GPU minicourse

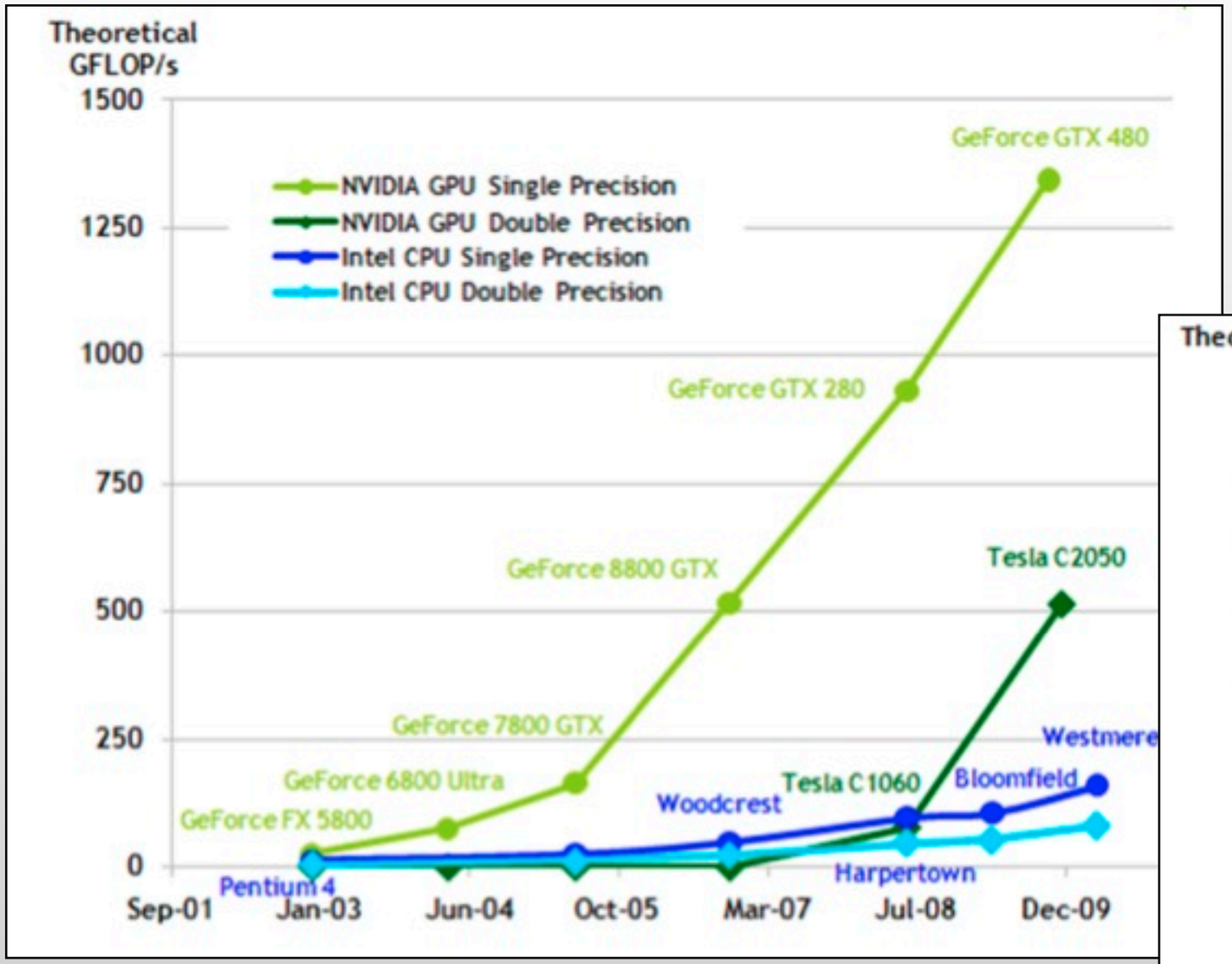
Jonathan Dursi & Harald Pfeiffer

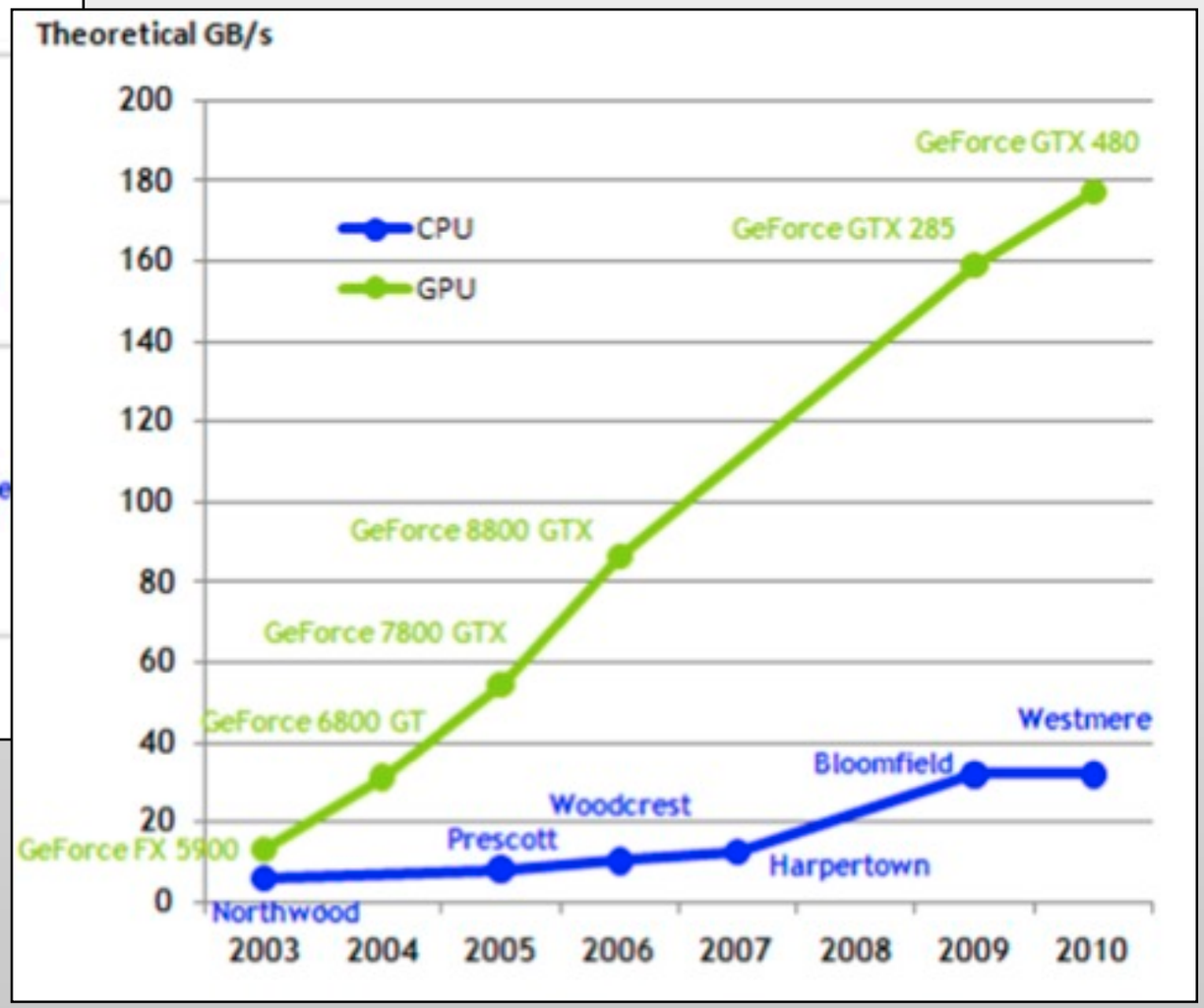
Fall 2012

Welcome!

CITA–ICAT

- Goal: PRACTICAL skills
- Outline
  - Today: general overview (HP)
  - Oct 19 -- More details on CUDA (JD)
  - Oct 26 -- CUDA devel tools: SDK examples, debugger, libraries, profiler (HP)
  - Nov 2 -- Memory access & coalescing (JD)
  - Nov 9 -- Occupancy & latency (HP)
  - Nov 16 -- Introduction to OpenCL (JD)
  - Nov 23 -- Using Multiple GPUs





(CUDA C Programming Guide)

