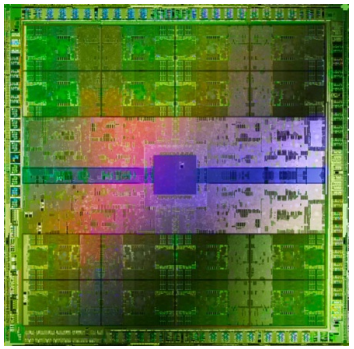
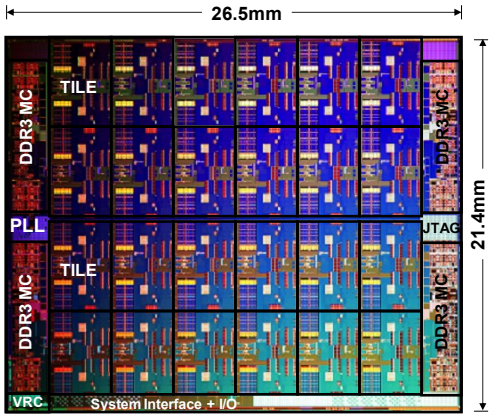
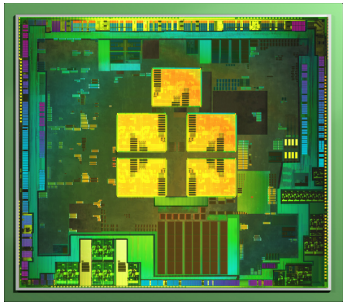


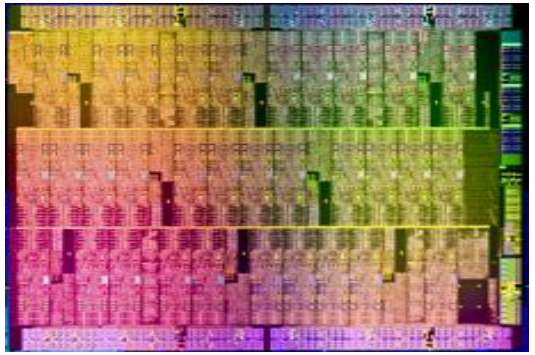
NVIDIA GTX 480 processor



Intel labs 48 core SCC processor



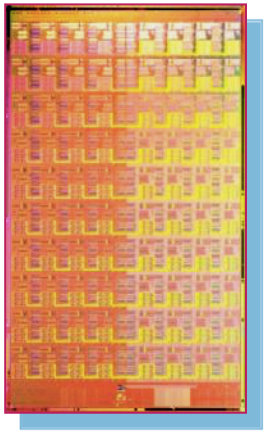
NVIDIA Tegra 3 (quad Arm Corex A9 cores + GPU)



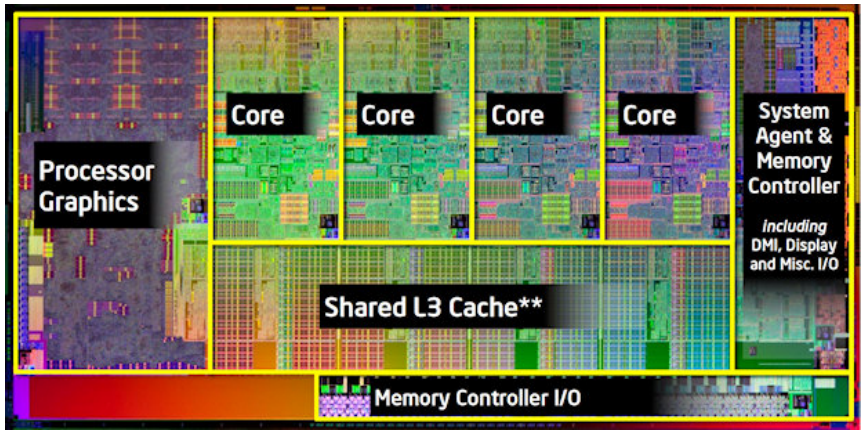
An Intel MIC processor

GPUs and the Heterogeneous programming problem

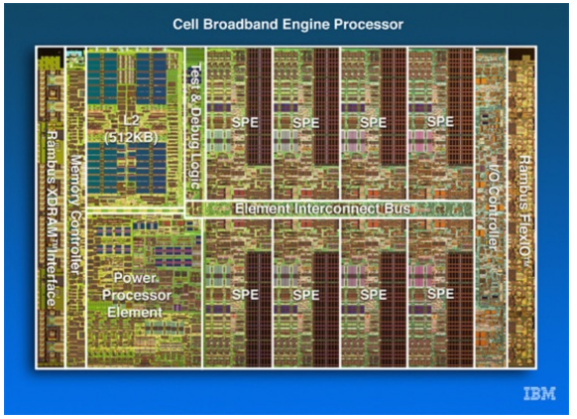
Tim Mattson (Intel Labs)



Intel Labs 80 core Research processor



Intel "Sandybridge" processor



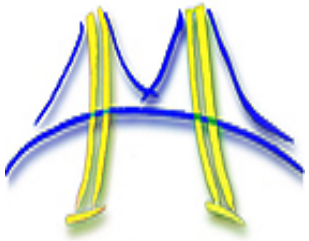
IBM Cell Broadband engine processor

Other than the Intel lab's research processors. Die photos from UC Berkeley CS194 lecture notes

Disclaimer

READ THIS … its very important

- The views expressed in this talk are those of the speakers and not their employer.
- This is an academic style talk and does not address details of any particular Intel product. You will learn nothing about Intel products from this presentation.
- This was a team effort, but if we say anything really stupid, it's our fault … don't blame our collaborators.

