Introduction to GPUs

CS378 – Spring 2015

Sreepathi Pai

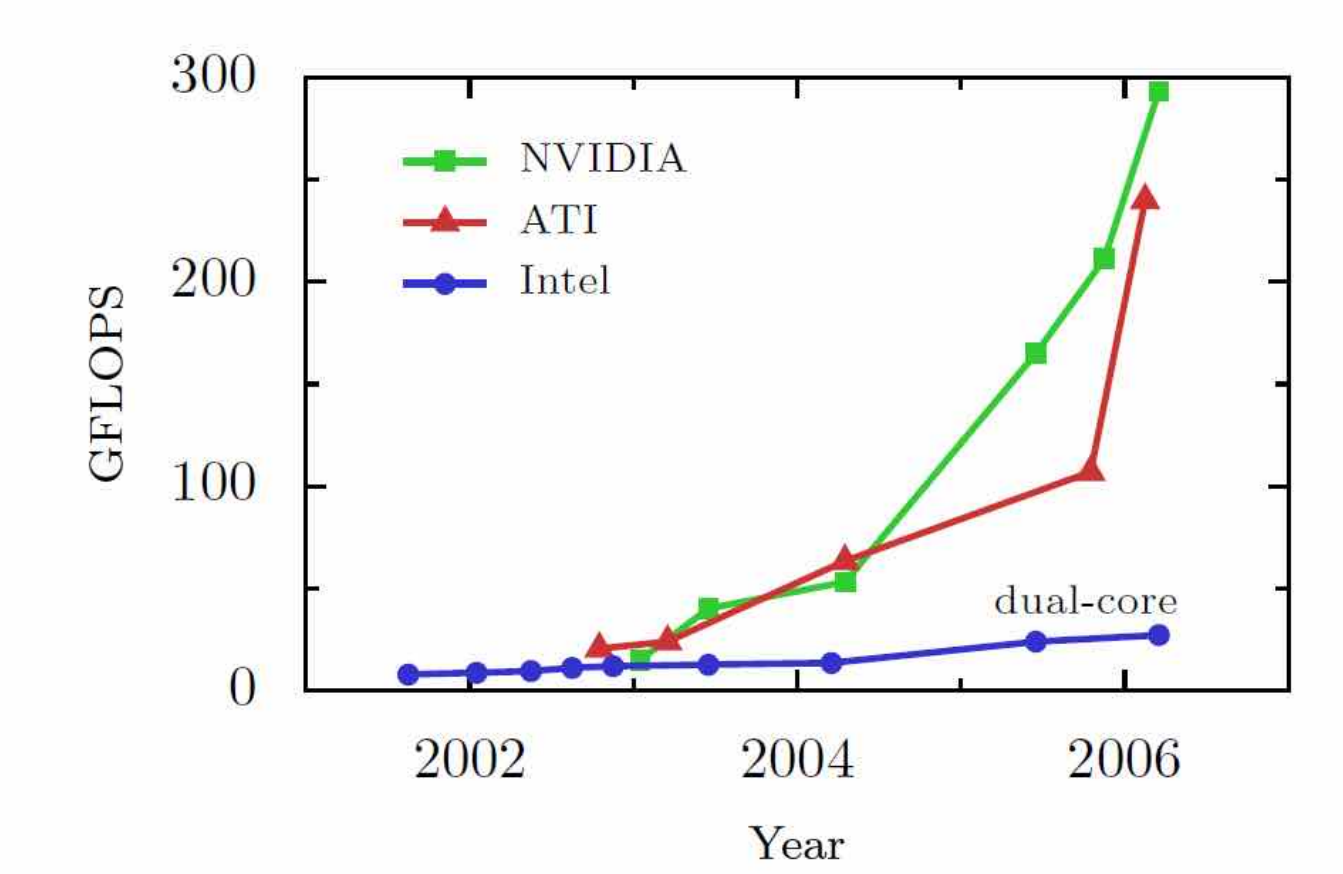
Outline

- Introduction to Accelerators
  - Specifically, GPUs
  - What can you use them for?
  - When should you consider using them?
- GPU Programming Models
  - How to write programs that use GPUs

Beyond the CPU

- Single-core CPU performance has essentially stagnated
- 10 years ago, the solution was *multicore* and increased parallelism
  - x86: 16 cores on chip, AMD Interlagos, 2011
- Applications need increased performance
  - Big data, brain simulation, scientific simulation, ...
- So if you're going to write parallel code, are there faster, viable alternatives to the CPU?