Short history

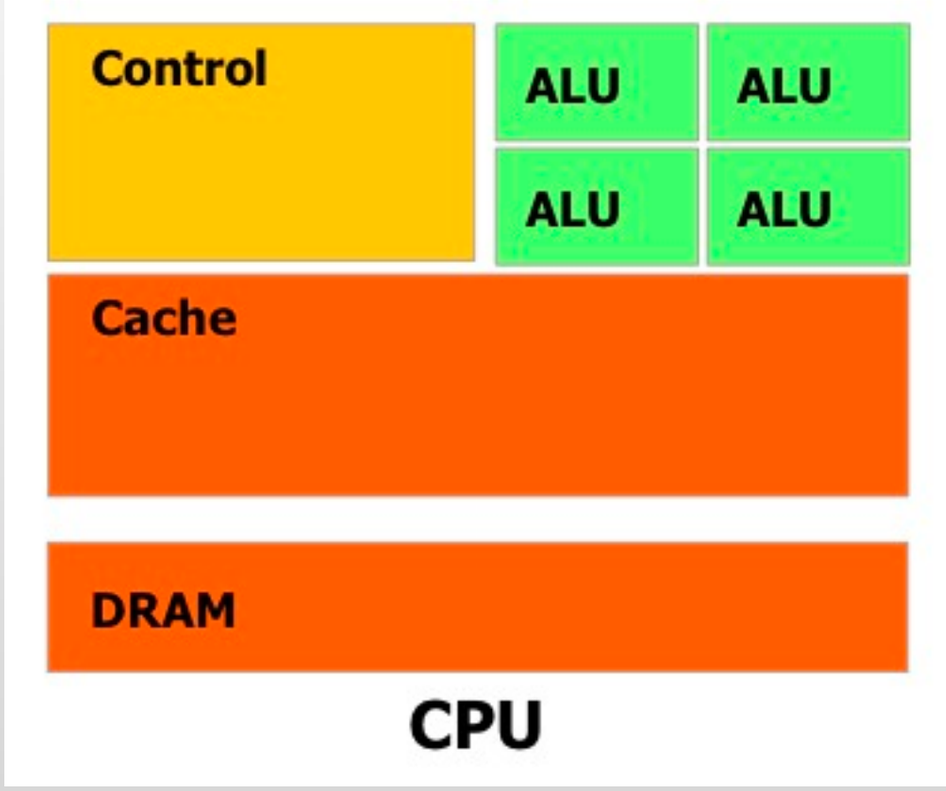
CITA–ICAT

- 1980s, 1990s:
  - fixed-function graphics processing pipelines
  - development of APIs (e.g. OpenGL) to use graphics cards
- early 2000's (GeForce 3, Radeon 9700)
  - make vertex (and later shader) somewhat programmable
  - Data independence encourages many-core design
- late 2000's
  - GPUs more easily programmable
  - CUDA C/C++ compiler and OpenCL to ease programming
- 2010 Fermi
  - high double-precision performance
- 2012/13 Kepler, Intel MIC