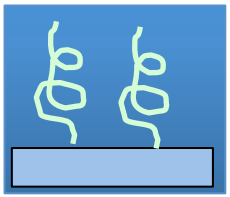


Slides marked with this symbol were produced-with Kurt Keutzer and his team for CS194 … A UC Berkeley course on Architecting parallel applications with Design Patterns.

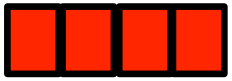
Outline

- The SIMT platform
- Understanding the GPU and GPGPU programming
- The 100X GPU/CPU speedup Myth
- The dream of performance portability
- The future of the GPU

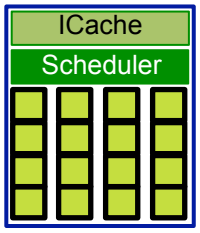
Hardware Diversity: Basic Building Blocks



CPU Core: one or more hardware threads sharing an address space. Optimized for low latencies.

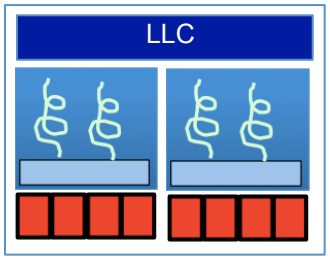


SIMD: Single Instruction Multiple Data.
Vector registers/instructions with 128 to 512 bits so a single stream of instructions drives multiple data elements.

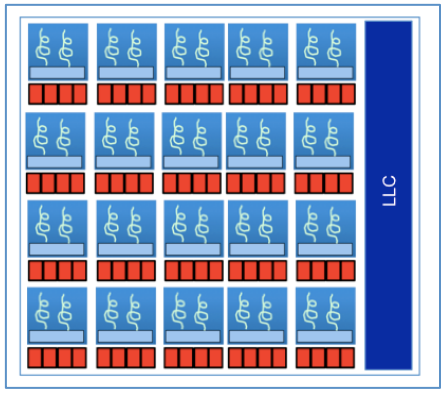


SIMT: Single Instruction Multiple Threads.
A single stream of instructions drives many threads. More threads than functional units. Over subscription to hide latencies. Optimized for throughput.

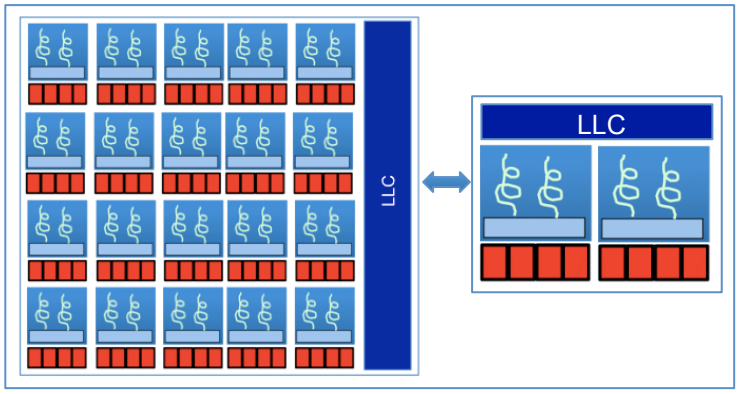
Hardware Diversity: Combining building blocks to construct nodes