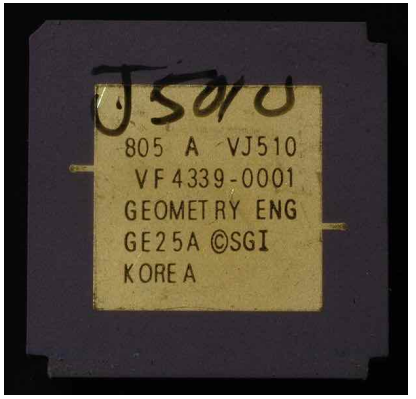
GPU Floating Point Performance

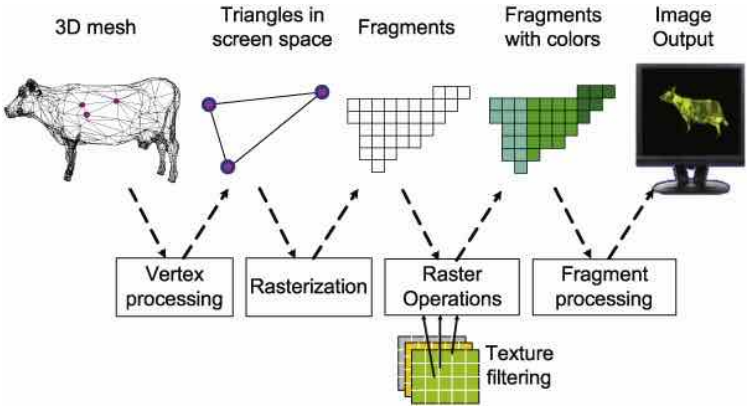


Owens et al., A Survey of General-Purpose Computation on Graphics Hardware

The rise of the GPU - 1990s

- The CPU has always been slow for Graphics Processing
  - Visualization
  - Games
- Graphics processing is inherently parallel and there is a lot of parallelism – O(*pixels*)
- GPUs were built to do graphics processing *only*
- Initially, hardwired logic replicated to provide parallelism
  - Little to no programmability



