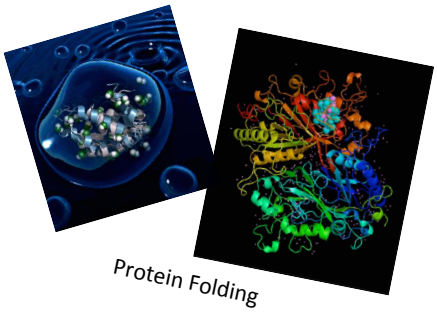
Introduction to GPUs

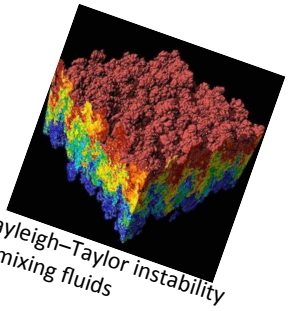
Nikos Hardavellas

Some slides/material from:
UToronto course by Andreas Moshovos
UIUC course by Wen-Mei Hwu and David Kirk
UCSB course by Andrea Di Blas
Universitat Jena by Waqar Saleem
NVIDIA by Simon Green and many others

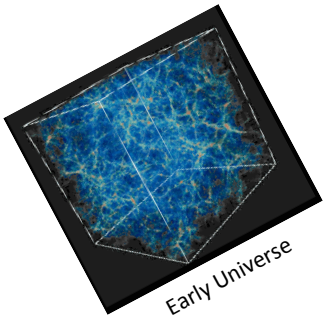
We have big computational challenges to solve



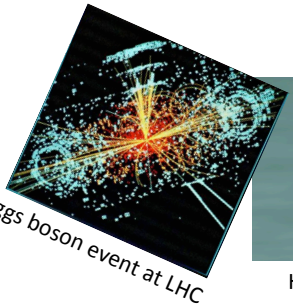
Protein Folding



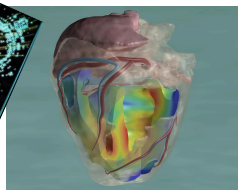
Rayleigh–Taylor instability in mixing fluids



Early Universe



Higgs boson event at LHC

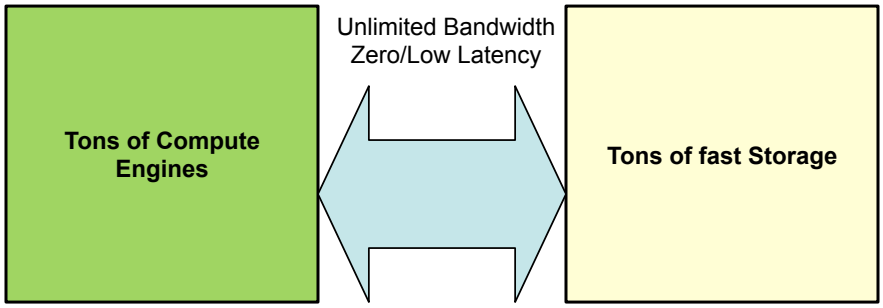


Heart simulation

What computation machine do we want?

Understanding Semiconductor Technology Limitations

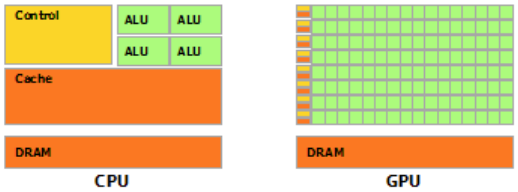
This is what we would like to have



➡ Great, but good luck getting it. What can we actually get?

Let's see what we can get: **Calculation Capability**

- **How many calculation units can be built?**
- Today's silicon chips
  - 15B+ transistors (e.g., NVIDIA GP100)
  - 52b multiplier
    - 30K transistors
    - ~500K multipliers
  - Chip area:
    - 260 mm^2 (mid-range)
    - 600 mm^2 (high-end, really large)
  - 85 µm^2 for FP unit (overestimated)
    - ~3-7K FP units
- Frequency ~ 3Ghz common today



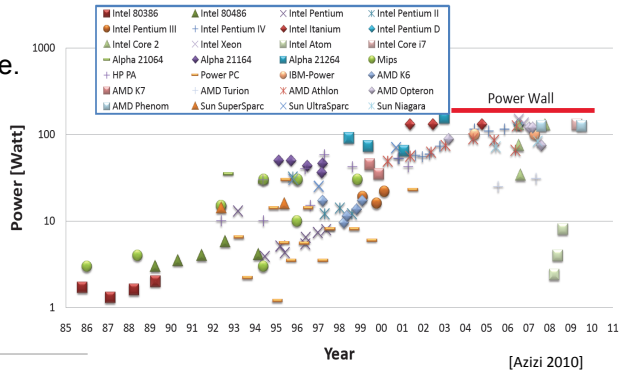
➡ Can build lots of calculation units! (Trend: even more)

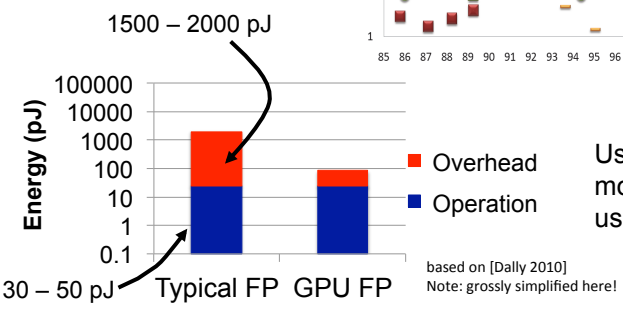
➡ But, why GPUs and not Multicores with 1000+ cores?

Why GPUs and not Multicores with 1000+ cores?

Power is a limited resource. Chips have already hit a *"Power Wall"*.

How can we get more performance?





based on [Dally 2010]
Note: grossly simplified here!

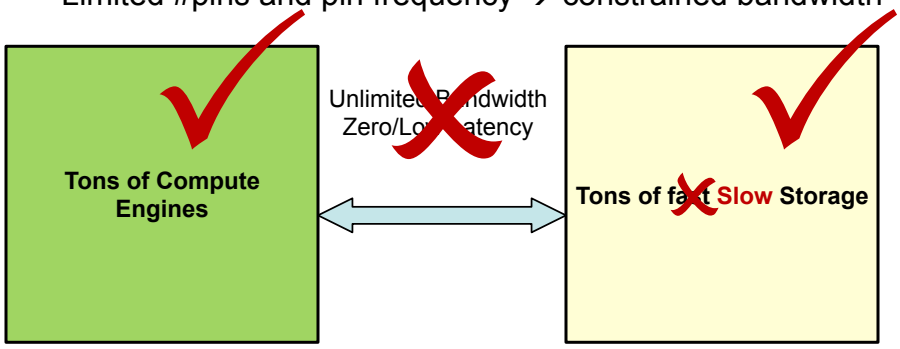
Use the available power more **efficiently** to get more useful work done per Watt.