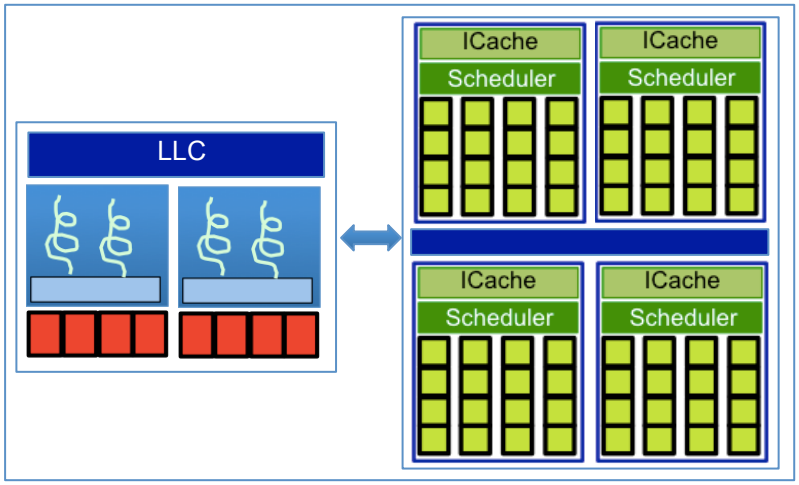
Multicore CPU



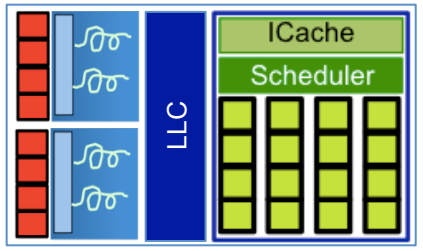
Manycore CPU



Heterogeneous:
CPU + manycore coprocessor



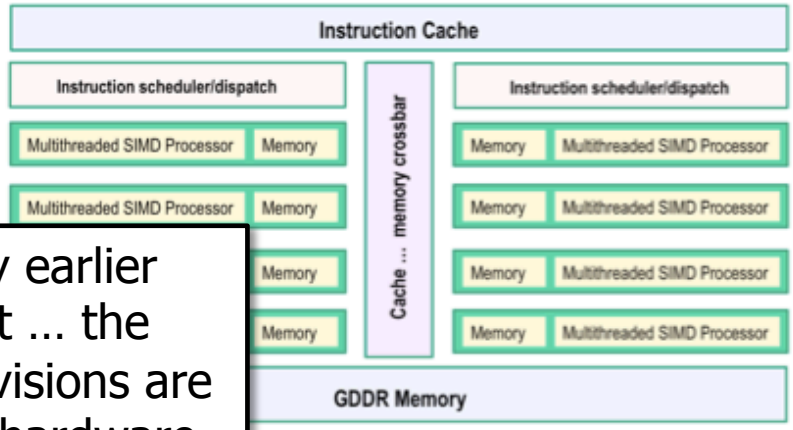
Heterogeneous: CPU+GPU



Heterogeneous:
Integrated CPU+GPU

The software platform debate!

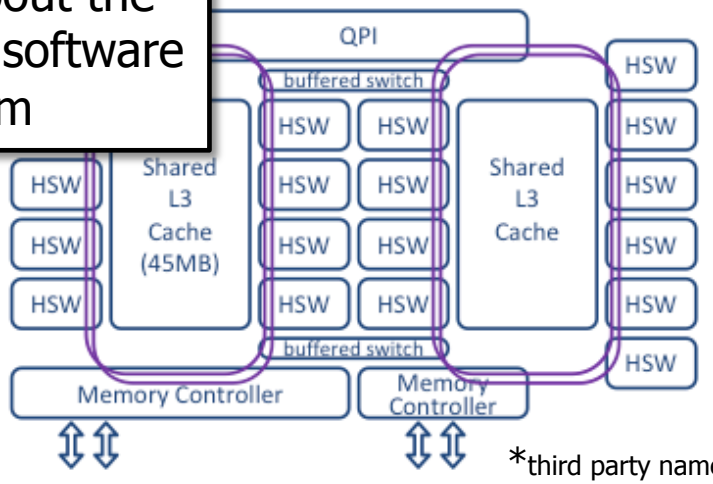
It's really about competing software platforms



GPU

- **Single Instruction multiple threads.**
  - turn loop bodies into kernels.
  - HW intelligently schedules kernels to hide latencies.
- *Dogma*: a natural way to express huge amounts of data parallelism
- Examples: CUDA, OpenCL, OpenACC

Recall my earlier comment … the competing visions are less about hardware and more about the programmers software platform