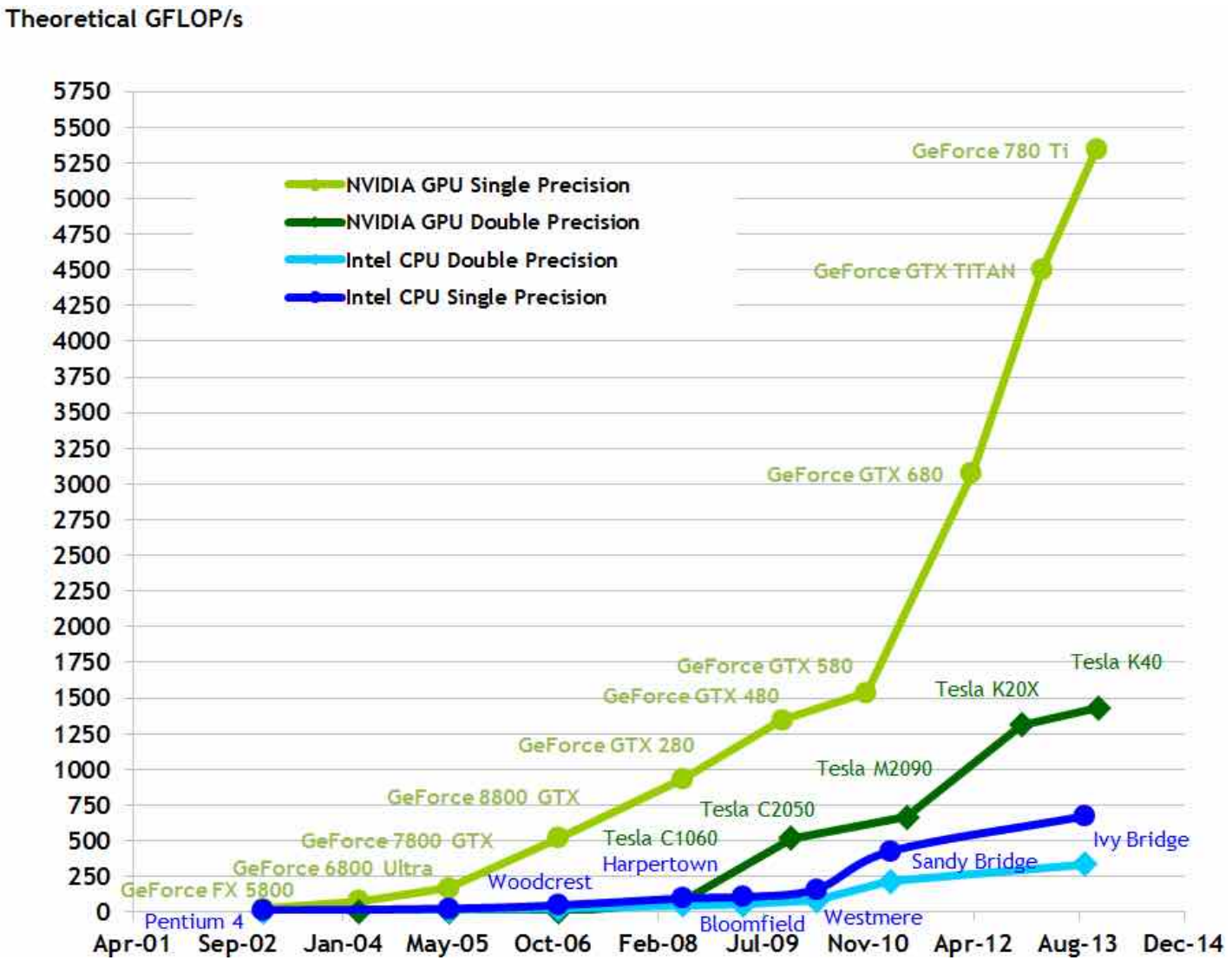


Wikipedia; Kaufman et al. (2009)

GPUs – a decade ago

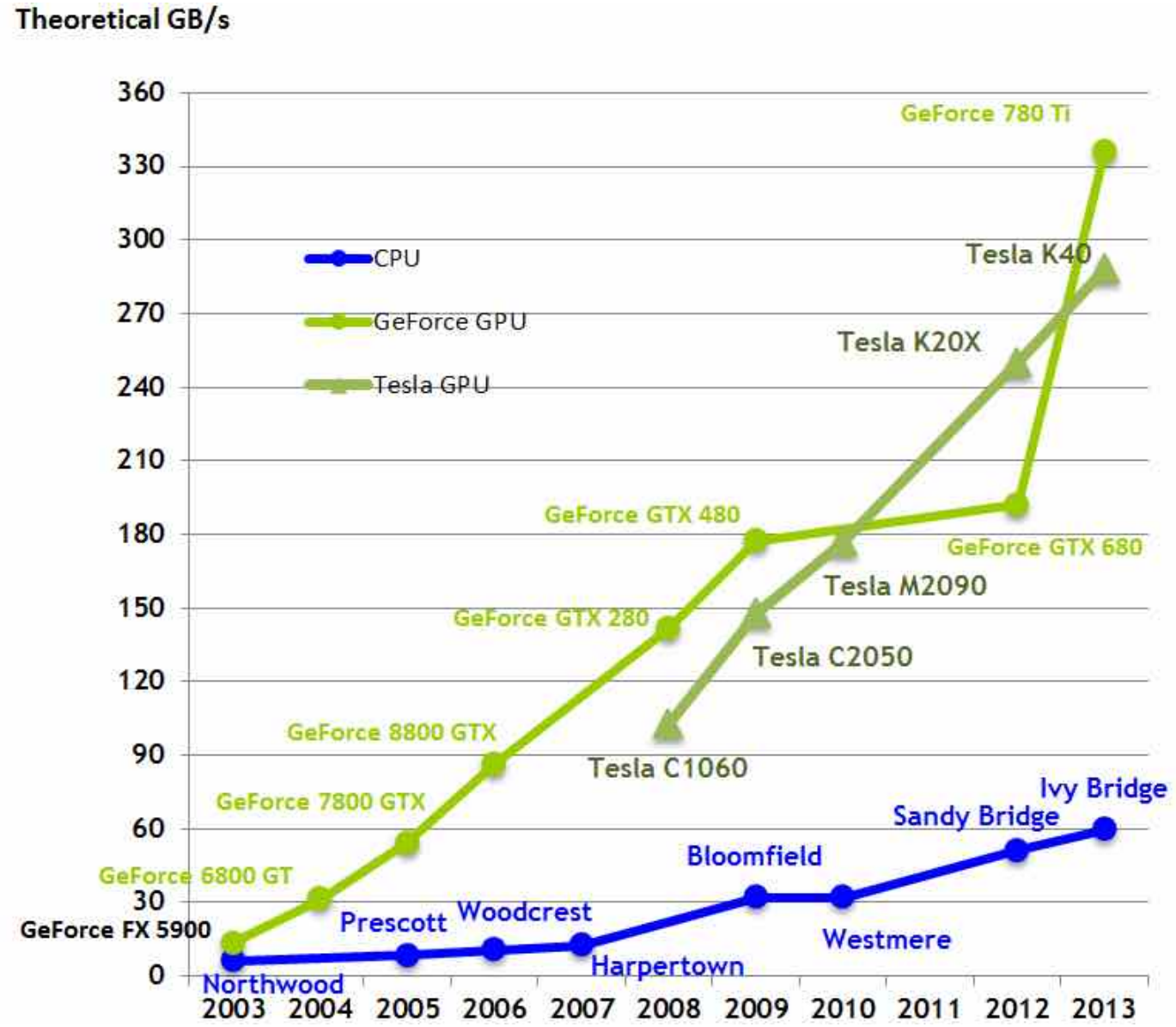
- Like CPUs, GPUs benefited from Moore's Law
- Evolved from fixed-function hardwired logic to flexible, programmable ALUs
- Around 2004, GPUs were programmable “enough” to do some non-graphics computations
  - Severely limited by graphics programming model (shader programming)
- In 2006, GPUs became “fully” programmable
  - NVIDIA releases “CUDA” language to write non-graphics programs that will run on GPUs