➡ General-purpose cores have too much overhead.

➡ Amortize overhead among many arith. units → GPUs

How about Communication/Storage

- Need data feed and storage
  - 1000's cores need tons of storage to be kept busy
  - The larger the slower
  - Multiple cycles to access (even on die) → high latency
  - Limited #pins and pin frequency → constrained bandwidth



→ How can we use 1000's cores + tons of storage?

→ PARALLELISM !!!

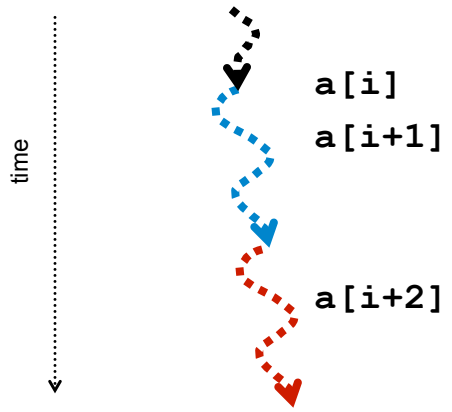
Some things are naturally parallel

e.g., image fading…

How do we program an image fader?
I'll introduce three programming models in three slides!

Sequential Execution Model

```
int a[N]; // a is image, N is large
for (i =0; i < N; i++){
  a[i] = a[i] * fade;
}
```



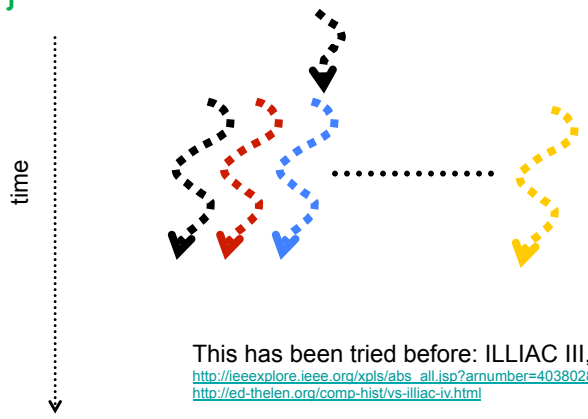
Flow of control / Thread
One instruction at the time
Optimizations possible at the machine level

This is the predominant CPU model

Lots of optimizations to shorten the time required for each individual operation

Data Parallel Execution Model / SIMD

```
int a[N]; // N is large
for all elements do in parallel {
  a[i] = a[i] * fade;
}
```

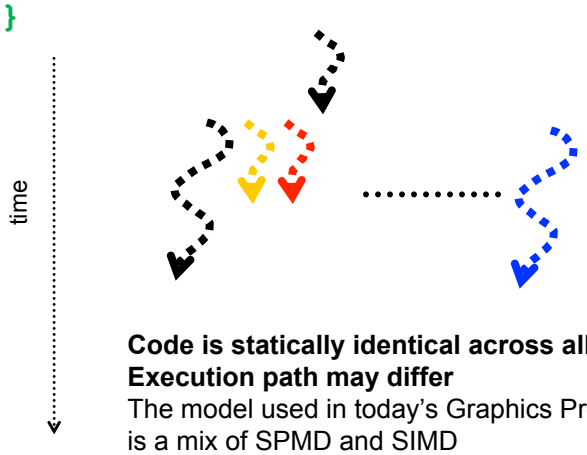


Most modern CPUs offer some limited support for SIMD

This has been tried before: ILLIAC III, UIUC, 1966
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4038028&tag=1
http://ed-thelen.org/comp-hist/vs-illiac-iv.html

Single Program Multiple Data / SPMD

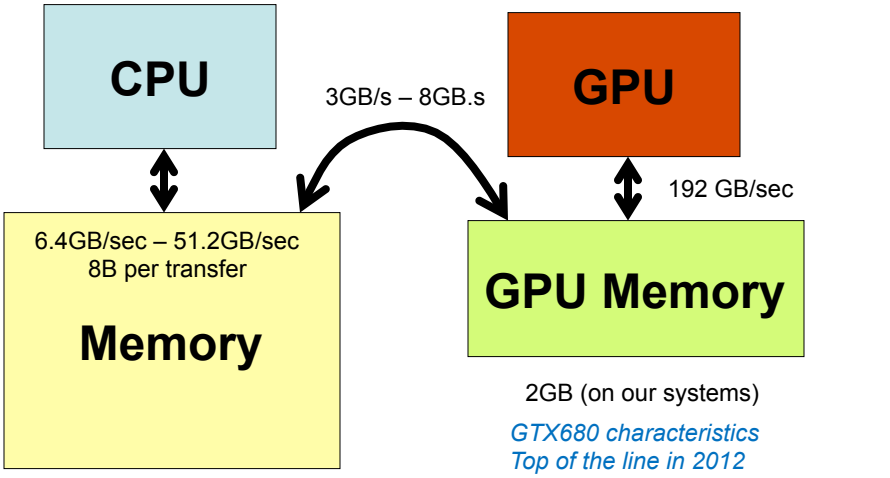
```
int a[N]; // N is large
for all elements do in parallel {
  if (a[i] > threshold) a[i]*= fade;
}
```



Code is statically identical across all threads
Execution path may differ
The model used in today's Graphics Processor Units (GPUs) is a mix of SPMD and SIMD

