Programming Massively Parallel Processors

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

1

Administrative

- Prerequisites:
  - EECS 213 or equivalent understanding of computer systems and architecture
  - EECS 211 or EECS 230 or equivalent intermediate C programming experience
- Prerequisites By Topic
  - Basic concepts of computer architecture
    - processors, ALUs, memories, caches, input-output
  - Basic concepts of computer systems
    - memory accessing and layout, padding, alignment
  - Intermediate concepts on programming using C/C++
    - control flow, scoping, type casting, pointers, (helpful: templates)
  - Simple concepts of data structures
    - E.g., arrays and linked lists in programs
  - Knowledge of scientific and engineering applications
    - For EECS 468 projects

2

Administrative

- Grades

	EECS-368	EECS-468
Lab assignments	80%	50%
Final project		30%
Class participation	20%	20%

  - I didn't say "class presence"; I said "**class participation**", i.e.,
    - Interact during lecture: ask questions, answer questions
    - Answer questions on piazza
  - http://piazza.com/northwestern/winter2020/eecs368468
- No exams
  - Registrar-scheduled exam day: Thursday, 03/19/2020, 12-2pm
  - This will be our overflow day. Plan to be there.
- Labs, projects
  - Groups of 2-3. **Search for teammates on piazza**
  - The instructors will create groups on Canvas. You will sign up on those.
  - Late assignments: 10% grade penalty up to 24h late. Zero after that.

3

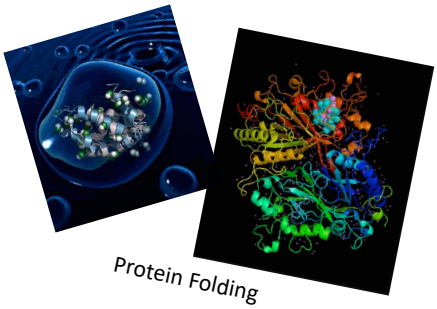
Administrative

- Instructor: Nikos Hardavellas
  - nikos@northwestern.edu
  - Office hours: Tuesdays 3:30-4:30pm
  - Where: Mudd 3517

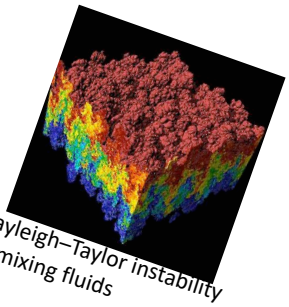
- TA: Emirhan (Emir) Poyraz
  - IbrahimPoyraz2014@u.northwestern.edu
  - Office hours: Wednesdays 2-3pm
  - Where: Wilkinson Lab, Tech M338

4

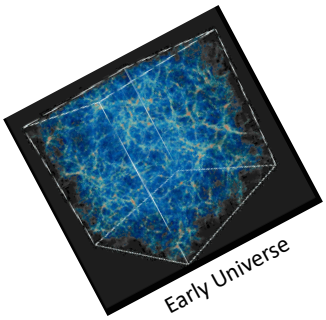
We have big computational challenges to solve



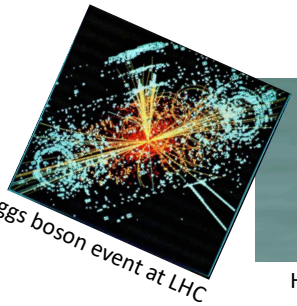
Protein Folding



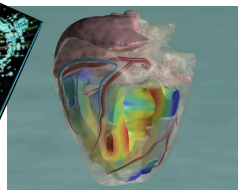
Rayleigh–Taylor instability in mixing fluids



Early Universe



Higgs boson event at LHC



Heart simulation

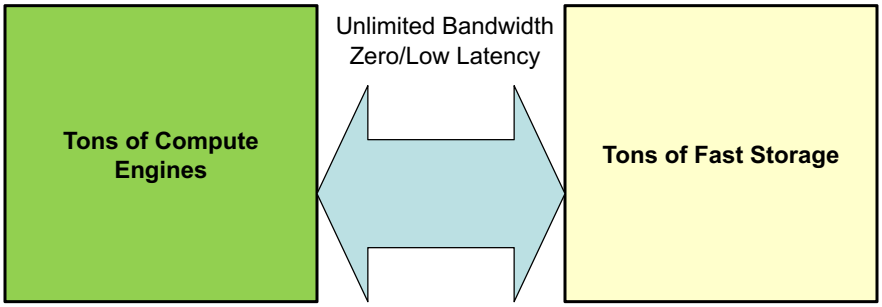
5

What computation machine do we want?

6

Understanding Semiconductor Technology Limitations

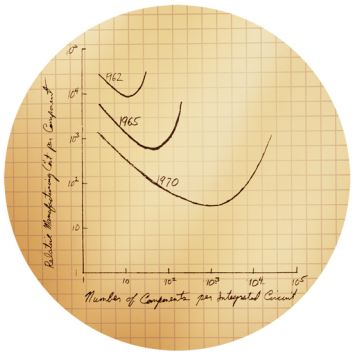
This is what we would like to have



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

7

Moore's Law



Originally presented in 1965

Moore claimed that the number of transistors per unit of area will double every year

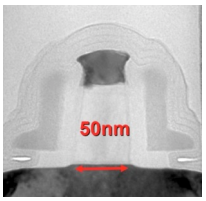
Revised in 1975, as progress slowed: "doubles every two years"

We typically use the average: "**the number of transistors per unit of area doubles every 18 months**"

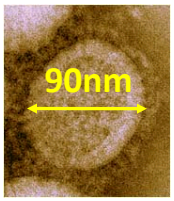
Why does it happen? Because transistors are getting smaller!