GPU Performance Today



NVIDIA, CUDA C Programming Guide

Memory Bandwidth



NVIDIA, CUDA C Programming Guide

GPUs today

- GPUs are widely deployed as accelerators
- Intel Paper
  - 10x vs 100x Myth
- GPUs so successful that other CPU alternatives are dead
  - Sony/IBM Cell BE
  - Clearspeed RSX
- Kepler K40 GPUs from NVIDIA have performance of 4TFlops (peak)
  - CM-5, #1 system in 1993 was ~60 Gflops (Linpack)
  - ASCI White (#1 2001) was 4.9 Tflops (Linpack)

Titan (#1 2012)

Tianhe 1A (#1 2010)

Top 500 Supercomputers

Accelerator-based Systems

- CPUs have always depended on *co-processors*
  - I/O co-processors to handle slow I/O
  - Math co-processors to speed up computation
- These have mostly been *transparent*
  - Drop in the co-processor and everything sped up
- The GPU is **not** a transparent accelerator for general purpose computations
  - Only graphics code is sped up transparently
- Code must be rewritten to target GPUs

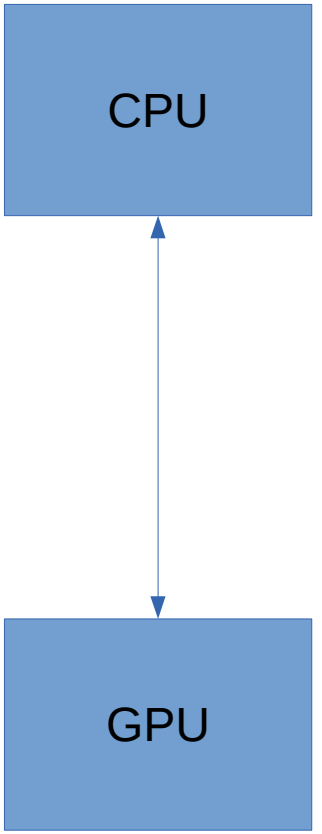
Using a GPU

1. You must retarget code for the GPU
   - rewrite, recompile, translate, etc.

The Two (Three?) Kinds of GPUs

Type 1: Discrete GPUs

Primary benefits: More computational power, more memory B/W

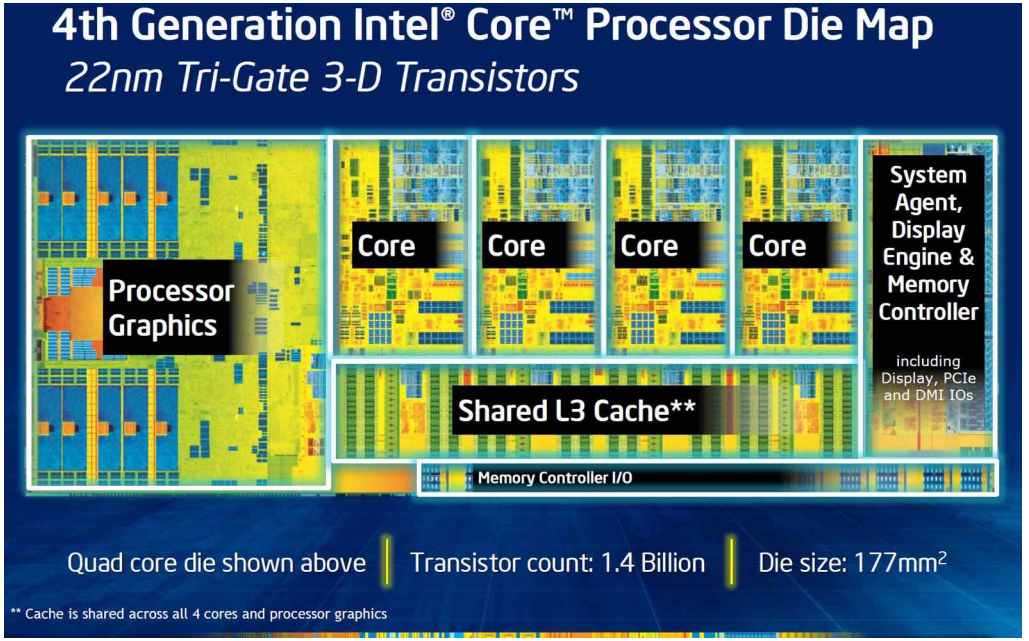


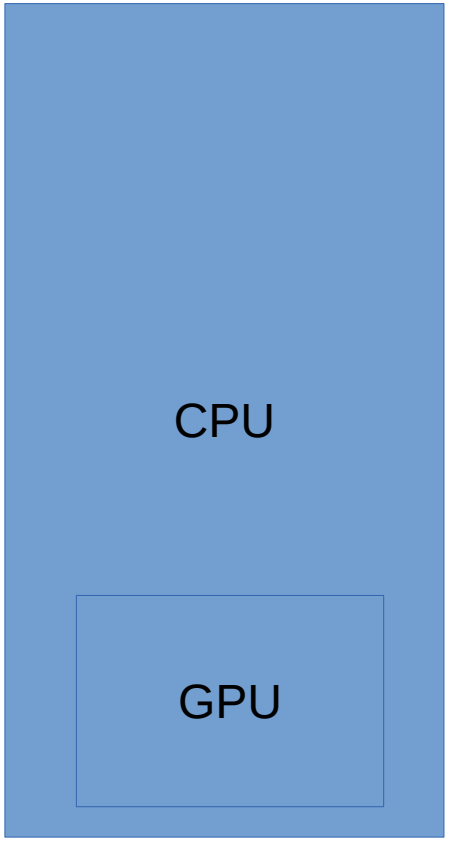
The Two (Three?) Kinds of GPUs

Type 2 and 3: Integrated GPUs

Primary distinction: Share system memory

Primary benefits: Less power (energy)