Programmer's view

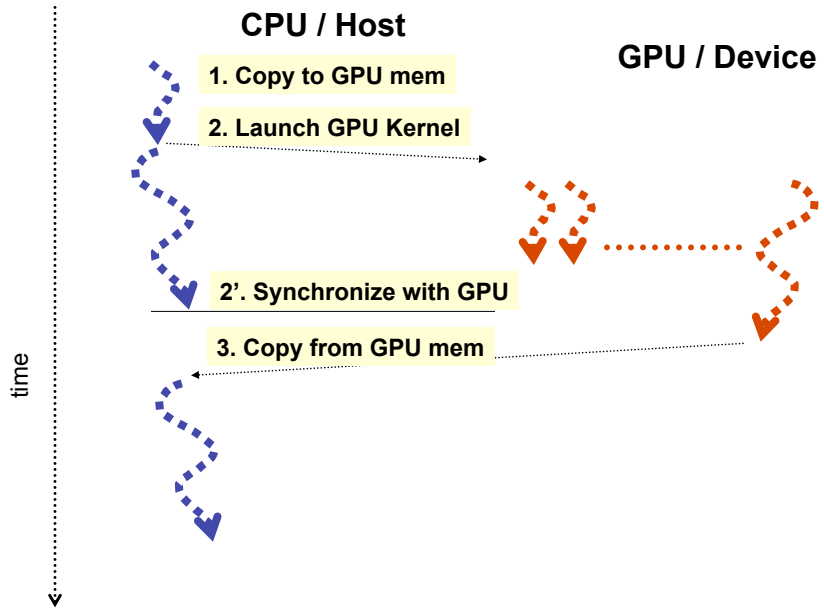
- GPU as a co-processor



GTX680 characteristics
Top of the line in 2012
Data illustrated from that timeframe