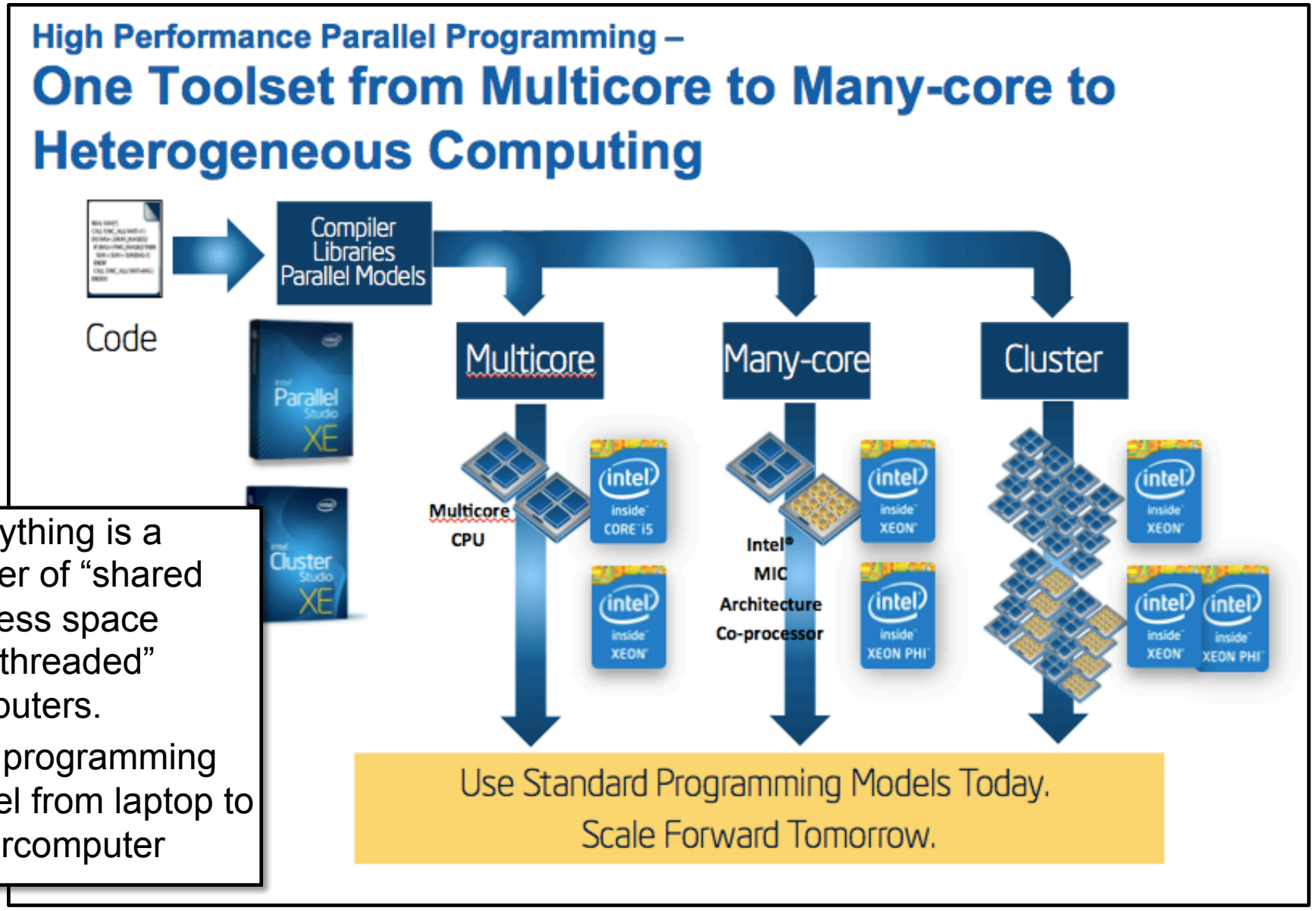


CPU

- **Shared Address space, multi-threading.**
  - Many threads executing with coherent shared memory.
- *Dogma*: The legacy programming model people already know. Easier than alternatives.
- Examples: OpenMP, Pthreads, C++11

*third party names are the property of their owners

The official strategy at Intel.



- Everything is a cluster of "shared address space multithreaded" computers.
- One programming model from laptop to supercomputer

What about the competing platform:
Single Instruction multiple thread (SIMT)?

- Dominant as a proprietary solution based on CUDA and OpenACC.
- But there is an Open Standard response (supported to varying degrees by all major vendors)

OpenCL

SIMT programming for CPUs, GPUs, DSPs, and FPGAs. Basically, an Open Standard that generalizes the SIMT platform pioneered by our friends at NVIDIA®

OpenMP™

OpenMP 4.0 added target and device directives … Based on the same work that was used to create OpenACC. Therefore, just like OpenACC, you can program a GPU with OpenMP!!!

The long term viability of the SIMT platform depends on the user community demanding (and using) the Open Standard alternatives!

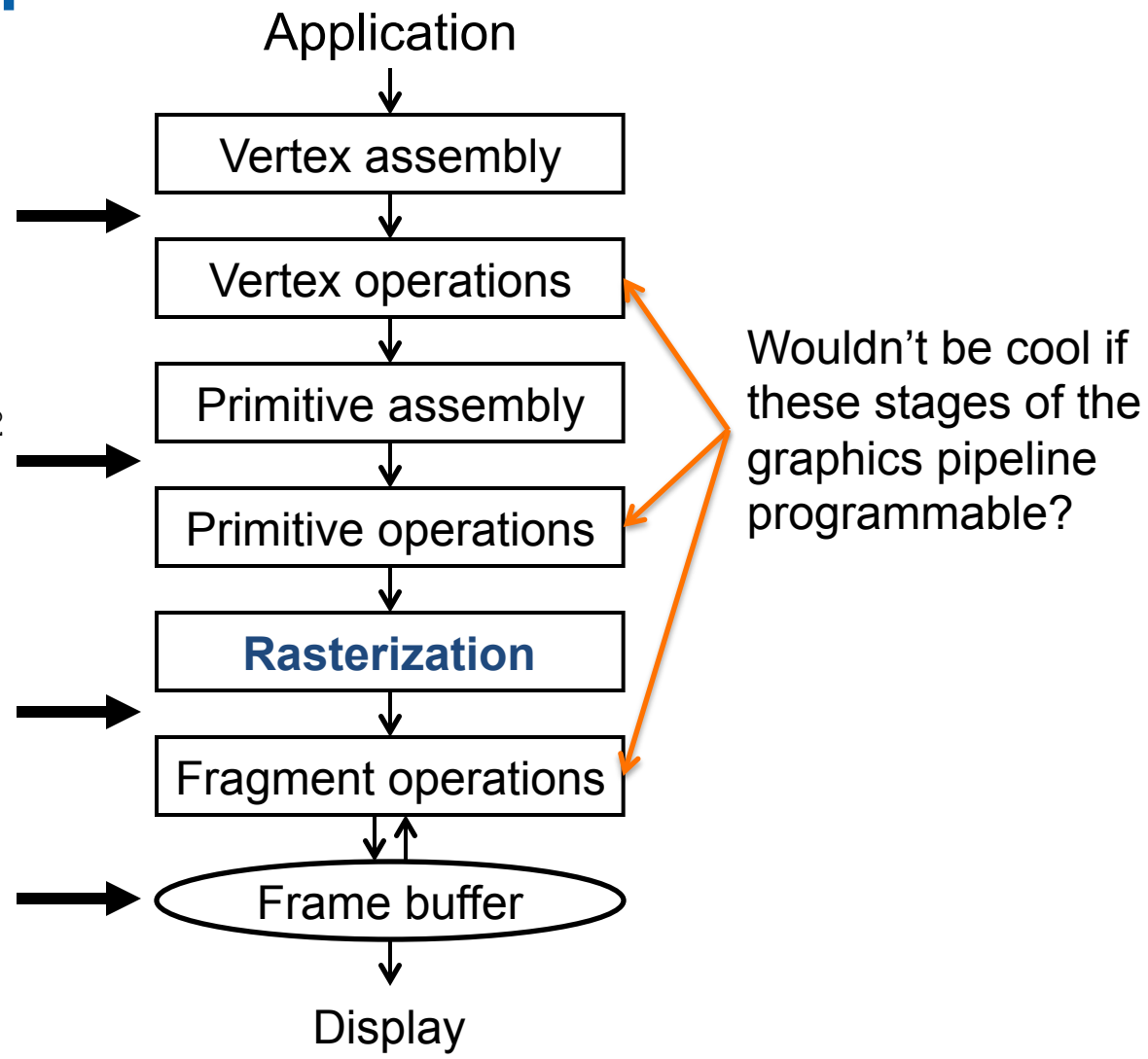
*third party names are the property of their owners

