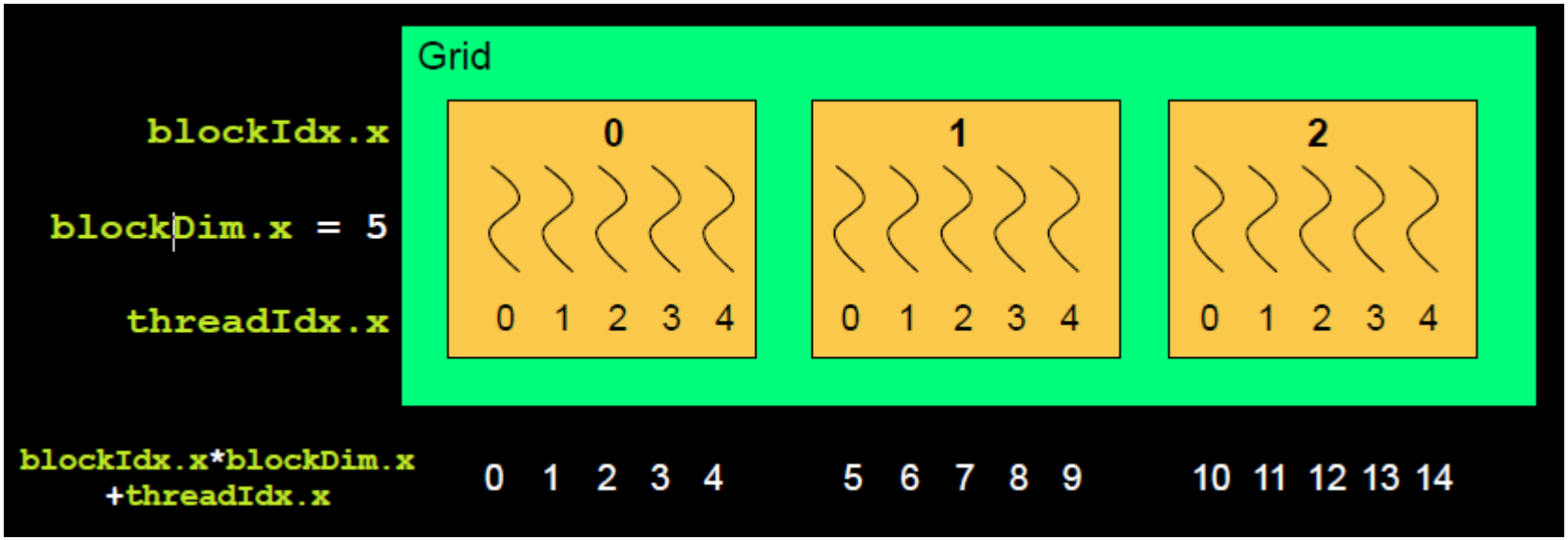
Function qualifiers

	Executed on the:	Only callable from the:
`__device__ float deviceFunc()`	device	device
`__global__ void kernelFunc()`	device	host
`__host__ float hostFunc()`	host	host

- __global__ defines a kernel function
  - Must return void
- __device__ and __host__ can be used together

Thread Indexing

- `kernel<<<nBlocks,nThreads>>>(...)`
- Kernel built-in variables
  - threadIdx.{x,y,z} – thread ID within a block
  - blockIdx.{x,y,z} – block ID within a grid
  - blockDim.{x,y,z} – number of threads within a block
  - gridDim.{x,y,z} – number of blocks within a grid



1D, 2D, 3D Indexing

```
// 1D: N blocks, each having M threads (for arrays)
mykernel<<<N,M>>>(...);
```

⇨

```
blockIdx.x = 0…N-1
threadIdx.x = 0…M-1
```

```
// 2D: N1*N2 blocks, each having M1*M2 threads (for matrices and images)
mykernel<<<dim3(N1,N2),dim3(M1,M2)>>>(...);
```

⇨

```
blockIdx.x = 0…N1-1   blockIdx.y = 0…N2-1
threadIdx.x = 0…M1-1  threadIdx.y = 0…M2-1
```

```
// 3D: N1*N2*N3 blocks, each having M1*M2*M3 threads (for volumetric data)
mykernel<<<dim3(N1,N2,N3),dim3(M1,M2,M3)>>>(...);
```