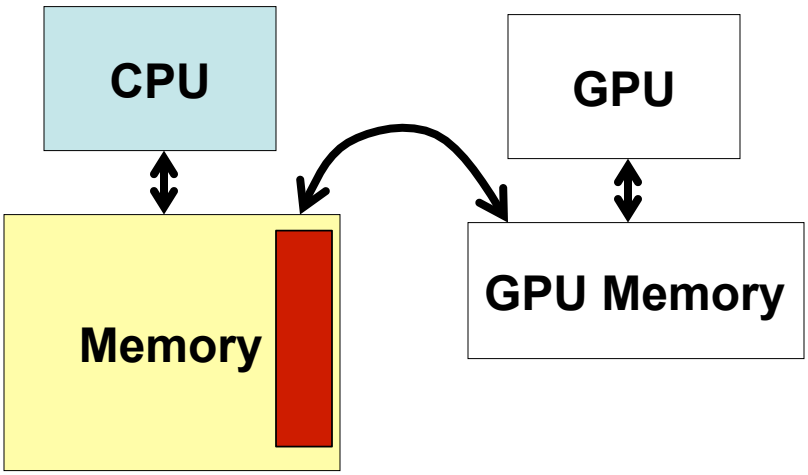
Key Suppliers: Nvidia and AMD

Execution Timeline



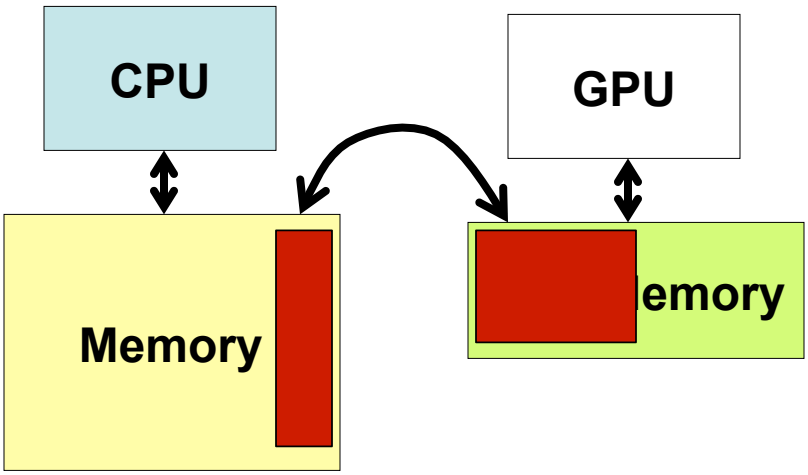
Programmer's view

- First create data on CPU memory



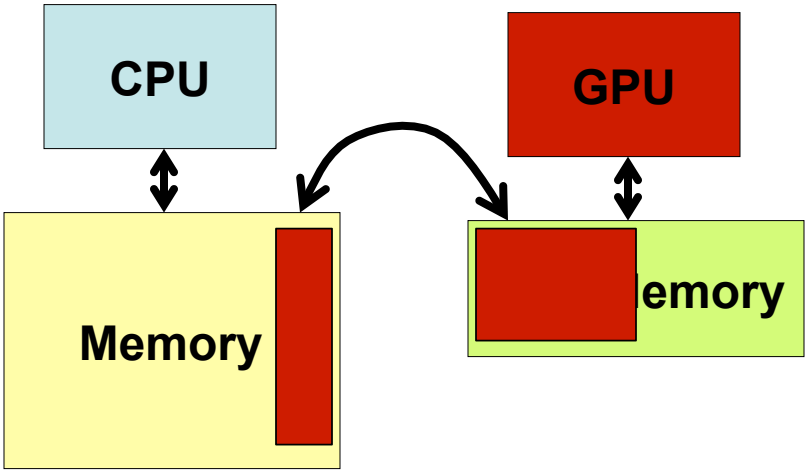
Programmer's view

- Then Copy to GPU



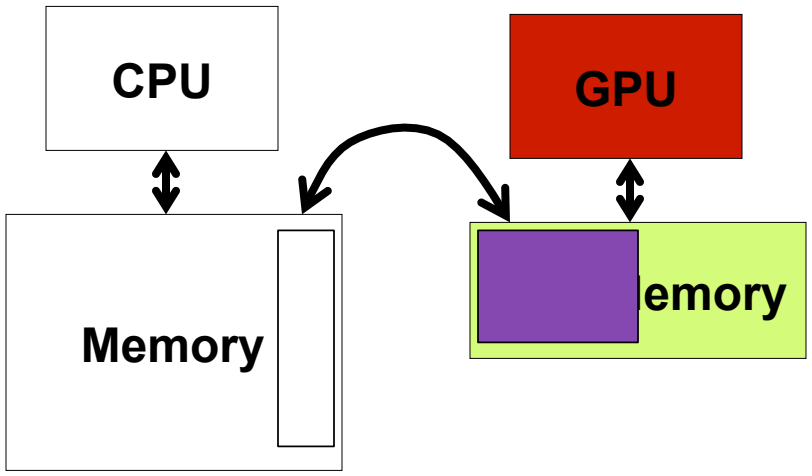
Programmer's view

- GPU starts computation → runs a **kernel**
- CPU can also continue



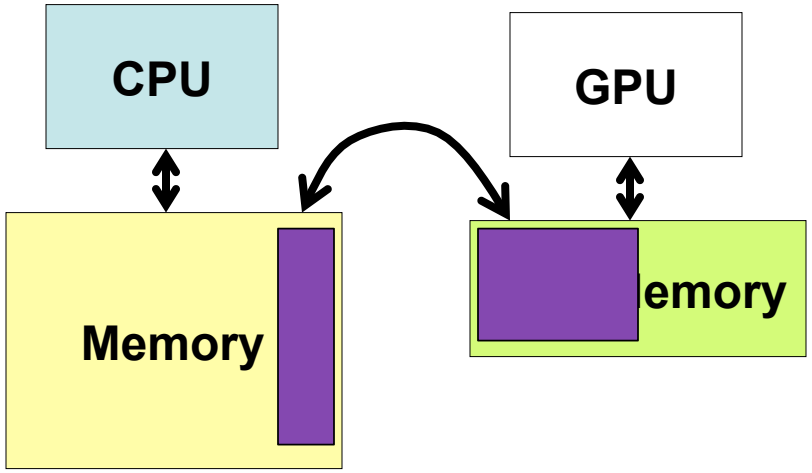
Programmer's view

- CPU and GPU Synchronize



Programmer's view

- Copy results back to CPU



Computation partitioning:

- **CUDA exposes the hardware to the programmer**
- **Programmer must manually partition work appropriately**

- At the highest level:
  - Think of computation as a series of loops:
    - for (i = 0; i < big_number; i++)
      - a[i] = some function
    - for (i = 0; i < big_number; i++)
      - a[i] = some other function
    - for (i = 0; i < big_number; i++)
      - a[i] = some other function

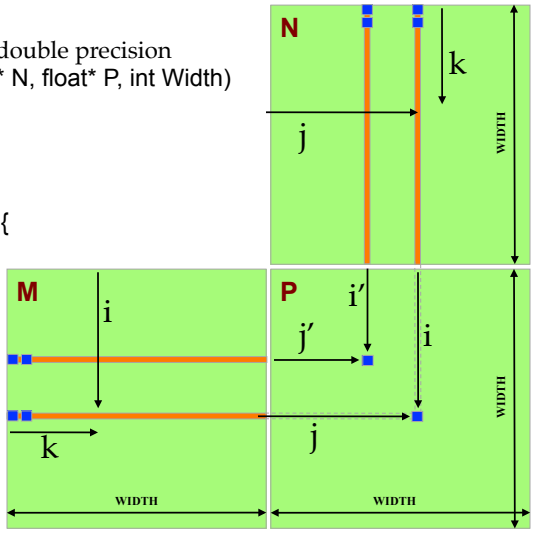
Kernels

  - Per-Kernel partitioning: Think of data as an array

Example: Matrix Multiplication

P = M x N

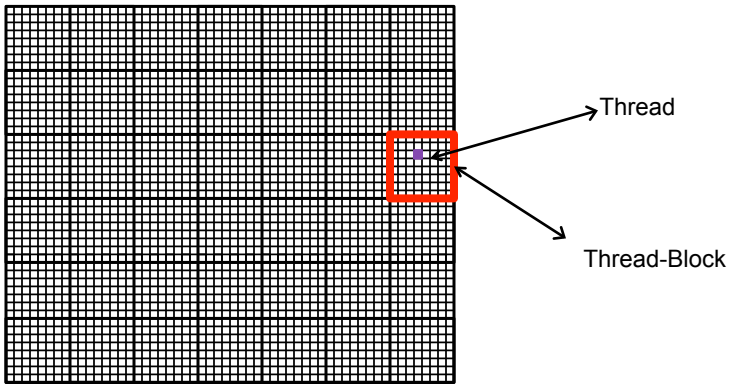
```
// Matrix mul on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
   for (int i = 0; i < Width; ++i)
      for (int j = 0; j < Width; ++j) {
         double sum = 0;
         for (int k = 0; k < Width; ++k) {
            double a = M[i * width + k];
            double b = N[k * width + j];
            sum += a * b;
         }
         P[i * Width + j] = sum;
      }
}
```



Each element's ID (coordinates) determines which data to work on

Per Kernel Computation Partitioning

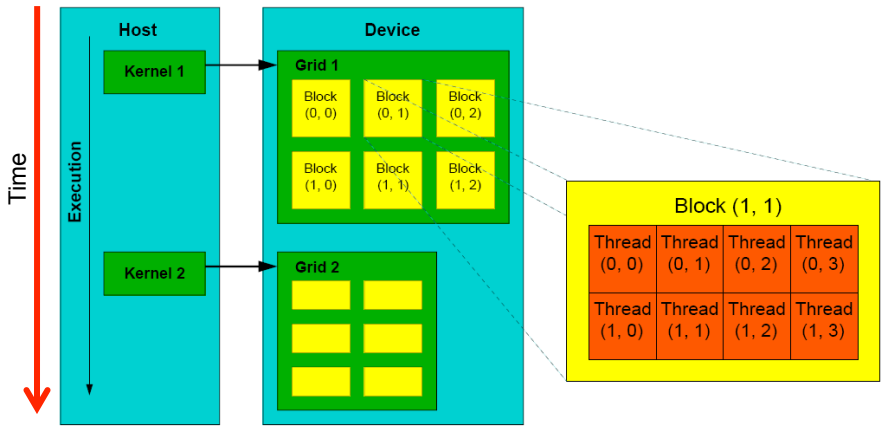
- Computation Grid Example: 2D Case



- Thread-block scheduled in one streaming multiprocessor (**SM**)
  - Once scheduled, cannot be pre-empted (must finish execution)
  - Thread-blocks execute same program (SPMD)
  - Threads within a thread-block organized into warps (32 SIMD threads)
- Threads use their ID to determine which data to work on

GBT: Grids of Blocks of Threads

Programmer's view of data and computation partitioning



- Threads within a block can communicate/synchronize
  Run on the same streaming multiprocessor (SM)
- Threads across blocks cannot communicate
  Shouldn't touch each other's data (behavior undefined)
  Threads use their ID to determine which data to work on