Outline

- The SIMT platform
- Understanding the GPU and GPGPU programming
- The 100X GPU/CPU speedup Myth
- The dream of performance portability
- The future of the GPU

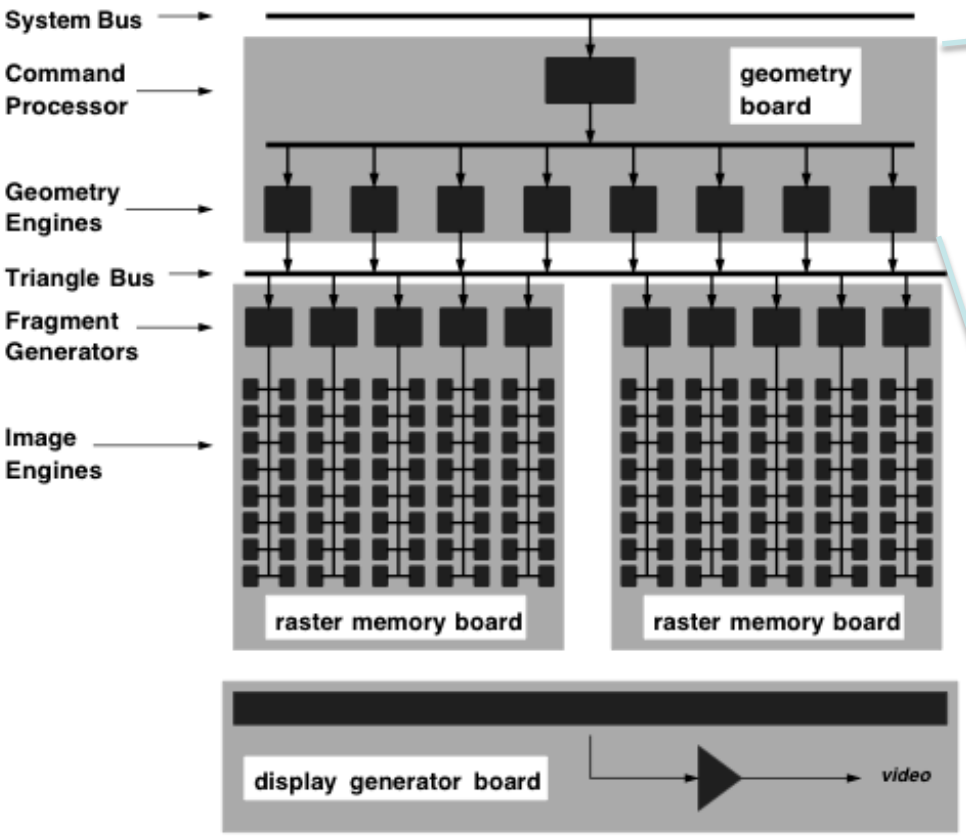
Let's take a deeper look at the GPU: The vertex pipeline

```
struct {
  float x,y,z,w;
  float r,g,b,a;
} vertex;

struct {
  vertex v0,v1,v2
} triangle;

struct {
  short int x,y;
  float depth;
  float r,g,b,a;
} fragment;

struct {
  int depth;
  byte r,g,b,a;
} pixel;
```