8

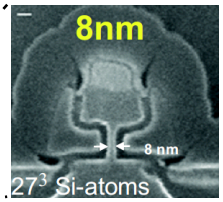
How Small is "Small"?

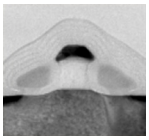


90nm transistor
(Intel, 2005)

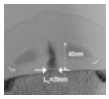


Swine Flu A/H1N1
(CDC)





65nm
2007



45nm
2009

32nm
2011

22nm
2013

16nm
2015

9

Analogy to Moore's Law

$10 base; 60% growth

Year	Wealth	Comments
0	$10	Base Cash

10

Analogy to Moore's Law

$10 base; 60% growth

Year	Wealth	Comments
0	$10	Base Cash
3	$40	Still live at home with mom & dad

11

Analogy to Moore's Law

$10 base; 60% growth

Year	Wealth	Comments
0	$10	Base Cash
3	$40	Still live at home with mom & dad
16	$18K	Buy car

12

Analogy to Moore's Law

$10 base; 60% growth

Year	Wealth	Comments
0	$10	Base Cash
3	$40	Still live at home with mom & dad
16	$18K	Buy car
21	$193K	Buy an apartment in Evanston

13

Analogy to Moore's Law

$10 base; 60% growth

Year	Wealth	Comments
0	$10	Base Cash
3	$40	Still live at home with mom & dad
16	$18K	Buy car
21	$193K	Buy median apartment in Evanston
36	$223M	Need fundamentally new ways to spend money

14

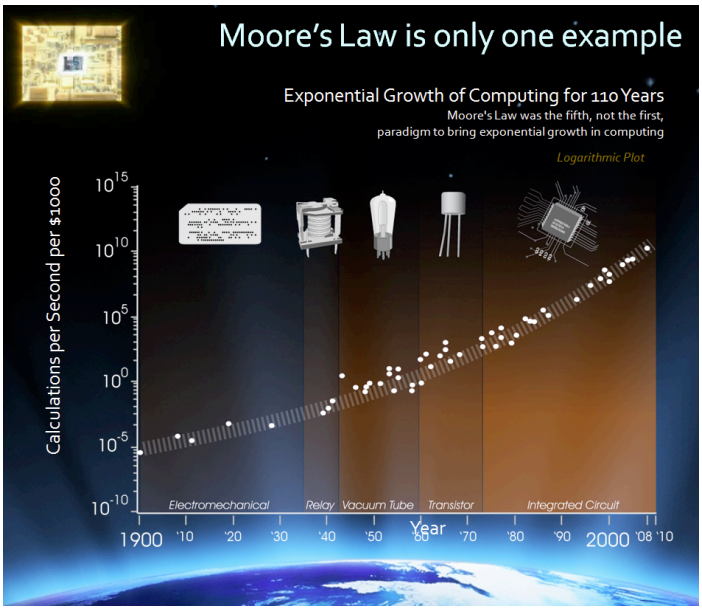
Analogy to Moore's Law

$10 base; 60% growth

Year	Wealth	Comments
0	$10	Base Cash
3	$40	Still live at home with mom & dad
16	$18K	Buy car
21	$193K	Buy median apartment in Evanston
36	$223M	Need fundamentally new ways to spend money
56	$2.7T	Replace US Federal Government

15

Exponential Growth in Computing for 110 years

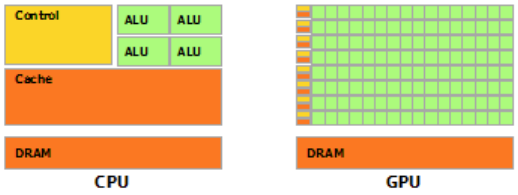


Ray Kurzweil

16

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

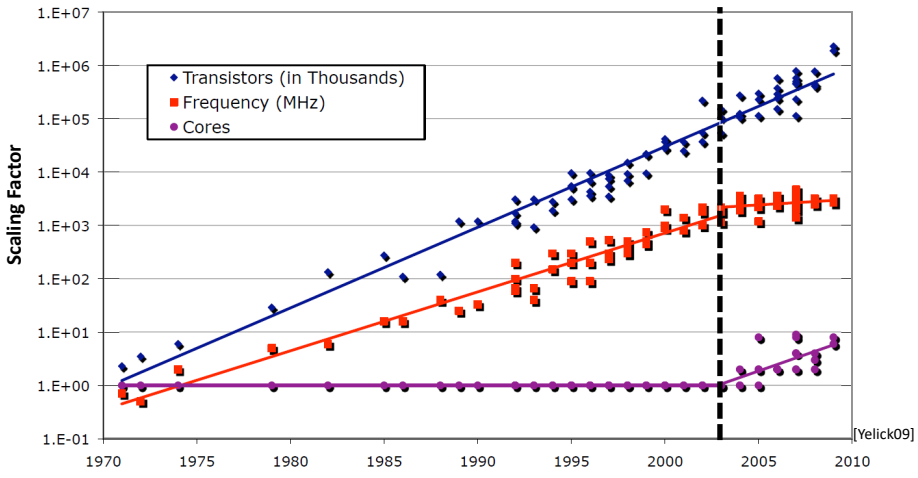


➡ Can build lots of calculation units! (Trend: even more)