Intel

The NVIDIA Kepler

GPU RAM

GPU RAM

NVIDIA Kepler GK110 Whitepaper

Using a GPU

1. You must retarget code for the GPU
2. The working set must fit in GPU RAM
3. You must copy data to/from GPU RAM

NVIDIA Kepler SMX

2-way
In-order

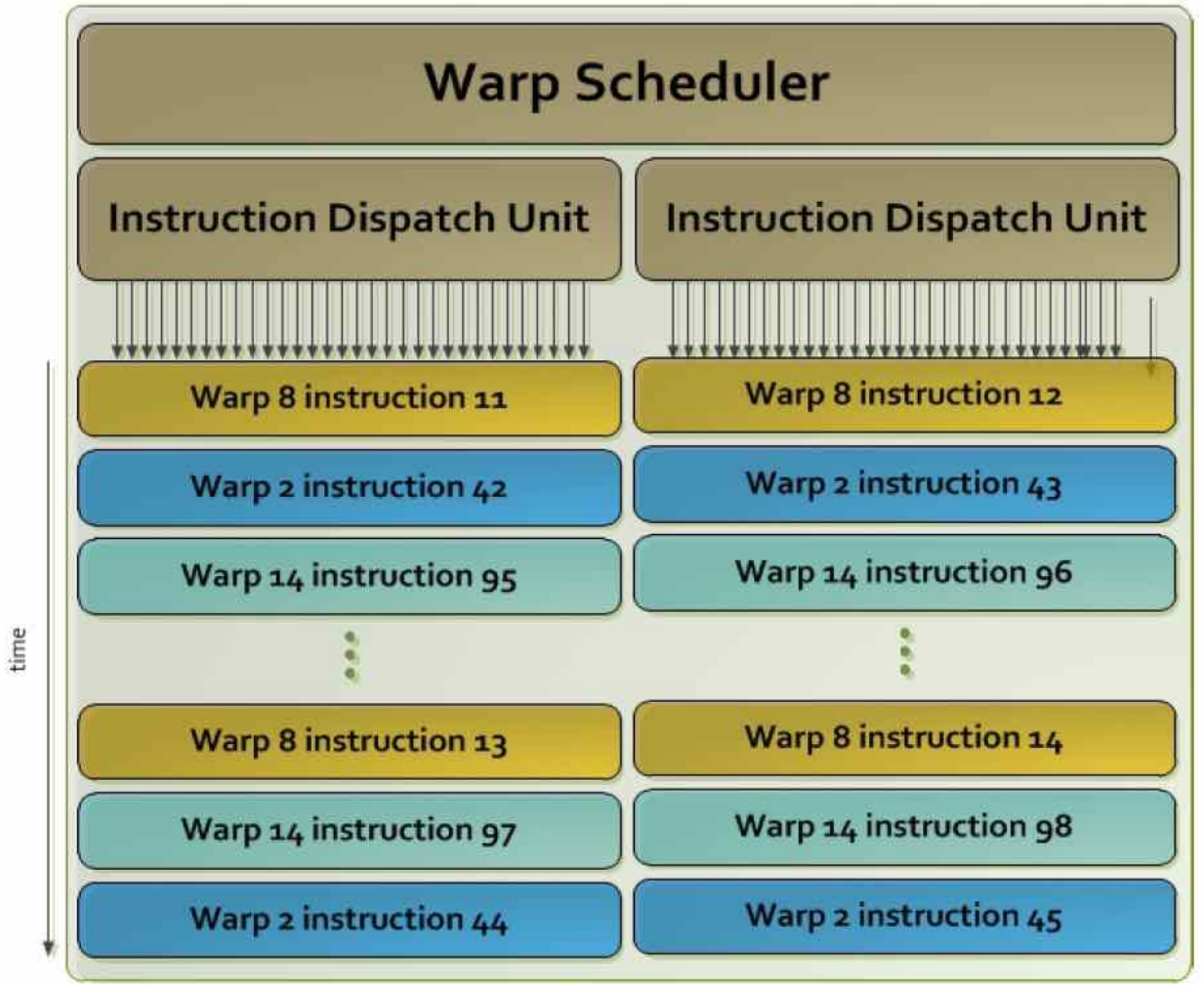
192 FP/Int
64 DP
32 LD/ST

256KB!

Partitioned
(user-defined)

NVIDIA Kepler GK110 Whitepaper

How Threads Are Scheduled



NVIDIA Kepler GK110 Whitepaper

What happens when threads in a warp diverge?

- Consider:
  - if(tid % 2) dothis(); else dothat();
- Transparently, GPU splits warp (branch divergence)
  - Record meeting point
  - Execute one side of branch, wait
  - Execute other side
  - Recombine at meeting point
- SIMT Execution
- May happen on cache misses too! (memory divergence)

CPU vs the GPU

Parameter	CPU	GPU
Clockspeed	> 1 GHz	700 MHz
RAM	GB to TB	12 GB (max)
Memory B/W	~ 60 GB/s	> 300 GB/s
Peak FP	< 1 TFlop	> 1 TFlop
Concurrent Threads	O(10)	O(1000) [O(10000)]
LLC cache size	> 100MB (L3) [eDRAM] O(10) [traditional]	< 2MB (L2)
Cache size per thread	O(1 MB)	O(10 bytes)
Software-managed cache	None	48KB/SMX
Type	OOO superscalar	2-way Inorder superscalar

Using a GPU

1. You must retarget code for the GPU
2. The working set must fit in GPU RAM
3. You must copy data to/from GPU RAM
4. Data accesses should be streaming
5. Or use scratchpad as user-managed cache
6. Lots of parallelism preferred (throughput, not latency)
7. SIMD-style parallelism best suited
8. High arithmetic intensity (FLOPs/byte) preferred

GPU Showcase Applications

- Graphics rendering
- Matrix Multiply
- FFT

See “Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU” by V.W.Lee et al. for more examples and a comparison of CPU and GPU

GPU Programming Models

Hierarchy of GPU Programming Models

Model	GPU	CPU Equivalent
Vectorizing Compiler	PGI CUDA Fortran	gcc, icc, etc.
“Drop-in” Libraries	cuBLAS	ATLAS
Directive-driven	OpenACC, OpenMP-to-CUDA	OpenMP
High-level languages	pyCUDA, OpenCL, CUDA	python
Mid-level languages		pthreads + C/C++
Low-level languages	-	PTX, Shader
Bare-metal	Assembly/Machine code	SASS

- Ordered by increasing order of difficulty
- Ordered by increasing order of flexibility