⇨

```
blockIdx.x = 0…N1-1   blockIdx.y = 0…N2-1     blockIdx.z = 0…N3-1
threadIdx.x = 0…M1-1  threadIdx.y = 0…M2-1    threadIdx.z = 0…M3-1
```

```
#define N 512
__global__ void add (float*a, float *b, float *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
int main(void) {
        float *a, *b, *c, *d_a, *d_b, *d_c;
        int size = N * sizeof(float);
        a = (float*)malloc(size); random_vector (a, N);
        b = (float*)malloc(size); random_vector (b, N);
        c = (float*)malloc(size); "
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
        add<<<N,1>>>(d_a, d_b, d_c); // N blocks
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

```
#define N 512
__global__ void add (float*a, float *b, float *c) {
        c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
int main(void) {
        float *a, *b, *c, *d_a, *d_b, *d_c;
        int size = N * sizeof(float);
        a = (float*)malloc(size); random_vector (a, N);
        b = (float*)malloc(size); random_vector (b, N);
        c = (float*)malloc(size); "
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
        add<<<1,N>>>(d_a, d_b, d_c); // N threads in 1 block
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

```
#define N (1024*1024)
#define M 512
__global__ void add (float*a, float *b, float *c) {
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        c[index] = a[index] + b[index];
}
int main(void) {
        float *a, *b, *c, *d_a, *d_b, *d_c;
        int size = N * sizeof(float);
        a = (float*)malloc(size); random_vector (a, N);
        b = (float*)malloc(size); random_vector (b, N);
        c = (float*)malloc(size);
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
        add<<<N/M,M>>>(d_a, d_b, d_c);
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

```
#define N 1432657
#define M 512
__global__ void add (float*a, float *b, float *c, int n) {
        int index = threadIdx.x + blockIdx.x * blockDim.x;
        if (index < n)  c[index] = a[index] + b[index];
}
int main(void) {
        float *a, *b, *c, *d_a, *d_b, *d_c;
        int size = N * sizeof(float);
        a = (float*)malloc(size); random_vector (a, N);
        b = (float*)malloc(size); random_vector (b, N);
        c = (float*)malloc(size);
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);
        cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
        add<<<(N + M-1)/M,M>>>(d_a, d_b, d_c, N);
        cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
        free(a); free(b); free(c);
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
}
```

Exercise

Implement the previous vector addition and compare their result to a CPU version (« golden model »).

To estimate the error, use the root-mean-square deviation:

```
float comparef (float* a, float* b, int n) {
  double diff = 0.0;
  for (int i=0;i<n;i++)
    diff += (a[i]-b[i])*(a[i]-b[i]);
  return (float)(sqrt(diff/(double)n));
}
```

Error handling

All CUDA calls return a cudaError_t value.

```
cudaError_t err;
err = cudaMemcpy(d, h, nbytes, cudaMemcpyDeviceToHost);
if (err != cudaSuccess) {
      printf("Error: %s\n", cudaGetErrorString(err));
}
```

However, only a few bad things can be caught at launchtime:

- No CUDA-capable GPU found
- Using features that your GPU does not support (double-precision, …)
- Too many blocks or threads

Error handling

- Most bugs cannot be caught until after the launch:
  - Array overruns
  - Division by zero, NaN, …
  - Dereferencing zero pointers

CUDA kernel invocations do not return any value. Kernel errors call can be checked after execution by calling cudaGetLastError():

```
cudaError_t err;
kernel<<<N,M>>>( ... );
cudaDeviceSynchronize();
err = cudaGetLastError();
if (err != cudaSuccess) {
  printf("Error: %s\n", cudaGetErrorString(err));
}
```

Error handling… more beautifully

```
#define checkCudaErrors(err)  __checkCudaErrors(err, __FILE__, __LINE__)

inline void __checkCudaErrors
(cudaError err, const char *file, const int line )
{
    if (err!=cudaSuccess)
    {
        fprintf(stderr, "%s(%i) : CUDA Runtime API error %d: %s.\n",
           file, line, (int)err, cudaGetErrorString(err));
        system("pause");
        exit(1);
    }
}
```

checkCudaErrors(cudaMemcpy(d, h, nbytes, cudaMemcpyDeviceToHost));

Device side printf()

- SM 2.0 (Fermi) and above only
- C-style format string
- Buffered output flushing only at explicit sync points
- Unordered (multi threaded output)
- Include the stdio.h header
- Compile for Fermi: *nvcc –arch=compute_20 –o output test.cu*