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

17

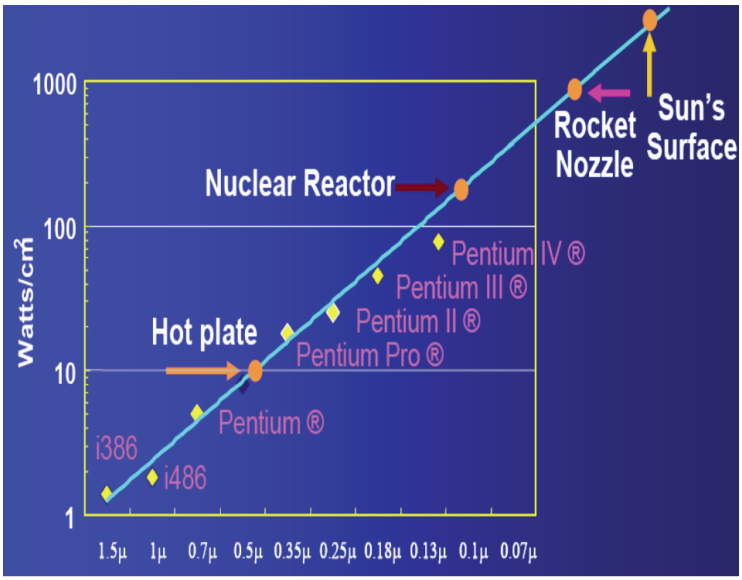
Moore's Law in Trouble ?



➡ What happened in Nov 2002?

18

The Power Wall: Increasing Thermal Density



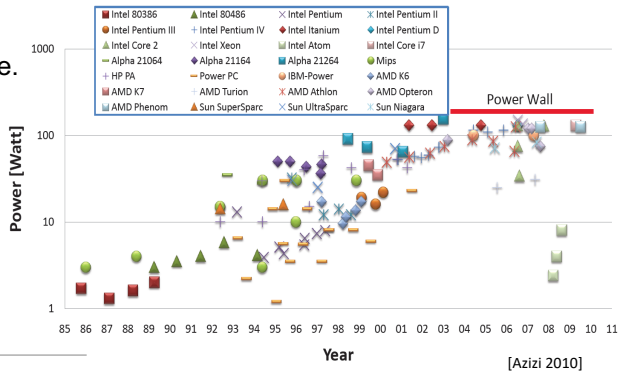
- More transistors + higher switching speeds → higher temperature

19

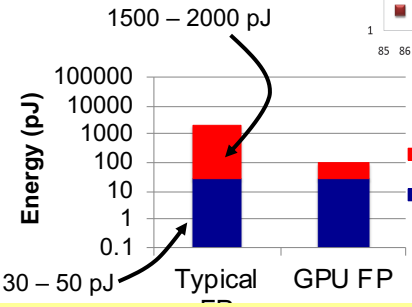
Why GPUs and not Multicores with 1000+ cores?

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

How can we get more performance?



[Azizi 2010]



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

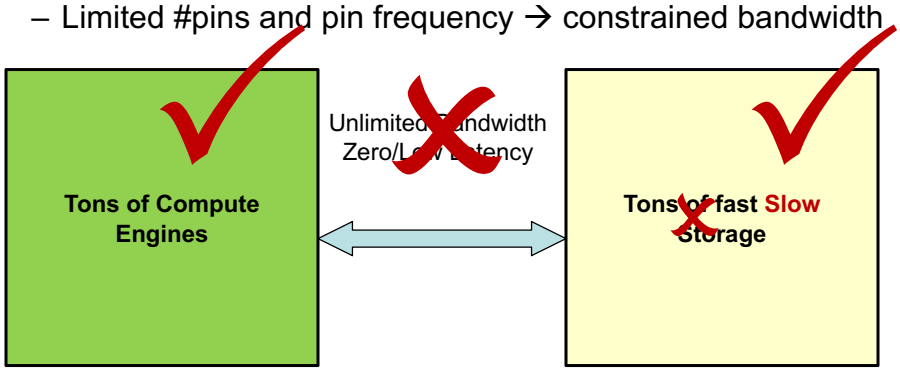
Use the available power more **efficiently** to get more useful work done per Watt.

- General-purpose cores have too much overhead.
- Amortize overhead among many arith. units → GPUs

20

How about Communication/Storage

- Need data feed and storage
  - 1000's compute engines need tons of storage to keep busy
  - But, the larger the storage, the slower it is
  - So you can get tons of SLOW storage
  - Limited #pins and pin frequency → constrained bandwidth



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

➡ PARALLELISM !!!