Execution Model: Ordering

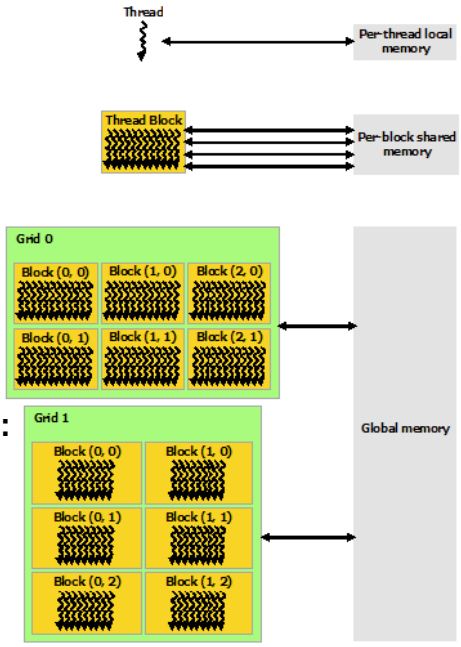
- **Execution order is undefined**
- Do not assume nor use:
  - block 0 executes before block 1
  - Thread 10 executes before thread 20
  - And **any** other ordering even if you can observe it
  - Future implementations may break this ordering
  - It's not part of the CUDA definition
  - Why? More flexible hardware options

Execution Model Summary (for your reference)

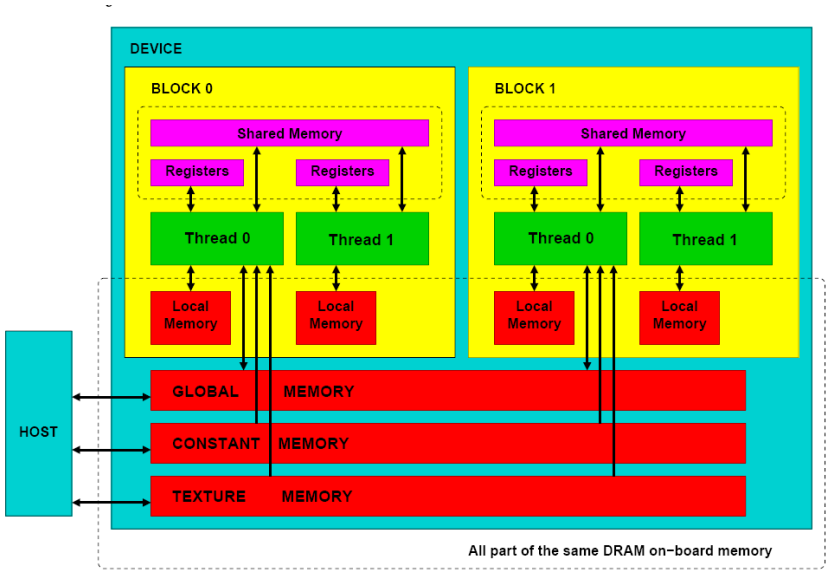
- **Grid of blocks of threads**
  - 1D/2D/3D grid of blocks, 1D/2D/3D blocks of threads
- **All blocks are identical:**
  - same structure and # of threads
  - Blocks are SPMD, warps within blocks are SIMD
- **Block execution order is undefined**
- Same block threads:
  - can synchronize and share data fast (via global or shared memory)
- Threads from different blocks:
  - Cannot cooperate
  - Communication through global memory
- Blocks do not migrate: execute on the same SM
- Several blocks may run over the same SM
- **Threads and Blocks have IDs**
  - Simplifies data indexing
  - Can be 1D, 2D, or 3D (threads)

Memory Hierarchy

- **Per-thread memory:**
  - Registers
  - Local memory
- **Per-block memory:**
  - Shared memory
- **Per-grid memory:**
  - Global memory
  - Constant memory (R/O)
  - Texture memory (R/O)
- **Host accessing GPU memory:**
  - Global memory
  - Constant memory
  - Texture memory



Programmer's view: Memory Model



Arrows show whether read and/or write is possible

Memory Model Summary

Memory	Location	Cached	Access	Scope
Local	off-chip	No	R/W	thread
Shared	on-chip	N/A	R/W	all threads in a block
Global	off-chip	No	R/W	all threads + host
Constant	off-chip	Yes	RO	all threads + host
Texture	off-chip	Yes	RO	all threads + host

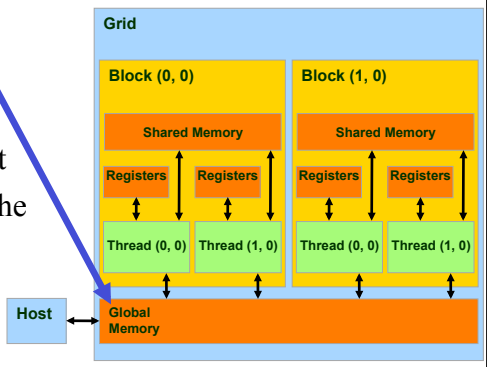
The host can R/W:
global, constant, and texture memories

CUDA API Highlights: Easy and Lightweight

- The API is an extension to the ANSI C programming language
  - → Low learning curve
- The hardware is designed to enable lightweight runtime and driver
  - → High performance

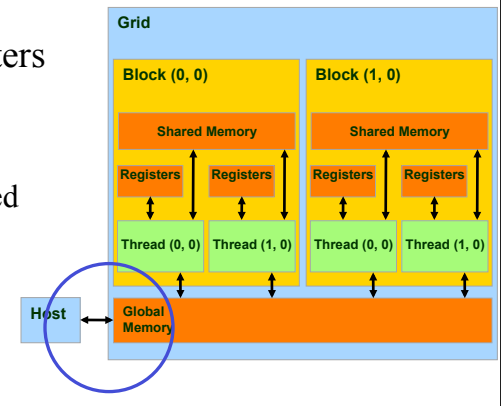
CUDA Device Memory Allocation

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



CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Asynchronous transfer



CUDA API: Example

```
int a[N];
   for (i =0; i < N; i++)
     a[i] = a[i] + x;
```

1. Allocate CPU Data Structure
2. Initialize Data on CPU
3. Allocate GPU Data Structure
4. Copy Data from CPU to GPU
5. Define *Execution Configuration*
6. Run Kernel
7. CPU synchronizes with GPU
8. Copy Data from GPU to CPU
9. De-allocate GPU and CPU memory

My first CUDA Program / Skeleton

GPU

```
__global__ void arradd (float *a, float f, int N)
 {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) a[i] = a[i] + f;
}
```

CPU

```
int main()
{
  float h_a[N]; /* allocate cpu container */
  for (int i=0; i < N; i++) h_a[i] = (float) i; /* initialize */

  float *d_a;
  cudaMalloc ((void **) &d_a, SIZE);

  cudaMemcpy (d_a, h_a, SIZE, cudaMemcpyHostToDevice));

  arradd <<< n_blocks, block_size >>> (d_a, 10.0, N);

  cudaThreadSynchronize ();
  cudaMemcpy (h_a, d_a, SIZE, cudaMemcpyDeviceToHost));
  CUDA_SAFE_CALL (cudaFree (a_d));
}
```