“Drop-in” Libraries

- “Drop-in” replacements for popular CPU libraries, examples from NVIDIA:
  - CUBLAS/NVBLAS for BLAS (e.g. ATLAS)
  - CUFFT for FFTW
  - MAGMA for LAPACK and BLAS
- These libraries may still expect you to manage data transfers manually
- Libraries *may* support multiple accelerators (GPU + CPU + Xeon Phi)

GPU Libraries

- NVIDIA Thrust
  - Like C++ STL
- Modern GPU
  - At first glance: high-performance library routines for sorting, searching, reductions, etc.
  - A deeper look: Specific “hard” problems tackled in a different style
- NVIDIA CUB
  - Low-level primitives for use in CUDA kernels
  -

Directive-driven Programming

- OpenACC, new standard for “offloading” parallel work to an accelerator
- Currently supported only by PGI Accelerator compiler
- gcc 5.0 support is ongoing
- OpenMPC, a research compiler, can compile OpenMP code + extra directives to CUDA

```
#include <stdio.h>
#define N 1000000

int main(void) {
  double pi = 0.0f; long i;

  #pragma acc parallel loop reduction(+:pi)
  for (i=0; i<N; i++) {
     double t= (double)((i+0.5)/N);
     pi +=4.0/(1.0+t*t);
  }

  printf("pi=%16.15f\n",pi/N);
  return 0;
}
```

Python-based Tools (pyCUDA)

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy
from pycuda.compiler import SourceModule

mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)

multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

CUDA Source Code

Automatic Data Transfers

Threads

OpenCL

- C99-based dialect for programming heterogenous systems
- Originally based on CUDA
  - nomenclature is different
- Supported by more than GPUs
  - Xeon Phi, FPGAs, CPUs, etc.
- Source code is portable (somewhat)
  - Performance may not be!
- Poorly supported by NVIDIA

CUDA

- “Compute Unified Device Architecture”
- First language to allow general-purpose programming for GPUs
  - preceded by shader languages
- Promoted by NVIDIA for their GPUs
- Not supported by any other accelerator
  - though commercial CUDA-to-x86/64 compilers exist
- We will focus on CUDA programs

CUDA Architecture

- From 10000 feet – CUDA is like pthreads
- CUDA language – C++ dialect
  - *Host code* (CPU) and GPU code in same file
  - Special language extensions for GPU code
- CUDA Runtime API
  - Manages runtime GPU environment
  - Allocation of memory, data transfers, synchronization with GPU, etc.
  - Usually invoked by host code
- CUDA Device API
  - Lower-level API that CUDA Runtime API is built upon
  -

CUDA π-calculation (GPU)

```
/* -*- mode: c++ -*- */
#include <cuda.h>
#include <stdio.h>

__global__
void pi_kernel(int N, double *pi_value) {
  int tid = blockIdx.x * blockDim.x + threadIdx.x;
  int nthreads = blockDim.x * gridDim.x;

  double pi = 0;

  for(int i = tid; i < N; i+=nthreads) {
    double t = ((i + 0.5) / N);
    pi += 4.0/(1.0+t*t);
  }

  pi_value[tid] = pi;
}
```

For CUDA Runtime API

Indicates GPU Kernel that can be called by CPU

Calculates thread identifier and total number of threads

pi_value is stored in GPU memory i.e. it is a pointer to GPU memory

CUDA π-calculation (CPU)

```
int main(void) {
  int NTHREADS = 2048;
  int N = 10485760;

  double *c_pi, *g_pi;
  size_t g_pi_size = NTHREADS * sizeof(double);

  c_pi = (double *) calloc(NTHREADS, sizeof(double));

  if(cudaMalloc(&g_pi, g_pi_size) != cudaSuccess) {
    fprintf(stderr, "failed to allocate memory!\n");
    exit(1);
  };

  pi_kernel<<<NTHREADS/256, 256>>>(N, g_pi);

  if(cudaMemcpy(c_pi, g_pi, g_pi_size, cudaMemcpyDeviceToHost) != cudaSuccess) {
    fprintf(stderr, "failed to copy data back to CPU!\n");
    exit(1);
  }

  double pi = 0;
  for(int i = 0; i < NTHREADS; i++)
    pi += c_pi[i];

  printf("pi=%16.15f\n", pi/N);
  return 0;
}
```