21

Some things are naturally parallel

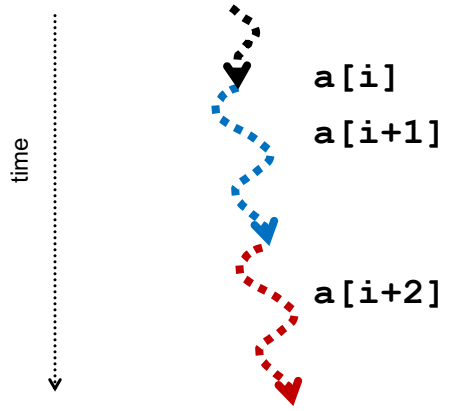
e.g., image fading…

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

22

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

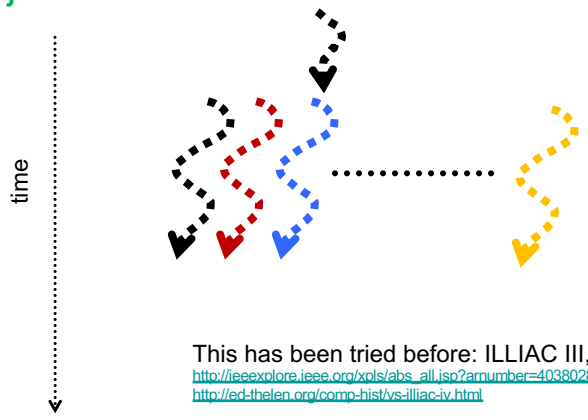
This is the predominant CPU model

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

23

Data Parallel Execution Model / SIMD

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



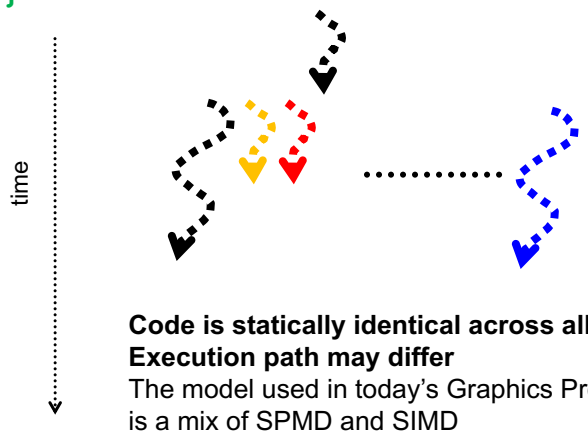
Most modern CPUs offer some limited support for SIMD

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

24

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

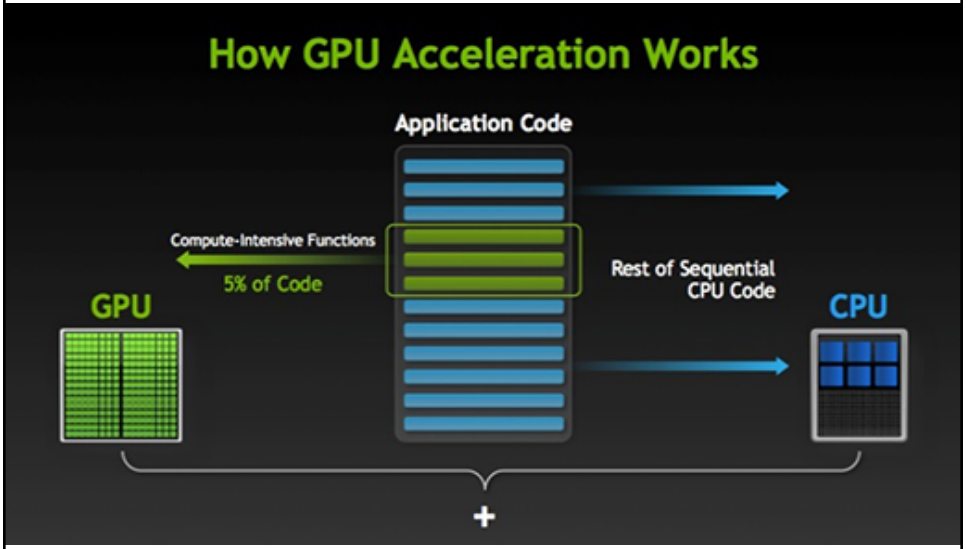
25

CPU vs. GPU overview

- CPU:
  - Handles sequential code well
    - Latency optimized: do all very fast
  - Can't take advantage of massively parallel code
  - Off-chip bandwidth lower – narrow pipes
  - Lower peak computation capability
- GPU:
  - Requires massively parallel computation
    - Bandwidth optimized: do lots concurrently
  - Handles some control flow
  - Higher off-chip bandwidth – wide pipes
  - Higher peak computation capability