1. Allocate CPU Data container

```
float *ha;

main (int argc, char *argv[])
{
  int N = atoi (argv[1]);
  ha = (float *) malloc (sizeof (float) * N);

  ...
}
```

No memory allocated on the GPU side

- Pinned memory allocation results in faster CPU to/from GPU copies
- But pinned memory cannot be paged-out
  - `cudaMallocHost (…)`

2. Initialize CPU Data

```
float *ha;

int i;

for (i = 0; i < N; i++)
  ha[i] = i;
```

3. Allocate GPU Data container

```
float *da;

cudaMalloc ((void **) &da, sizeof (float) * N);
```

- Notice: no assignment side
  - NOT: da = cudaMalloc (…)
- Assignment is done internally:
  - That's why we pass &da
- Space is allocated in **Global Memory** on the GPU

GPU Memory Allocation

- The host manages GPU memory allocation:
  - **`cudaMalloc (void **ptr, size_t nbytes)`**
  - Must explicitly cast to (`void **`)
    - `cudaMalloc ((void **) &da, sizeof (float) * N);`
  - **`cudaFree (void *ptr);`**
    - `cudaFree (da);`
  - **`cudaMemset (void *ptr, int value, size_t nbytes);`**
    - `cudaMemset (da, 0, N * sizeof (int));`
- Check the **CUDA Reference Manual**

4. Copy Initialized CPU data to GPU

```
float *da;
float *ha;

cudaMemCpy ((void *) da,                    // DESTINATION
            (void *) ha,                    // SOURCE
            sizeof (float) * N,             // #bytes
            cudaMemcpyHostToDevice);        // DIRECTION
```

Host/Device Data Transfers

- The host initiates all transfers:
- **cudaMemcpy**(void *dst, void *src,
  size_t nbytes,
  enum cudaMemcpyKind direction)

- enum cudaMemcpyKind
  - cudaMemcpy**Host**To**Device**
  - cudaMemcpy**Device**To**Host**
  - cudaMemcpy**Device**To**Device**

5. Define Execution Configuration

- How many blocks and threads/block

```
int threads_block = 64;
int blocks = N / threads_block;
if (blocks % N != 0)
    blocks += 1;
```

- Alternatively:

```
int threads_block = 64;
int blocks = (N + threads_block - 1) /
             threads_block;
```

6. Launch Kernel & 7. CPU/GPU Synchronization

```
darradd <<<blocks, threads_block>>>(da, 10f, N);
cudaThreadSynchronize (); // forces CPU to wait
```

- <<<…>>> execution configuration
  - Instructs the GPU to launch `blocks x threads_block` threads
  - Each thread executes the kernel function darradd(da, x, N)
- darradd: kernel name
  - this is the C procedure each thread executes
- (da, x, N): arguments to the procedure darradd()
  - No variable arguments

Launch a Kernel with Multidimensional Blocks

- A kernel function must be called with an execution configuration:

```
dim3   DimGrid(100, 50);    // 5000 thread blocks
dim3   DimBlock(4, 8, 8);   // 256 threads per block
KernelFunc<<< DimGrid, DimBlock >>>(...args...);
```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking

CPU/GPU Synchronization

- CPU does not block on kernel execution
  - cudaMemcpy() to/from host is synchronous
  - Kernel requests are queued and processed in-order
  - Control returns to CPU immediately
- Good if there is other work to be done
  - e.g., preparing for the next kernel invocation
- Eventually, CPU must know when GPU is done
- Then it can safely copy the GPU results
- `cudaThreadSynchronize ()`
  - Block CPU until **all** preceding cuda…() and kernel requests have completed

8. Copy data from GPU to CPU & 9. DeAllocate Memory

```
float *da;
float *ha;

cudaMemCpy ((void *) ha,                    // DESTINATION
            (void *) da,                    // SOURCE
            sizeof (float) * N,             // #bytes
            cudaMemcpyDeviceToHost);        // DIRECTION

cudaFree (da);
// display or process results here
free (ha);
```

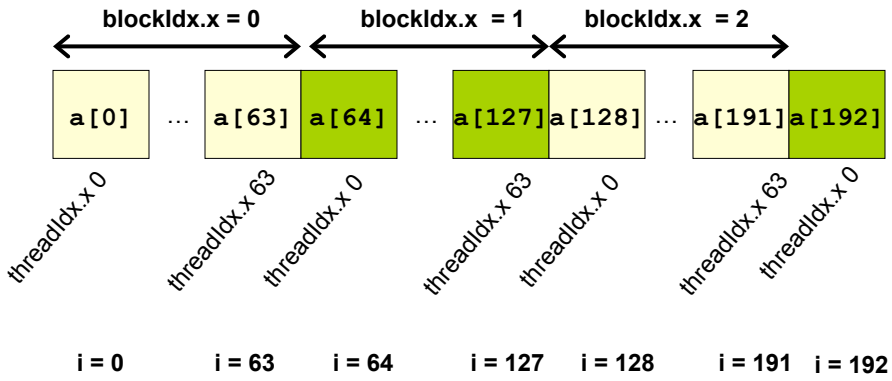
The GPU Kernel

```
__global__ darradd (float *da, float x, int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < N) da[i] = da[i] + x;
}
```

- **BlockIdx:** Unique Block ID.
  - Numerically asceding: 0, 1, …
- **BlockDim:** Dimensions of Block = how many threads it has
  - BlockDim.x, BlockDim.y, BlockDim.z
  - Unused dimensions default to 0
- **ThreadIdx:** Unique per Block Index
  - 0, 1, …
  - Per Block

Array Index Calculation Example

```
int i = blockIdx.x * blockDim.x + threadIdx.x;
```



Assuming blockDim.x = 64

Generic Unique Thread and Block Index Calculations #1

- **1D Grid / 1D Blocks:**

```
UniqueBlockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x +
threadIdx.x;
```

- **1D Grid / 2D Blocks:**

```
UniqueBlockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y
+ threadIdx.y * blockDim.x + threadIdx.x;
```

- **1D Grid / 3D Blocks:**

```
UniqueBockIndex = blockIdx.x;
UniqueThreadIndex = blockIdx.x * blockDim.x * blockDim.y
* blockDim.z + threadIdx.z * blockDim.y * blockDim.x +
threadIdx.y * blockDim.x + threadIdx.x;
```