Design choices & implications

CITA–ICAT

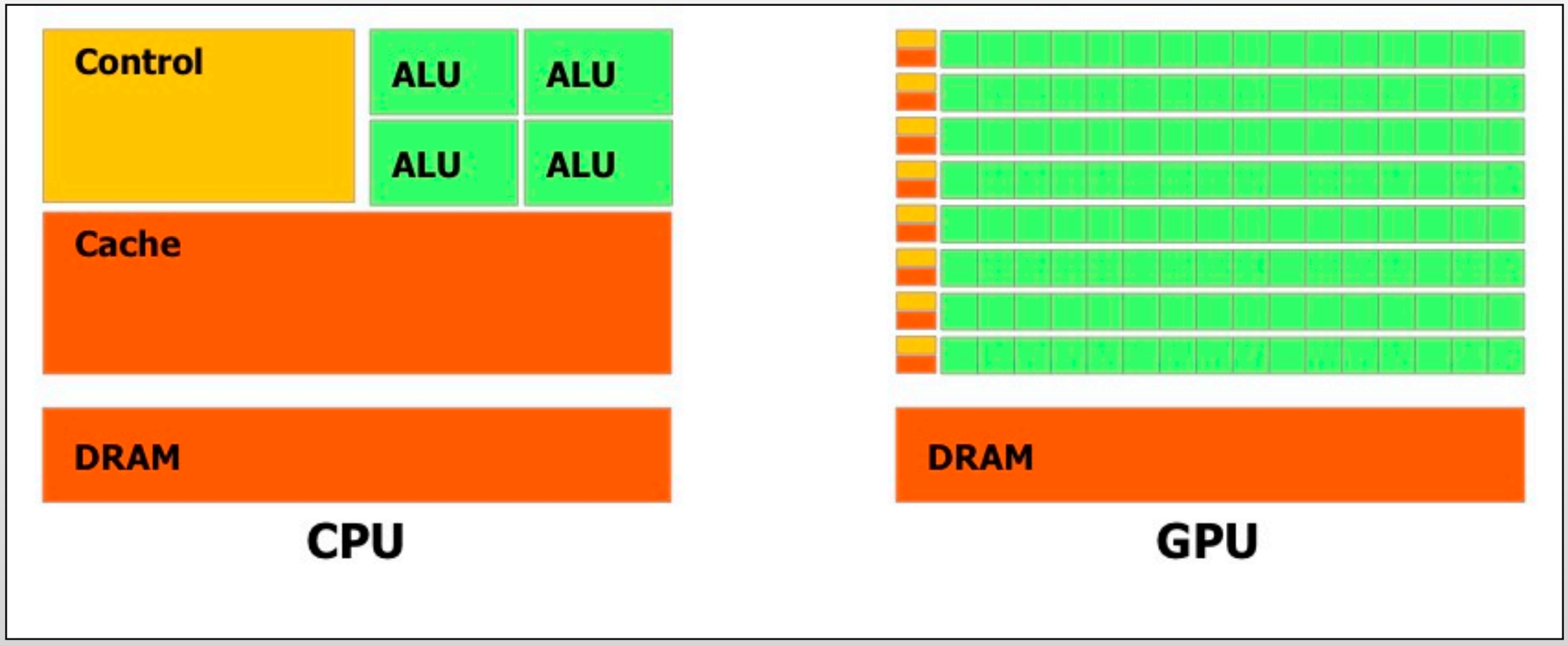
- CPU
  - flexible
  - execution of single thread fast (whatever instructions might come along)
  - large cache, sophisticated control unit
  - very few threads



GPU

CITA–ICAT

- Maximize fraction of chip dedicated to floating point units

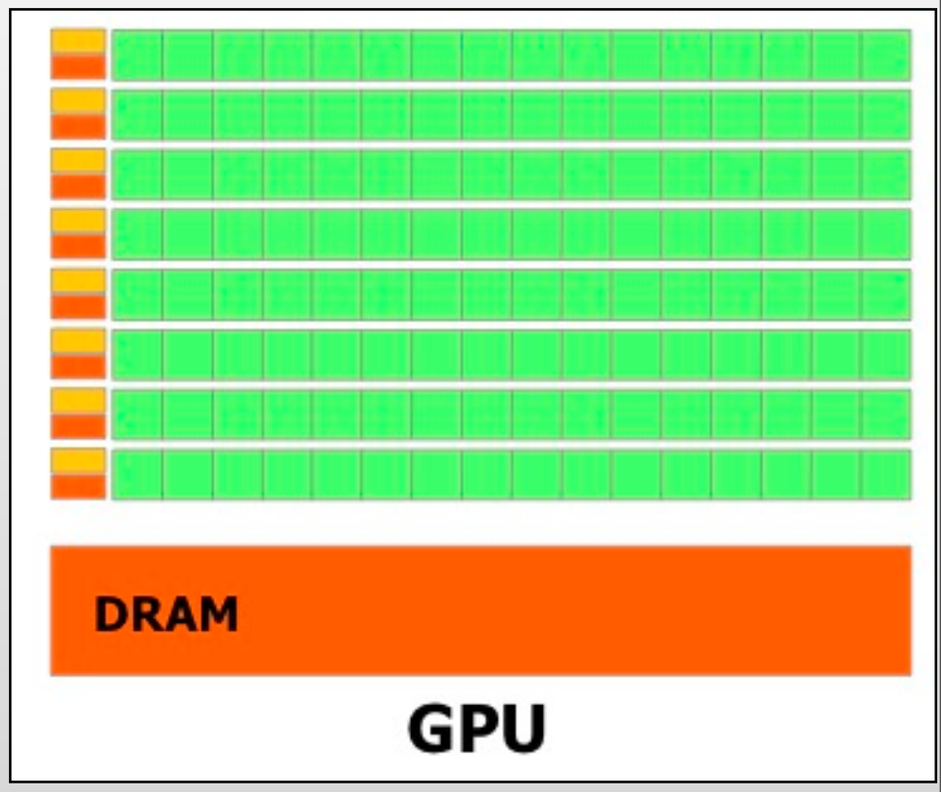


- Key design decision: Tuned for ***massively parallel*** programs with ***regular*** execution patterns.
- GPUs amazingly fast, if one works *with* their design decisions.