CPU and GPU pointers

CPU Allocation

GPU Allocation

Call GPU Kernel

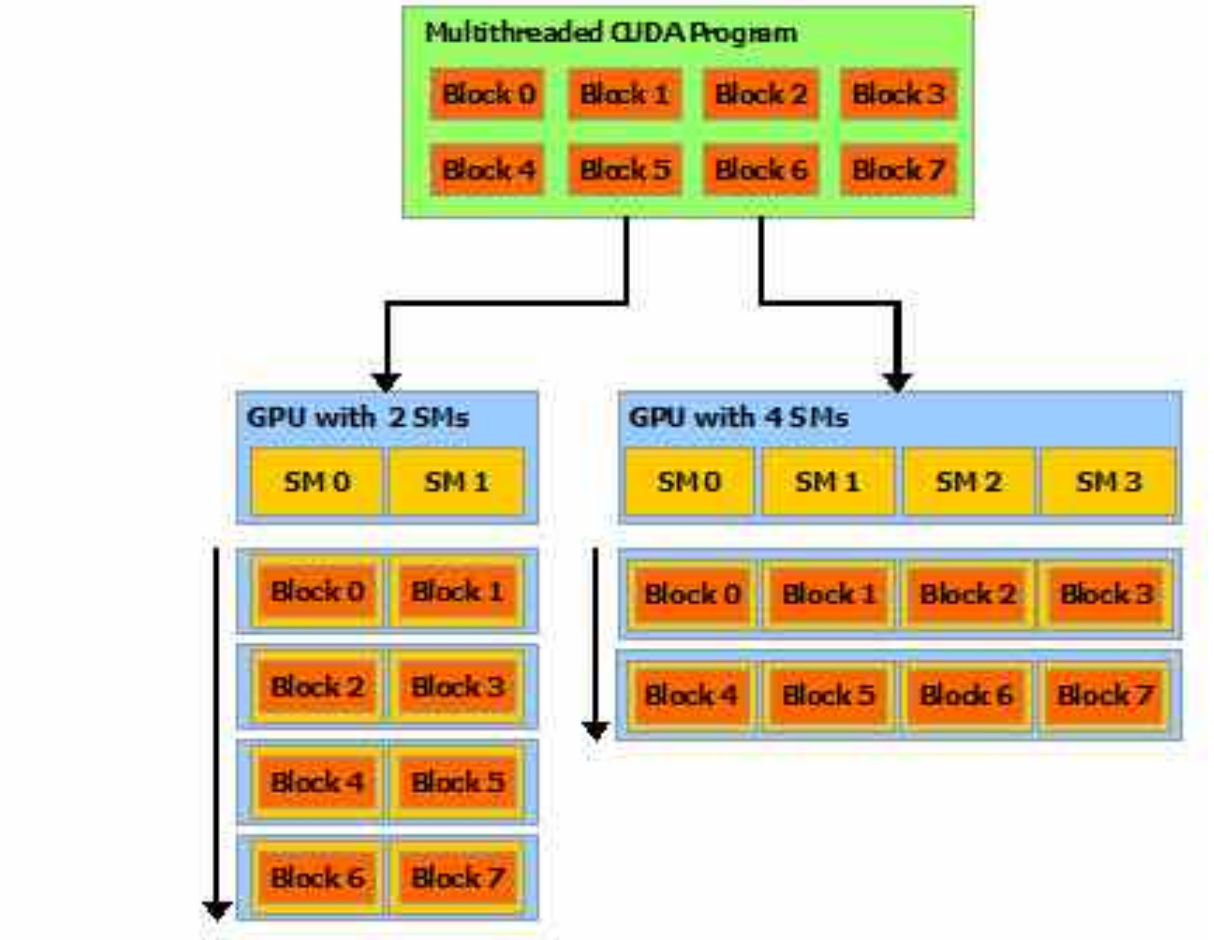
Copy back pi_value

Reduce pi_value on GPU

CUDA Threading Model Details

- Hierarchical Threading
  - Grid -> Thread Blocks -> Warps -> Threads
  - pthreads has flat threading
- Only threads within same thread block can communicate and synchronize with each other
- Maximum 1024 threads per thread block
  - Differs by GPU generation
- Thread block is divided into mutually exclusive, equally-sized group of threads called *warps*
  - Warp size is hardware-dependent
  - Usually 32 threads

Mapping Threads to Hardware in CUDA



CUDA Limitations

- No standard library
- No parallel data structures
- No synchronization primitives (mutex, semaphores, queues, etc.)
  - you can roll your own
  - only atomic*() functions provided
- Toolchain not as mature as CPU toolchain
  - Felt intensely in performance debugging
- It's only been a decade :)

Summary

- GPUs are very interesting parallel machines
- They're not going away
  - Xeon Phi might be pose huge challenge
- They're here and now
  - You can buy one of them!