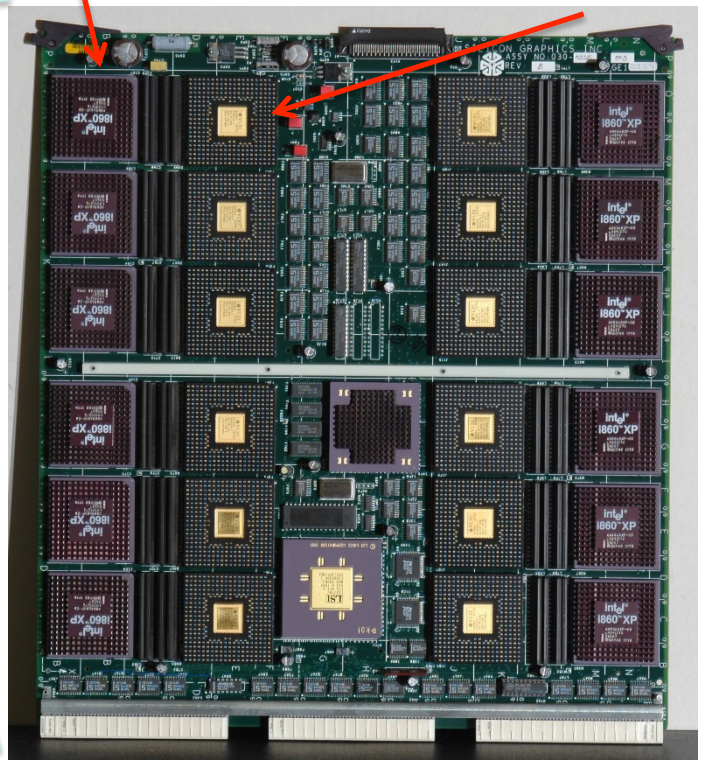
Thanks to Kurt Akeley

High-end GPUs have historically been programmable



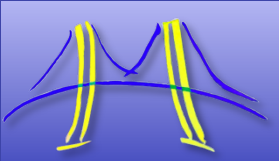
Intel i860 RISC CPU

Custom ASIC for processor interconnect



Silicon Graphics RealityEngine GPU
1993

- I860 billed as a “Cray-on-a-chip”
  0.80 micron technology
  2.5M transistors



Programming GPUs

First paper on GPGPU programming I could find dates to 1995 … though the term GPGPU didn't appear in the literature until ~2000.

Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware

Brian Cabral, Nancy Cam, and Jim Foran
Silicon Graphics Computer Systems*

Abstract

Volume rendering and reconstruction centers around solving two related integral equations: a volume rendering integral (a generalized Radon transform) and a filtered back projection integral (the inverse Radon transform). Both of these equations are of the same mathematical form and can be dimensionally decomposed and approximated using Riemann sums over a series of resampled images. When viewed as a form of texture mapping and frame buffer accumulation, enormous hardware enabled performance acceleration is possible.

1 Introduction

Volume Visualization encompasses not only the viewing but also the construction of the volumetric data set from the more basic projection data obtained from sensor sources. Most volumes used in rendering are derived from such sensor data. A primary example being Computer Aided Tomographic (CAT) x-ray data. This data is usually a series of two dimensional projections of a three dimensional volume. The process of converting this projection data back into a volume is called *tomographic reconstruction*.[1] Once a volume is tomographically reconstructed it can be visualized using volume rendering techniques.[5, 7, 13, 15, 16, 17]

These two operations have traditionally been decoupled, being handled by two separate algorithms. It is, however, highly beneficial to view these two operations as having the same mathematical and algorithmic form. Traditional volume rendering techniques can be reformulated into equivalent algorithms using hardware texture mapping and summing buffer. Similarly, the Filtered Back Projection CT algorithm can be reformulated into an algorithm which also uses texture mapping in combination with an accumulation or summing buffer.

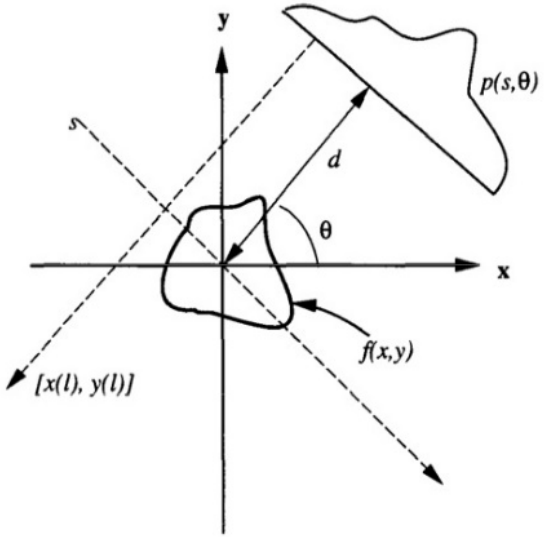


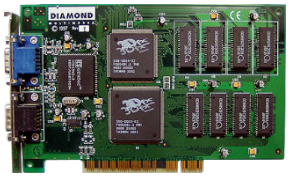
Figure 1: The Radon transform represents a generalized line integral projection of a 2-D (or 3-D) function $f(x, y, z)$ onto a line or plane.

der and reconstruct volumes at rates of 100 to 1000 times faster than CPU based techniques.

2 Background: The Radon and Inverse Radon Transform

We begin by developing the mathematical basis of volume rendering and reconstruction. The most fundamental of which is the Radon

The evolutions of the GPU



1st generation: Voodoo 3dfx (1996)

2nd Generation:
GeForce 256/Radeon 7500 (1998)

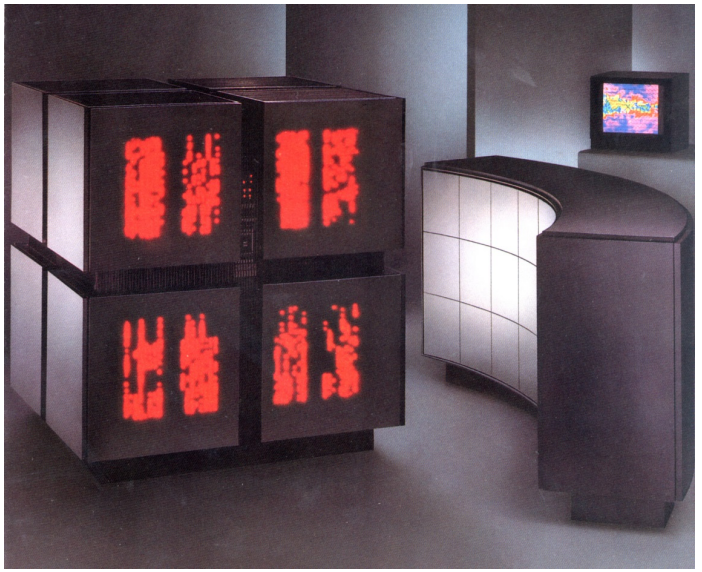
3rd Generation: GeForce3/Radeon 8500 (2001).
The first GPU to allow a limited programmability in the vertex pipeline.