- Source: http://forums.nvidia.com/lofiversion/index.php?t82040.html

Generic Unique Thread and Block Index Calculations #2

- **2D Grid / 1D Blocks:**

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.x +
threadIdx.x;
```

- **2D Grid / 2D Blocks:**

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex =UniqueBlockIndex * blockDim.y *
blockDim.x + threadIdx.y * blockDim.x + threadIdx.x;
```

- **2D Grid / 3D Blocks:**

```
UniqueBlockIndex = blockIdx.y * gridDim.x + blockIdx.x;
UniqueThreadIndex = UniqueBlockIndex * blockDim.z *
blockDim.y * blockDim.x + threadIdx.z * blockDim.y *
blockDim.z + threadIdx.y * blockDim.x + threadIdx.x;
```

- **UniqueThreadIndex means unique per grid.**

CUDA Function Declarations

	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
  - Can only call __device__ functions
- __device__ and __host__ can be used together
  - Both versions of the code generated

__device__ Example

- Add x to a[i] multiple times

```
__device__ float addmany (float a, float b, int count)
{
  while (count--) a += b;
  return a;
}

__global__ darradd (float *da, float x, int N)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if (i < N) da[i] = addmany (da[i], x, 10);
}
```

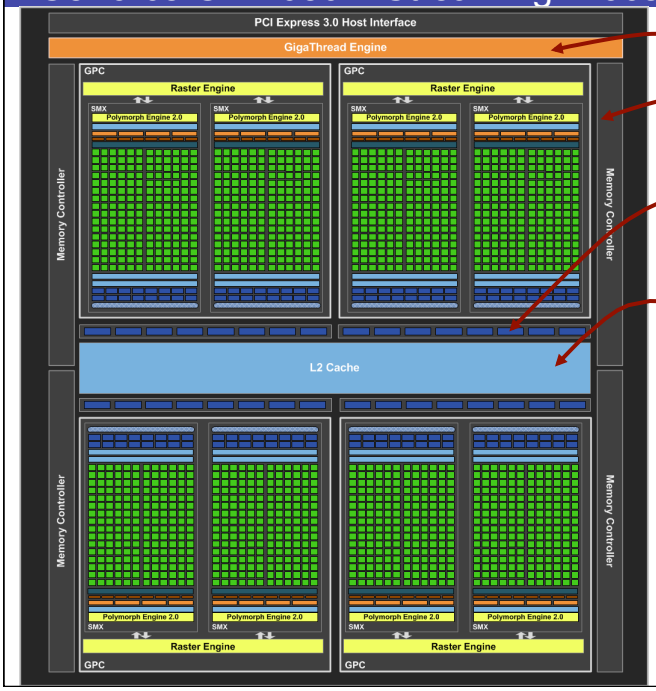
Kernel and Device Function Restrictions

- **__device__** functions cannot have their address taken
  - e.g., `f = &addmany; *f(…);`
- For functions executed on the device:
  - No recursion before compute capability 2.x
    - `darradd (…)`
      `{`
      `darradd (…)`
      `}`
    - **Supported for __device__ on Fermi (2.x capability)**
    - **Added support for __global__ on Kepler (3.x capability)**
  - No static variable declarations inside the function
    - `darradd (…)`
      `{`
      `static int canthavethis;`
      `}`
  - No variable number of arguments
    - e.g., something like `printf (…)`

Execution model guarantees

- Only that threads will execute
- Says nothing about the order
- Extreme cases:
  - #1: All threads run in parallel
  - #2: All threads run sequentially
    - Interleaving at synchronization points
- This is why the same CUDA program will run:
  - On the CPU
  - On a GPU with 1 unit
  - On a GPU with N units
    - Different models/price points

GeForce GTX 680 – Streaming Processor Array (SPA)



Block scheduling

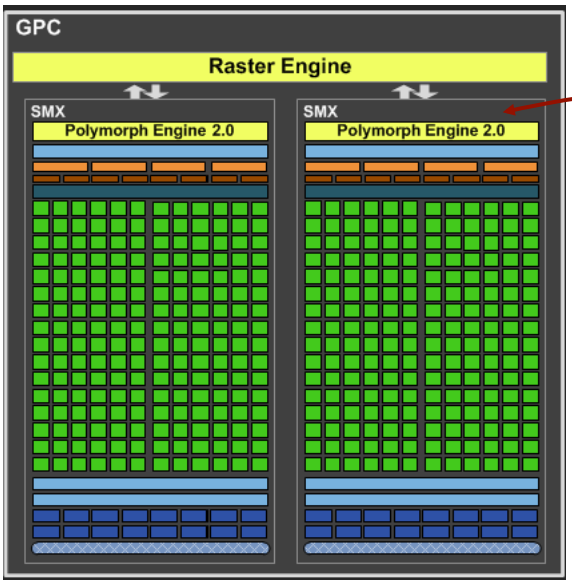
Memory access

ROP units (blending, Z-buffering, antialiasing)

512KB L2 cache

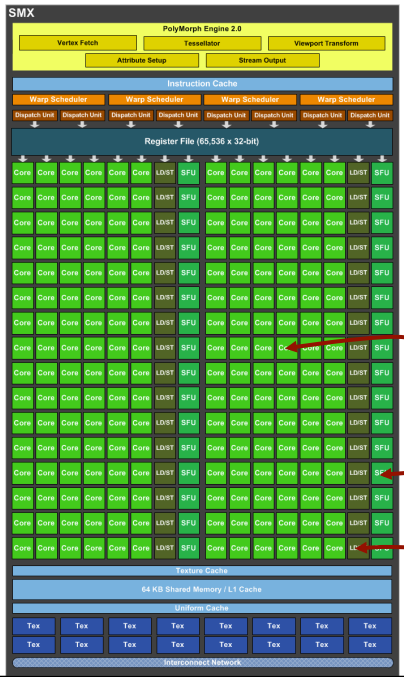
- 3.54bn transistors
- 1536 CUDA cores
- Core: 1.006GHz
  - Up to 1.1GHz
- Mem: 6.008GHz
- 3.09 TFLOPs
- 195W

GeForce GTX 680 – Graphics Processing Cluster



Streaming Multiprocessor (a.k.a. SM, SMX)

GeForce GTX 680 – Streaming Multiprocessor



- **SM (a.k.a. SMX, SMP)**
  - Streaming Multiprocessor
  - Multi-threaded processor
    - 192 CUDA cores
    - 1 to 1024 threads active
  - Shared instruction fetch per 32 threads (warp)
  - Fundamental processing unit for CUDA thread block
- **SP (a.k.a. CUDA core)**
  - Streaming Processor
  - Scalar ALU for a single CUDA thread
- **SFU**
  - Special function unit
- **LDST**
  - Memory access unit