26

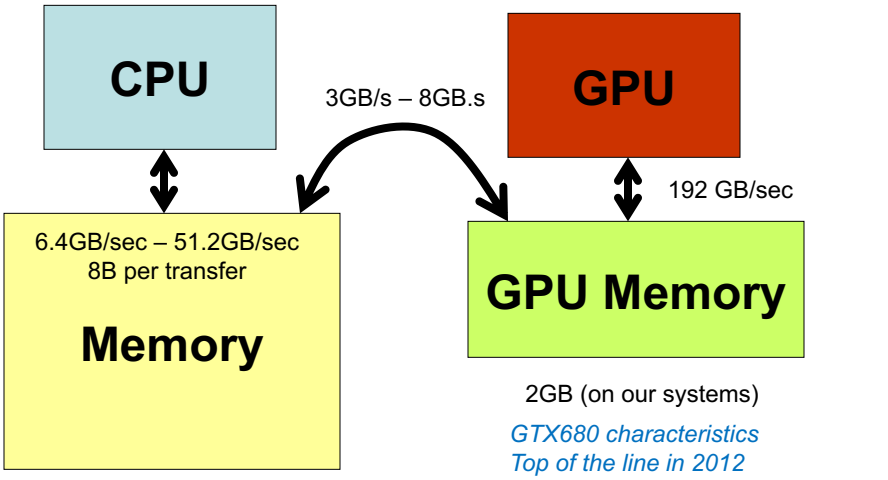
How GPU Acceleration Works



27

Programmer's view of GPU computation

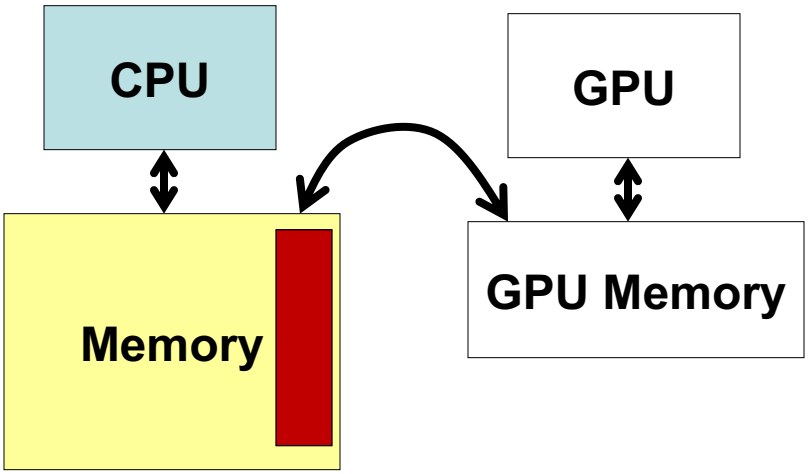
- GPU as a co-processor



28

Programmer's view of GPU computation

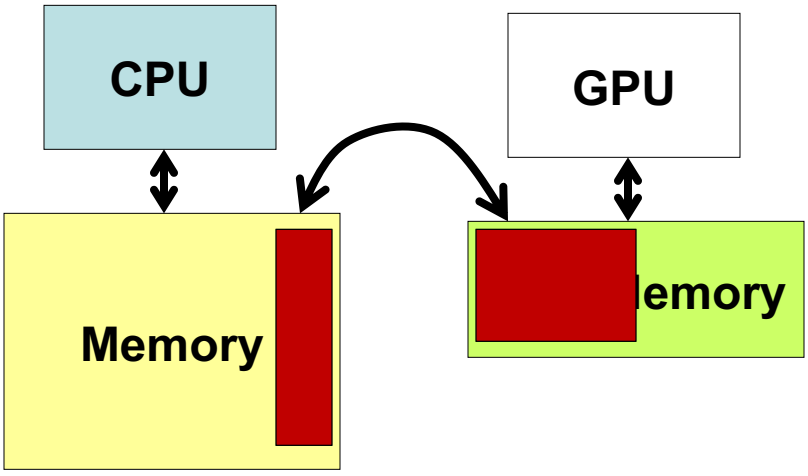
- First create data on CPU memory



29

Programmer's view of GPU computation

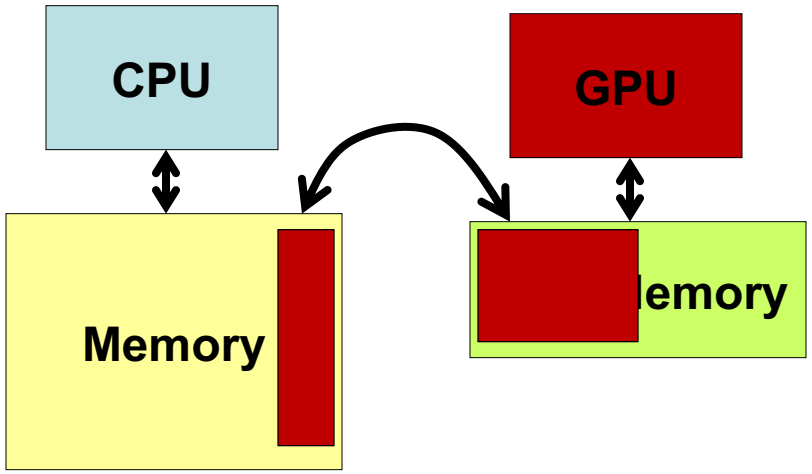
- Then Copy to GPU



30

Programmer's view of GPU computation

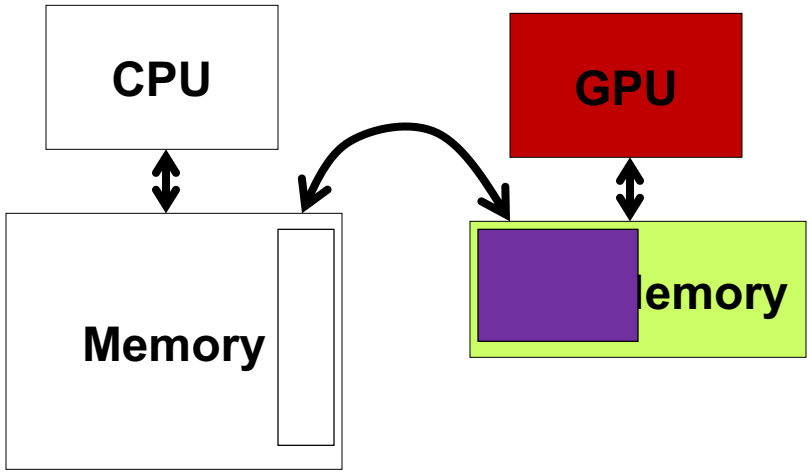
- GPU starts computation → runs a **kernel**
- CPU can also continue