4th Generation: Radeon 9700/GeForce FX (2002):
The first generation of "fully-programmable" graphics cards.

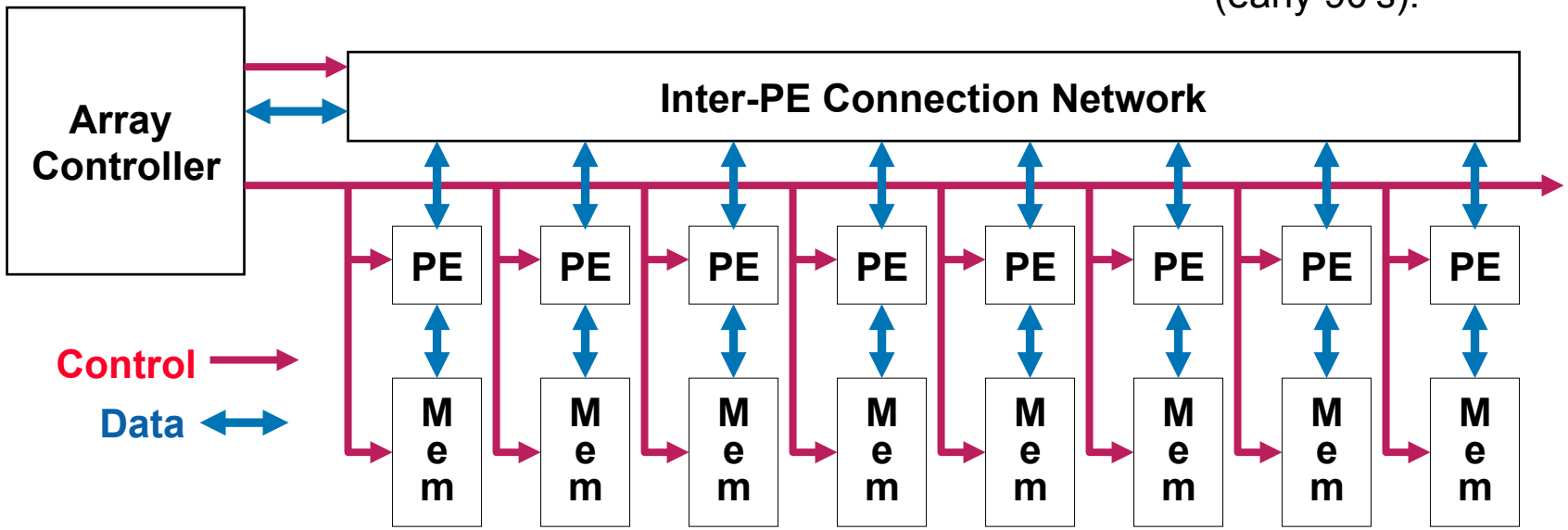
5th Generation: GeForce 8800/HD2900 (2006) and the birth of CUDA

Understanding GPGPU programming: SIMD Architecture

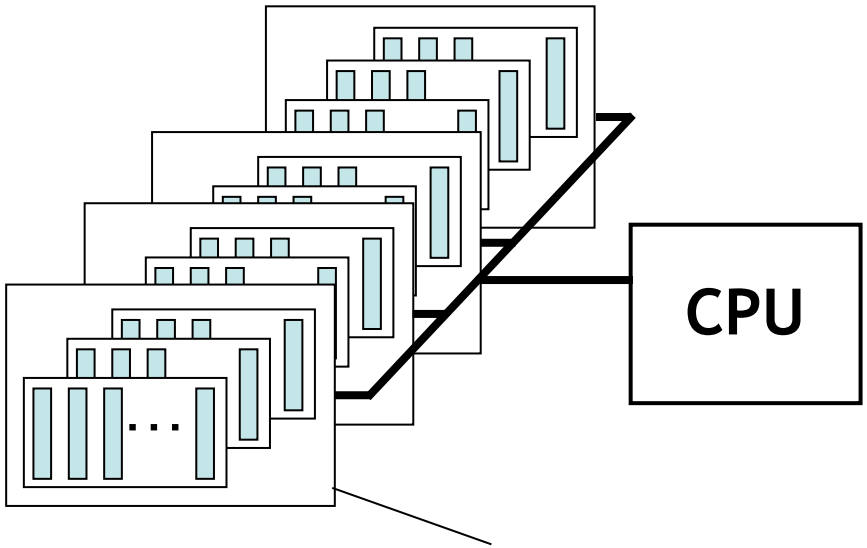
- Single Instruction Multiple Data (SIMD)
- Central controller broadcasts instructions to multiple processing elements (PEs)
  - Only requires one controller for whole array
  - Only requires storage for one copy of program
  - All computations fully synchronized



Thinking Machines Corp CM-200 (early 90's).