Execution Summary

- Break data into Blocks (grid)
- Break Blocks into Warps
- Each Warp == 32 threads
  - Warp size is not part of the CUDA specification
  - It has always been 32 so far
- Allocate Resources
  - Registers, Shared Memory, Barriers
- Then allocate for execution
  - Blocks execute in SPMD
  - Warps execute in SIMD
  - A Warp is the smallest schedulable unit

My first CUDA Program / Skeleton

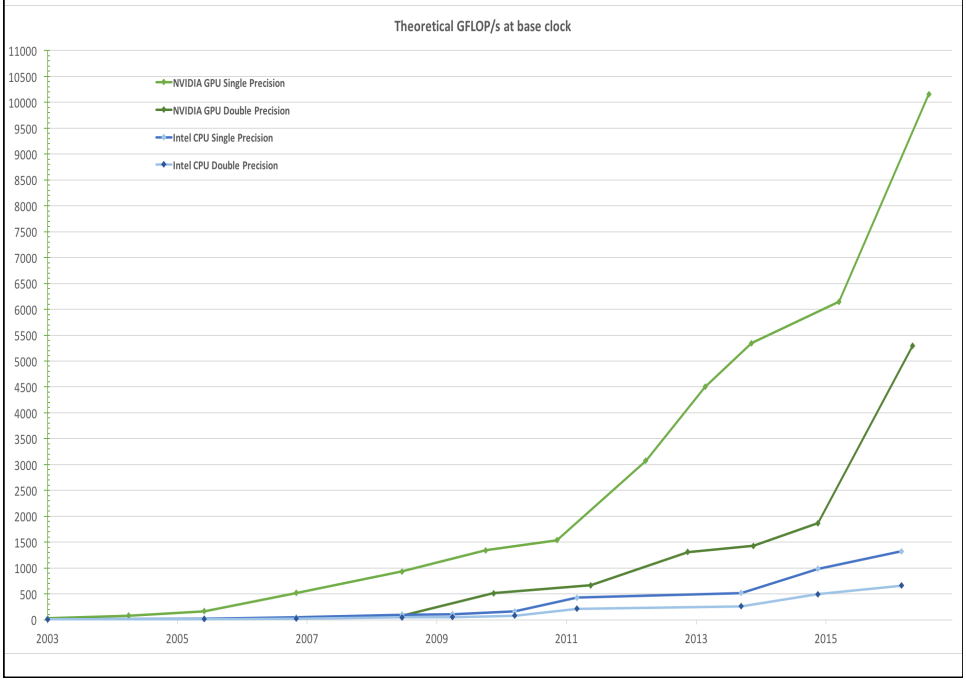
```
__global__ void arradd (float *a, float f, int N)
 {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < N) a[i] = a[i] + f;
}
```

GPU

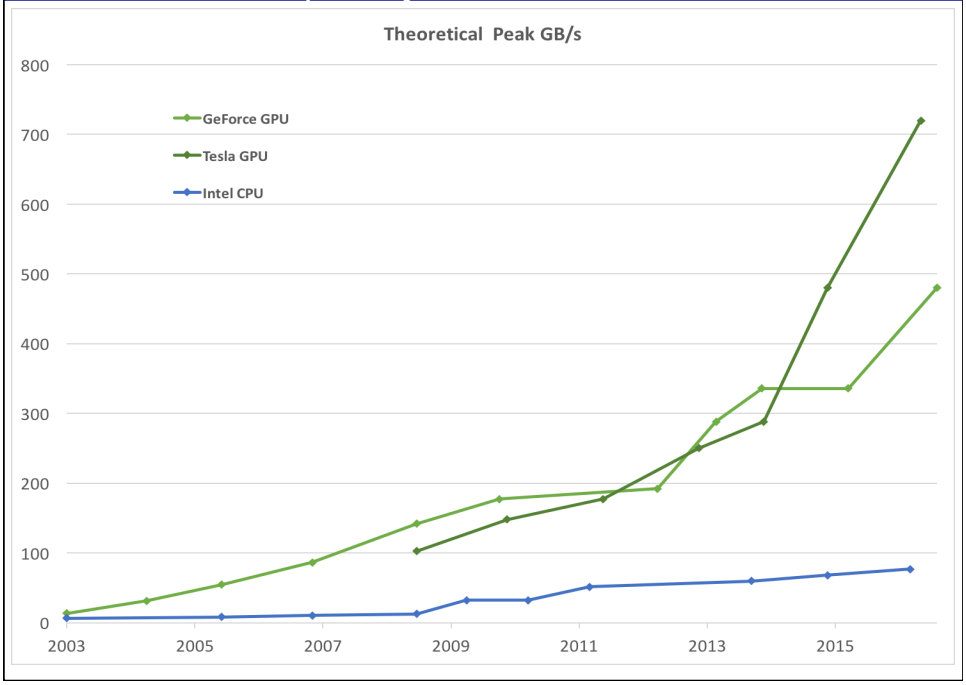
```
int main()
{
  float h_a[N]; /* allocate cpu container */
  for (int i=0; i < N; i++) h_a[i] = (float) i; /* initialize */

  float *d_a;
  cudaMalloc ((void **) &d_a, SIZE);

  cudaMemcpy (d_a, h_a, SIZE, cudaMemcpyHostToDevice));

  arradd <<< n_blocks, block_size >>> (d_a, 10.0, N);

  cudaThreadSynchronize ();
  cudaMemcpy (h_a, d_a, SIZE, cudaMemcpyDeviceToHost));
  CUDA_SAFE_CALL (cudaFree (a_d));
}
```

CPU

Data Processing Capability



Data Access Capability



Final Remarks

- Easy to write a CUDA program that works
- Difficult to squeeze every out ounce of performance
  - Thread divergence (control statements, e.g., if-then-else)
  - Global memory accesses (are they coalesced?)
  - Shared memory accesses (bank conflicts?)
  - Device occupancy
    - Limited hardware resources constrain #threads to execute
    - Wastes resources if GPU not used fully
    - User has knobs to control occupancy
      - #blocks, #thread per block, #regs per thread, #shared memory
  - Synchronization
  - Algorithmic modifications (e.g., thread assignment)
  - If done properly: **50-150x improvements are common**

For more details: EECS 368/468

http://docs.nvidia.com/cuda/cuda-c-programming-guide/