31

Programmer's view of GPU computation

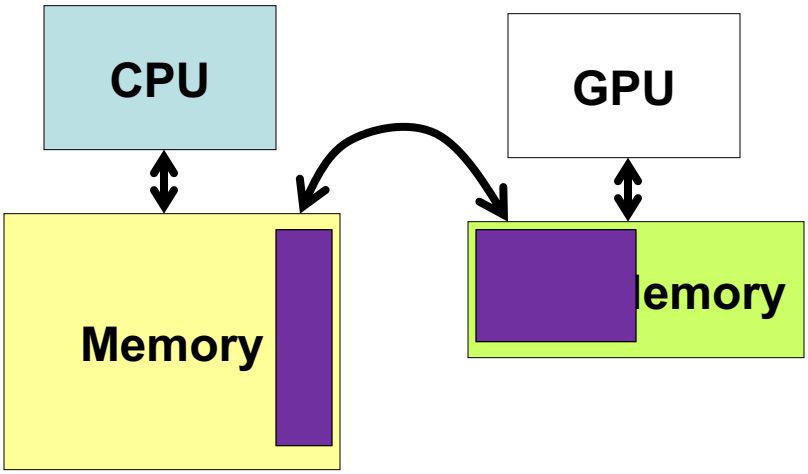
- CPU and GPU Synchronize



32

Programmer's view of GPU computation

- Copy results back to CPU



33

But what about performance?

- Focus on PEAK performance first:
  - What the manufacturer guarantees you'll never exceed
- Two Aspects:
  - **Data Processing Capability**
    - How many ops per sec
  - **Data Access Rate Capability**
    - Bandwidth

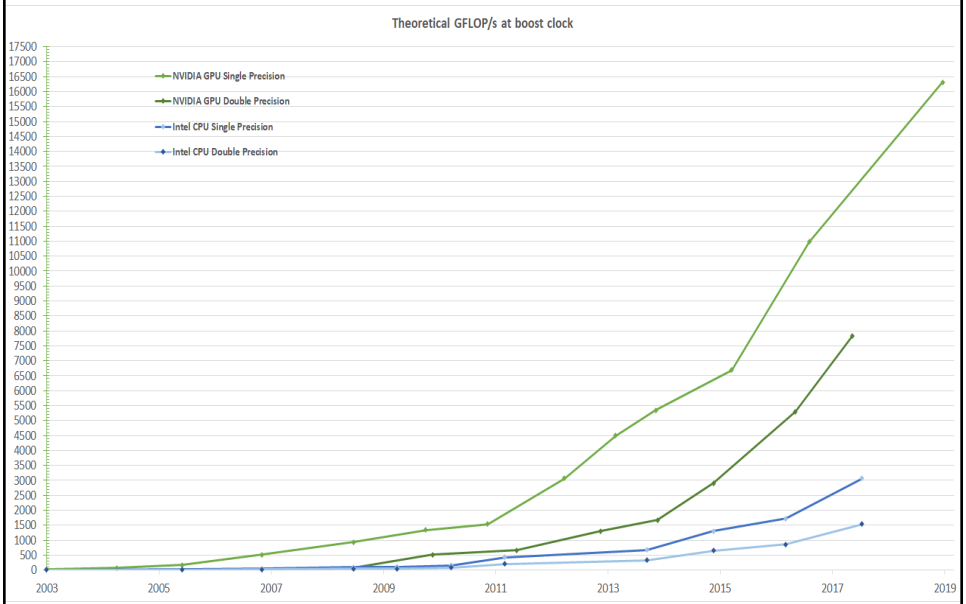
34

Data Processing Capability

- Focus on floating point data
- GFLOPS
  - Billion (giga) Floating-Point Operations per Second
- High-End CPU circa 2012 (Core i7 3900 Extreme)
  - 187 GFLOPS (single core TurboBoost)
  - 158.4 GFLOPS (base) x 4 cores = 633.6 GFLOPS
- High-End GPU circa 2012 / GTX680
  - 3090 GFLOPS or **5x capability**