Consequences

CITA–ICAT

- Tuned for *massively parallel* programs with *regular* execution patterns.
  - ~448 compute cores (~4 on CPU-cores)
  - packaged into ~14 *streaming multiprocessors* (SM, ~32cores/SM)
  - one control unit per SM
    - *threads* execute ~32-at-once (*warp*)
    - identical instructions or idle
  - small but fast cache
    - at most ~48KB per SM
    - each SM can only access its cache

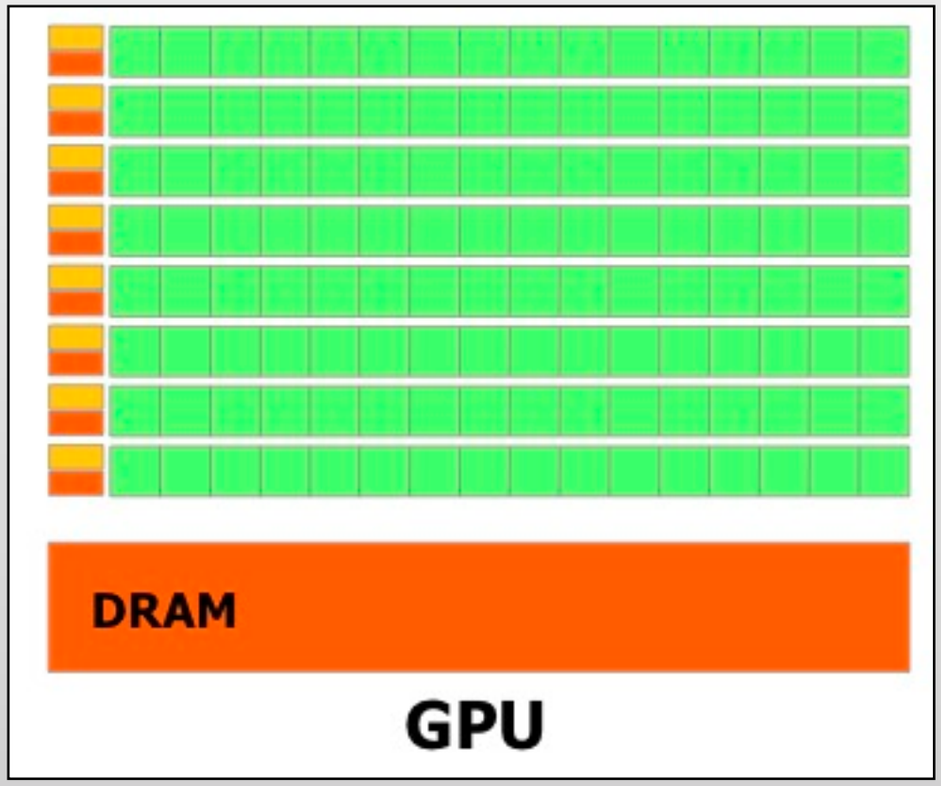


- "~NNN": numerical value depends on hardware (given specs for GTX470, M2050/M2070.)
- This slide just first taste - in-depth explanations in later lectures

Consequences cont'd

CITA–ICAT

- Tuned for *massively parallel* programs with *regular* execution patterns.
  - Data fetched in contiguous ~128byte blocks
    - regular data-access fast
    - random data-access inefficient
  - Starting new threads is slow, but switching between threads is fast
    - Run >>32 threads per SM
    - Threads start while other threads execute
  - Initiating memory-transfers from GPU-DRAM slow, but aggregate band-width large
    - Run >>32 threads per SM to hide latency (*occupancy*)



Literature

CITA–ICAT

- NVIDIA docs
  - http://developer.nvidia.com/category/zone/cuda-zone
  - http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation
    - CUDA C Programming Guide
    - CUDA C Best Practices Guide
- Further useful NVIDIA docs
  - CUDA Reference Manual
  - cuda-gdb user manual
- Books
  - Kirk+Hwu "Programming Massively Parallel Processors"
  - Sanders+Kandrot "CUDA by Example"

Code example (CPU version)