```
#include <stdio.h>
__device__ int var = 42;

__global__ void kernel(void)
{
  if (threadIdx.x == 0)
    printf("var:%d\n", var);
}

int main(void)
{
  kernel<<<1,1>>>();
  cudaDeviceSynchronize();
  cudaDeviceReset();
}
```

```
$ ./demo_printf
var:42
```

Device side assert()

- SM 2.0 (Fermi) and above only
- Stops if conditional == 0
- Prints message to stderr
- Printf() flushing rules apply
- Stops all subsequent host side calls with cudaErrorAssert
- Include the assert.h header
- Compile for Fermi:
- *nvcc –arch=compute_20 –o output test.cu*

```
#include <assert.h>
__device__ int var;

__global__ void kernel(void)
{
  assert(threadId.x <= 16);
}

int main(void)
{
  kernel<<<1,18>>>();
  cudaDeviceSynchronize();
  cudaDeviceReset();
}
```

```
$ ./demo_assert
/tmp/test_assert.cu:7: void
kernel(): block: [0,0,0],
thread: [17,0,0] Assertion
`threadIdx.x <=16` failed.
```

Exercise

- Write a program causing a
  - CUDA function error
  - Kernel runtime error

  and display it with the checkCudaErrors macro.

- Write a program with a 2D-indexed kernel. Each thread prints out its block index (x,y) and its thread index (x,y).

For more advanced debugging techniques, have a look at http://on-demand.gputechconf.com/gtc/2014/presentations/S4578-cuda-debugging-command-line-tools.pdf

Timing CUDA kernels