35

Data Processing Capability



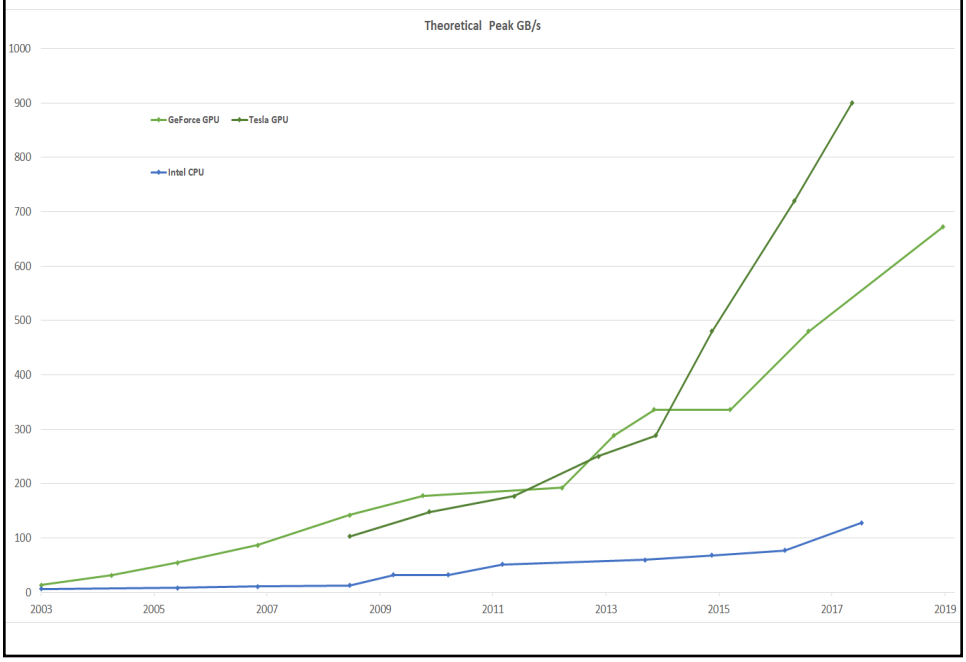
36

Data Access Capability

- **High-End CPU circa 2012**
  - Bus width 4x64-bit (256-bit aggregate)
  - 51.2GB/sec (@ 1600MHz)
- **GPU / GTX680 circa 2012**
  - 192 GB/sec
  - Bus width 256-bit
  - **4x capability**

37

Data Access Capability



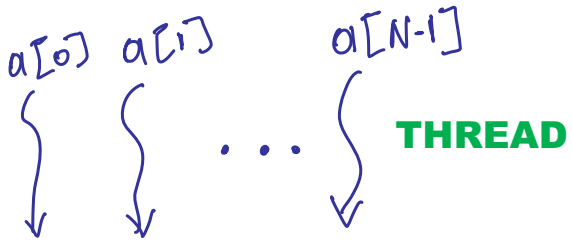
38

Target Applications

```
int a[N]; // N is large
for all elements of an array
  a[i] = a[i] * fade
```

Kernel

- Lots of **independent** computations
  - CUDA threads need not be completely independent
  - …but it helps if they are…



39

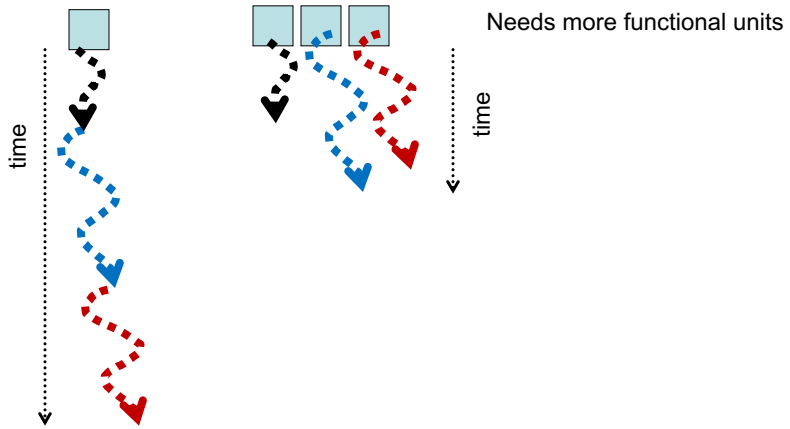
Programmer's View of the GPU

- GPU: a compute device that:
  - Is a coprocessor to the CPU or **host**
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as **kernels** which run in parallel on many threads

40

Why are threads useful? Parallelism

- Concurrency:
  - **Do multiple things in parallel**

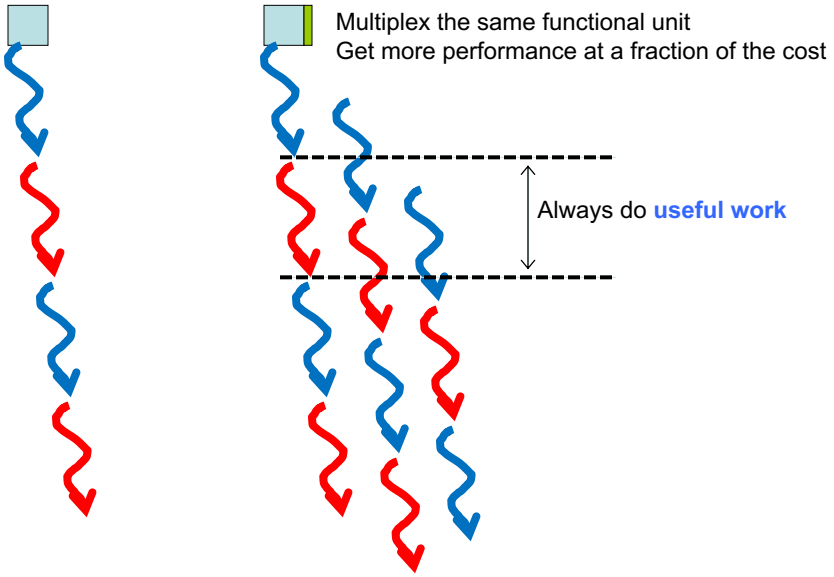


  - Uses more hardware → Gets higher performance
  - Application must have parallelism