CITA–ICAT

```cpp
// Example1.cpp
// Compute v[i] = a*u[i] + b  on CPU
#include <cmath>     // for sin() and M_PI
#include <iostream>  // for output

void MultAdd(const int N, double* v, const double* u,
             const double a, const double b) {
  for(int idx=0; idx<N; ++idx)
    v[idx] = a*u[idx]+b;
}

int main() {
  // setup variables
  const int N=1000;
  double* u = new double[N];
  for(int i=0; i<N; ++i) u[i] = sin(2.*M_PI*i/N);
  double* v = new double[N];
  // do work
  MultAdd(N, v, u, 2.5, 1.2);
  // output and clean up
  std::cout << "CPU: v[10]=" << v[10] << std::endl;
  delete[] u;   delete[] v;
};
```

```
[pfeiffer@marten GpuMinicourse]$
[pfeiffer@marten GpuMinicourse]$
[pfeiffer@marten GpuMinicourse]$ g++ Example1.cpp
[pfeiffer@marten GpuMinicourse]$ ./a.out
CPU: v[10]=1.35698
[pfeiffer@marten GpuMinicourse]$
[pfeiffer@marten GpuMinicourse]$
[pfeiffer@marten GpuMinicourse]$
```

CUDA example: main()

```
// Exaple1.cu    -*- c++ -*-
// Compute v[i] = a*u[i] + b  with CPU and with CUDA
#include <cmath>     // for sin() and M_PI
#include <iostream>  // for output
// as before
void MultAdd(const int N, double* v, const double* u,
             const double a, const double b) {
  for(int idx=0; idx<N; ++idx)
    v[idx] = a*u[idx]+b;
}
// to be discussed later
void MultAddCuda(const int N, double* v, const double* u,
                 const double a, const double b);

int main() {
  // setup variables
  const int N=1000;
  double* u = new double[N];
  for(int i=0; i<N; ++i) u[i] = sin(2.*M_PI*i/N);
  double* v = new double[N];
  double* w = new double[N];
  // do work (twice)
  MultAdd(N, v, u, 2.5, 1.2);
  MultAddCuda(N, w, u, 2.5, 1.2);
  // output and clean up
  std::cout << "CPU: v[10]=" << v[10] << ", GPU: w[10]=" << w[10]
            << ", difference " << v[10]-w[10] << std::endl;
  delete[] u;   delete[] v;   delete[] w;
};
```

CUDA example: MultAddCuda(...)

CITA–ICAT

```
// compute v[i] = a*u[i]+b  using CUDA
void MultAddCuda(const int N, double* v, const double* u,
                 const double a, const double b) {
  // (1)  allocate memory on device (i.e. GPU)
  double *vD, *uD;  //  (D = device)
  cudaMalloc((void**)&vD, N*sizeof(double));
  cudaMalloc((void**)&uD, N*sizeof(double));

  // (2)  copy input data onto device
  cudaMemcpy(uD, u, N*sizeof(double), cudaMemcpyHostToDevice);

  // (3) compute on device
  MultAddKernel<<<1, N>>>(N, vD, uD, a, b);  // to be discussed later

  // (4) copy results to host (i.e. CPU RAM)
  cudaMemcpy(v, vD, N*sizeof(double), cudaMemcpyDeviceToHost);

  // (5) clean up
  cudaFree(uD);
  cudaFree(vD);
}
```

- GPUs have separate RAM
- User handles memory allocation & copy

CUDA example: Kernel

CITA–ICAT

❖ CUDA *kernel*

```
__global__ void MultAddKernel(const int N, double* vD, const double* uD,
                              const double a, const double b) {
  int idx=threadIdx.x;
  vD[idx] = a*uD[idx]+b;
};
```

- Each call to MultAddKernel computes only one single element
- special variable "threadIdx" tells us which element
- Launch as many kernel-threads as there are elements

❖ *Launch* kernel:

```
// (3) compute on device
MultAddKernel<<<1, N>>>(N, vD, uD, a, b);
```

- <<<1, N>>> determines how many threads are launched (here N)
- the N threads will be processed in parallel

SciNet ARC Cluster

CITA–ICAT

- If you already have a SciNet account
  - email support@scinet to request access to the ARC Cluster
- Otherwise:
  - Get a temporary account today
  - https://support.scinet.utoronto.ca/wiki/index.php/GPU_Devel_Nodes

CITA GPU environment

- GPU machines
  - marten.cita.utoronto.ca 2x GTX470
  - bee.cita.utoronto.ca 1x Tesla C2050
    (used by Abdul, if he's logged in CUDA won't work)
  - tpb1, tpb2 on sunnyvale 2xTesla C1050 (each)
    (ssh ricky.cita.utoronto.ca, then ssh tpb{1,2})