```
cudaEvent_t cstart, cstop;
float cdiff;
cudaEventCreate(&cstart);
cudaEventCreate(&cstop);
cudaEventRecord(cstart);
kernel<<<N,M>>>(a,b,c);
cudaEventRecord(cstop);
cudaEventSynchronize(cstop);
cudaEventElapsedTime(&cdiff, cstart, cstop);
printf("CUDA time is %.2f ms\n", cdiff);
cudaEventDestroy(cstart);
cudaEventDestroy(cstop);
```

Exercise

Time a huge vector addition (16*1024*1024 cells) using different block sizes:

1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024.

What do you observe?

Attention: under Visual Studio, when mesuring CPU/GPU performance, always compile and run your programs in release mode.

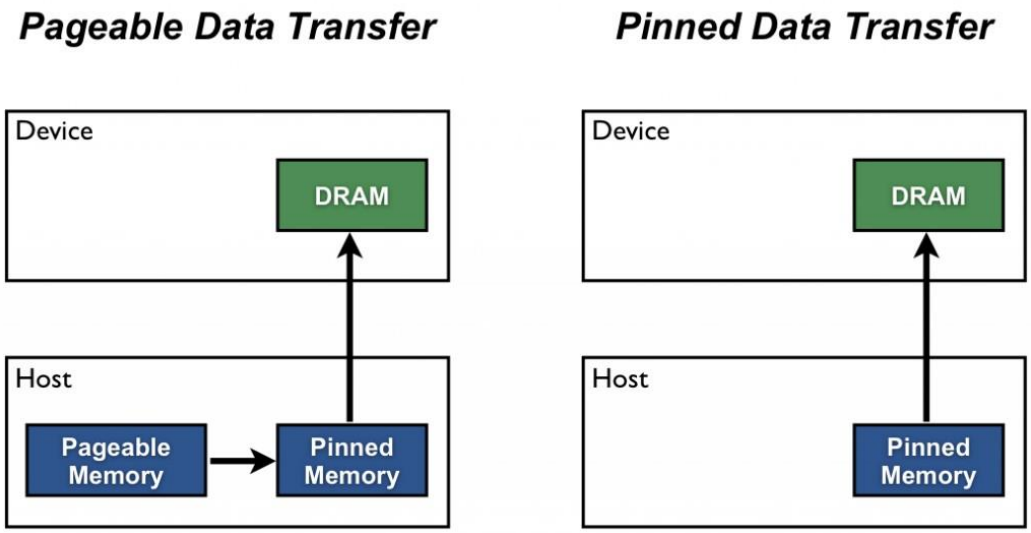
GPU Performance metrics

Choose the right metric:

- Bandwidth: for memory-bound kernels
  - The speed at which the GPU can access its memory
- GFLOP/s: for compute-bound kernels
  - Floating Point Operations Per Second

Pageable vs. Pinned host memory

- The GPU cannot access data directly from pageable host memory.
- When a data transfer from pageable host memory to device memory is invoked, the CUDA driver must first allocate a temporary page-locked, or “pinned”, host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory
- You can directly allocate pinned host memory using **cudaMallocHost()** instead of malloc, and deallocate it with **cudaFreeHost()** instead of free



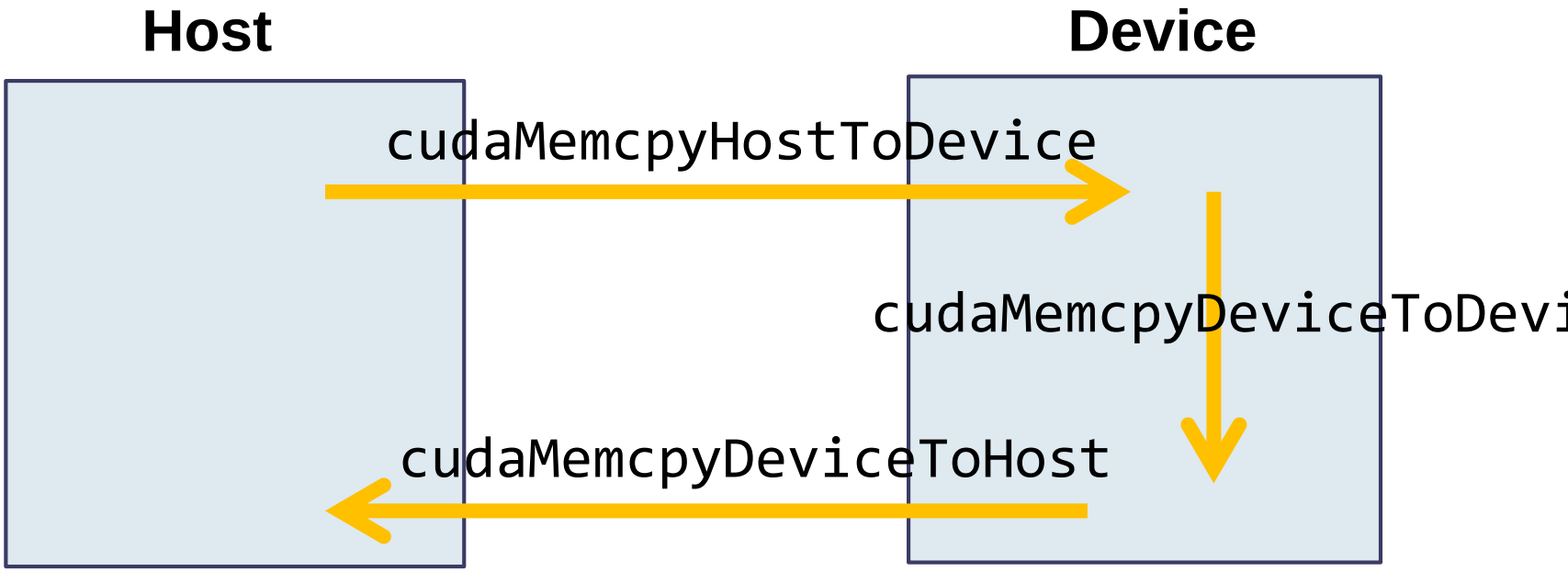
Pinned memory transfers are faster!

Excessive pinned memory may degrade system performance.

Exercise

- Execute the CUDA bandwith test (sample code).
- Now benchmark for yourself . Do you get the same results?
  - Copy a huge array from host to device, from device to device and back to host.
  - Bandwidth host ⇔ device = 1000*datasize / (1000^3*t_in_ms) Gb/s
  - Bandwidth device ⇔ device = 2*1000*datasize / (1000^3*t_in_ms) Gb/s

Pay attention to pageable vs. pinned host memory.



Exercise

Benchmark the performance of compute-bound problems, by executing a compute-intensive kernel.

- Take an array of N cells and initialize it to values between 0…1
- In each cell, compute T times the chaotic series

$$x_{n+1} = 4 * x_n * (1-x_n)$$

$$\text{GFlops} = \frac{3 * N * T * 1000}{t\_in\_ms * 10^9}$$

How many GFlops do you get?
How much faster is the GPU compared to the CPU?