41

Why are threads useful #2 – Tolerating stalls

- Often a thread stalls, e.g., memory access



42

GPU: bandwidth optimized – latencies are long

- A GPU ADD takes 24 GPU cycles
  - CPU ADD = 1 cycle (for a single core)
- The GPU cycle is ~4x of a CPU cycle
  - GTX680 vs. 3.8GHz Core i7
- In the time it takes to do 1 ADD in the GPU
  - CPU does ~100 ADDs (single)
  - **Need ~100 threads to break even with 1 CPU**
- 1000s of threads for GPU to be better

43

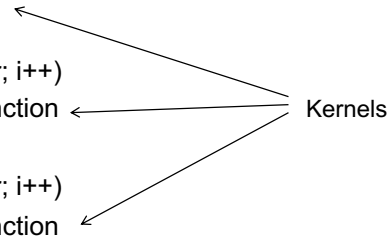
GPU vs. CPU Threads

- GPU threads are extremely lightweight
  - Very little creation overhead
  - In the order of microseconds
  - All done in hardware
- GPU needs 1000s of threads for full efficiency
  - Multi-core CPU needs only a few
- To run all these threads, GPUs are massively parallel architectures

44

Computation partitioning:

- 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

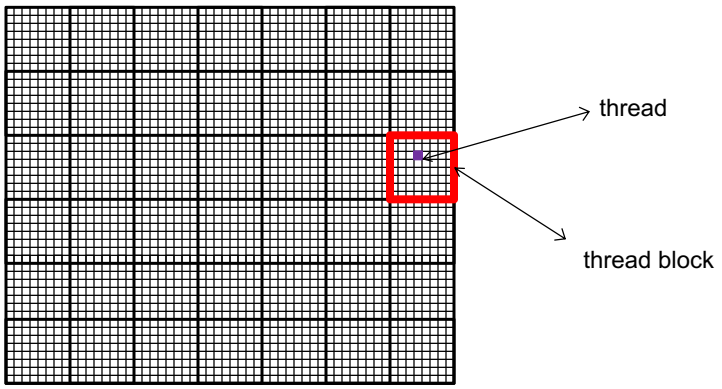


  - Per-Kernel partitioning: Think of data as an array

45

Per-Kernel Computation Partitioning

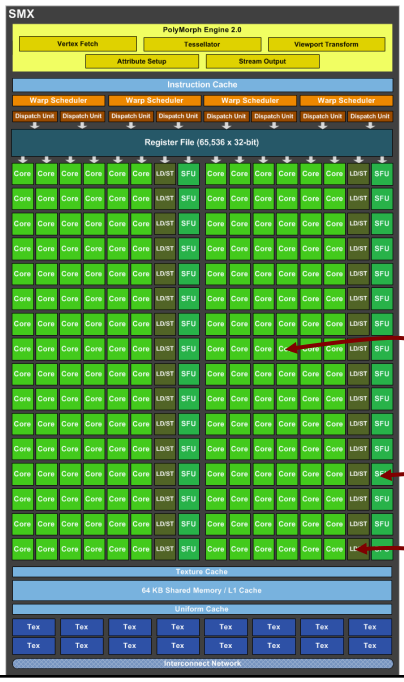
- Example of a Computation Grid: 2D Case



- Threads within a thread block run on the same multiprocessor
- Thread blocks scheduled independently across multiprocessors

46

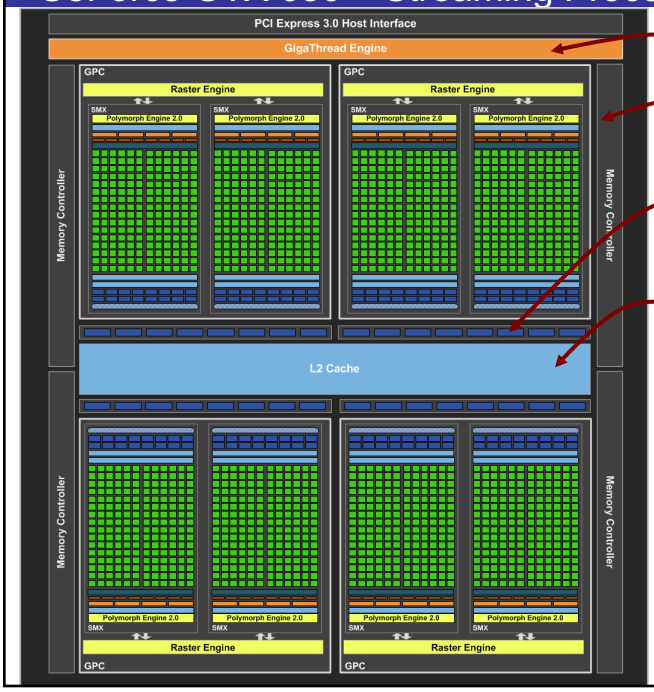
GeForce GTX 680 – Streaming Multiprocessor



- **SM (a.k.a. SMX, SMP)**
  - Streaming Multiprocessor
  - Multi-threaded processor
    - 192 CUDA cores
    - 1 to 2048 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

47

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

48

Programming Languages

- CUDA
  - NVIDIA
  - (Probably still) has market lead in GPUs
- OpenCL
  - Many including NVIDIA
  - CUDA superset
  - Somewhat different syntax
  - Can target many different devices, e.g., CPUs + programmable accelerators
  - Fairly new
- We'll focus on CUDA for now
- Both are evolving

49

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 (d_a));
}
```

CPU

50