- Use "modules" to configure environment

```
[pfeiffer@marten GpuMinicourse]$
[pfeiffer@marten GpuMinicourse]$ module load cuda
[pfeiffer@marten GpuMinicourse]$ module list
Currently Loaded Modulefiles:
  1) modules                      4) hdf5/1.8.5
  2) gcc/gcc-4.4.4                5) cuda/3.2
  3) openmpi/1.4.2-gcc-4.4.4
[pfeiffer@marten GpuMinicourse]$
[pfeiffer@marten GpuMinicourse]$
[pfeiffer@marten GpuMinicourse]$
```

Compiling + Running

CITA–ICAT

❖ nvcc = Nvidia CUDA compiler

```
[pfeiffer@marten GpuMinicourse]$
[pfeiffer@marten GpuMinicourse]$ nvcc -arch=sm_20 -o Example1  Example1.cu
[pfeiffer@marten GpuMinicourse]$
[pfeiffer@marten GpuMinicourse]$ ./Example1
CPU: v[10]=1.50603, GPU: w[10]=1.50603, difference 0
[pfeiffer@marten GpuMinicourse]$
[pfeiffer@marten GpuMinicourse]$
[pfeiffer@marten GpuMinicourse]$
```

- -arch=sm_20 chooses "*Compute Capability*" version 2.0
  - Compute capability 2.0 -- Fermi architecture (marten + bee)
  - Compute capability 1.3 -- Tesla architecture (tpb1, tpb2)

Critique (or: How the real world differs from the example)

CITA–ICAT

- of course, EVERYTHING will be different!
- Allocating/deallocating device-memory SLOW. Reuse memory
- Copy host-device and device-host SLOW. Minimize!
- In practice:
  - Copy data once to device
  - Do many calculations on device
  - Only copy end-result back

Critique cont'd

CITA–ICAT

- <<<1, N>>>MultAddKernel(...)
  - 2nd number (*block-size*) has MAXIMUM dependent on GPU hardware (1024 for Fermi, 512 for Tesla).
    - example fails for N above this maximum
  - 1st number gives number of independent blocks (*grid size*)
    - all threads within a block execute on the same SM (to allow access to shared memory).
    - With grid-size=1, only one SM will be used, and the other 13 will be idle.
- GPUs have excessive floating-point performance, relative to their memory bandwidth. Example does only two FLOP (one *, one +) per two doubles of data u[idx], v[idx].
  - Performance will be *bandwidth limited*

beyond NVIDIA/CUDA

CITA–ICAT

- AMD/ATI
  - GPU's reasonably similar to NVIDIA
  - separate GPU-memory, ~1000 compute-cores
  - Slightly less "general purpose" (e.g. may need to use 3-vectors for top-performance)
  - Program via OpenCL (slightly more cumbersome)

- Intel Phi (aka MIC -- Many Intel Cores)
  - Separate PCIe card, separate memory, >=50 "intel cores"
  - Program via standard intel compilers (makes coding look simple)
  - User must be aware of architecture constraints (e.g. memory transfer and layout) to get good speed-up.

beyond NVIDIA/CUDA

CITA–ICAT

❖ IBM BlueGene/Q

- Very basic/lightweight compute-nodes
- 16 cores/node, recommended to use 64 threads/node
- Each job should use 32*n nodes (i.e. 1024*n threads).

❖ Regular CPUs

- Last Intel generation: 4 cores/chip (typically 8cores/node)
- Current generation:
  - Intel Sandybridge 6, 8 cores/chip (typically 12, 16 cores/node)
  - AMD 12 cores/chip (typically 24 cores/node)

Power & Bandwidth

CITA–ICAT

- Power consumption rises *fast* with clock-speed
  - Best FLOPS/watts achieved for *many, slow* compute-cores

- Communication bandwidth rises *slower than FLOPS*

- You will face massively parallel, bandwidth constrained environments
  - *CUDA as good a learning ground as anything else*

Homework

CITA–ICAT

1. Code the example v[i] = a*u[i]+b
   1.1. Write code that computes this on GPU and on CPU, and compares the results.
   1.2. Run with increasing thread-counts N (<<<1,N>>>). How does CUDA tell you (or not) when things break?

2. Write a second CUDA program, evaluating a function of your choice.

3. By Thursday, 2pm:
   3.1. Email code, output and answer to question1.2. to HP & JD.