GPU Platform Model



- The GPUs are driven by a CPU which …
  - Manages the code to execute on the GPUs
  - Maintains a queue of kernels to execute
  - Manages memory on the GPU and movement between the CPU and the GPU

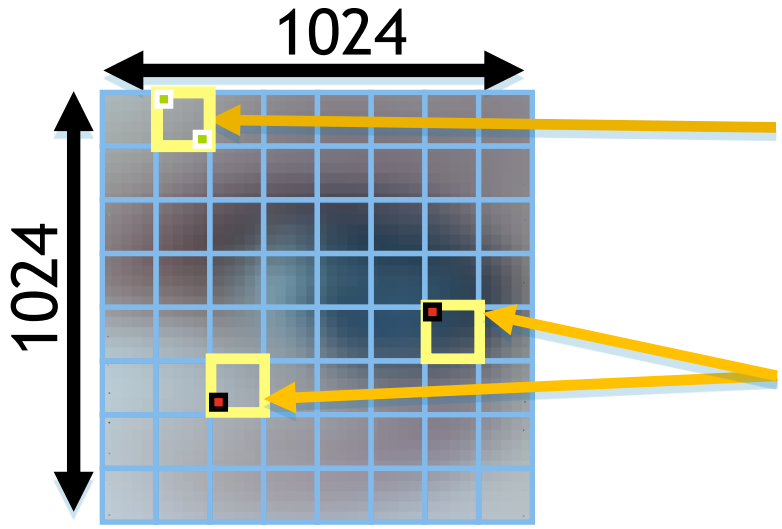
Kernel Parallelism

- Kernel Parallelism:
  - Implement data parallel problems:
    - Define an abstract index space that spans the problem domain.
    - Data structures in the problem are aligned to this index space.
    - Tasks (e.g. work-items in OpenCL or “threads” in CUDA) operate on these data structures for each point in the index space.
- This approach was popularized for graphics applications where the index space mapped onto the pixels in an image. Since 2006, It’s been extended to General Purpose GPU (GPGPU) programming.

Note: This is basically a fine grained extreme form of the SPMD pattern.

An N-dimensional domain of work-items

- Global Dimensions:
  - 1024x1024 (whole problem space)
- Local Dimensions:
  - 128x128 (**work-group**, executes together)



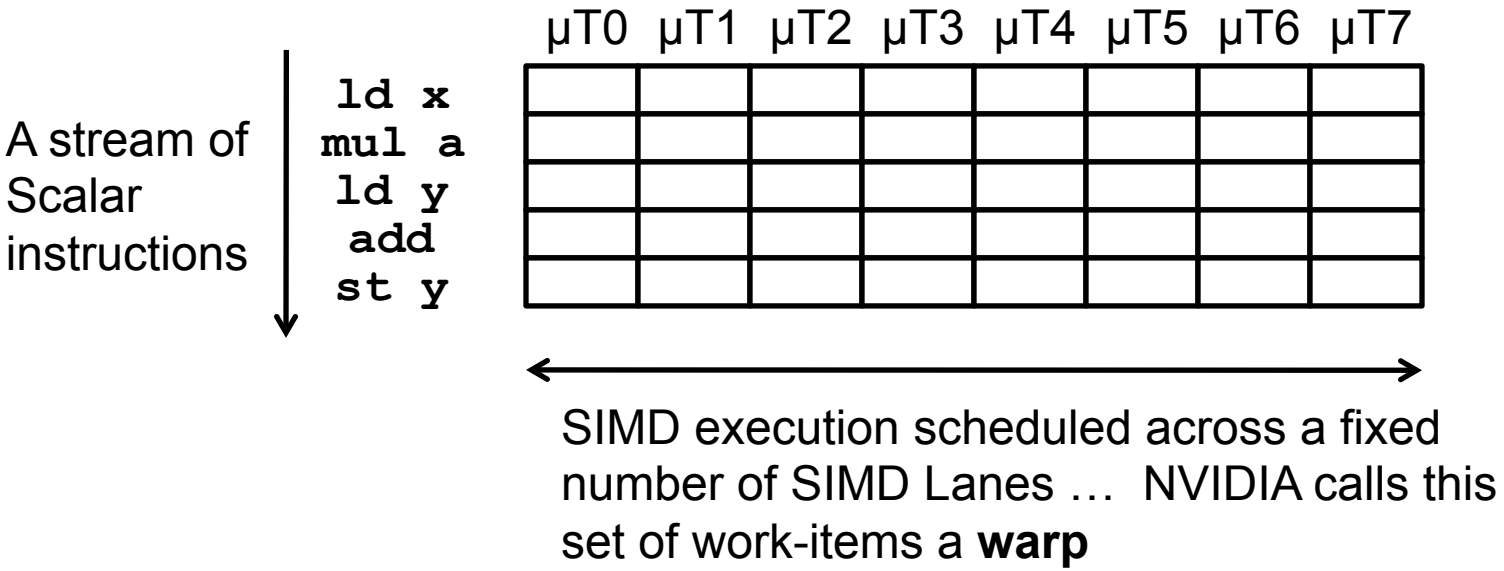
Synchronization between work-items possible only within work-groups: barriers and memory fences

Cannot synchronize between work-groups within a kernel

- Choose the dimensions that are "best" for your algorithm

SIMT: Single Instruction, Multiple Thread

- SIMT model: Individual scalar instruction streams (OpenCL **work-item or CUDA threads)** are grouped together for SIMD execution on hardware



Slide thanks to Krste Asanovic

What is an NVIDIA warp?

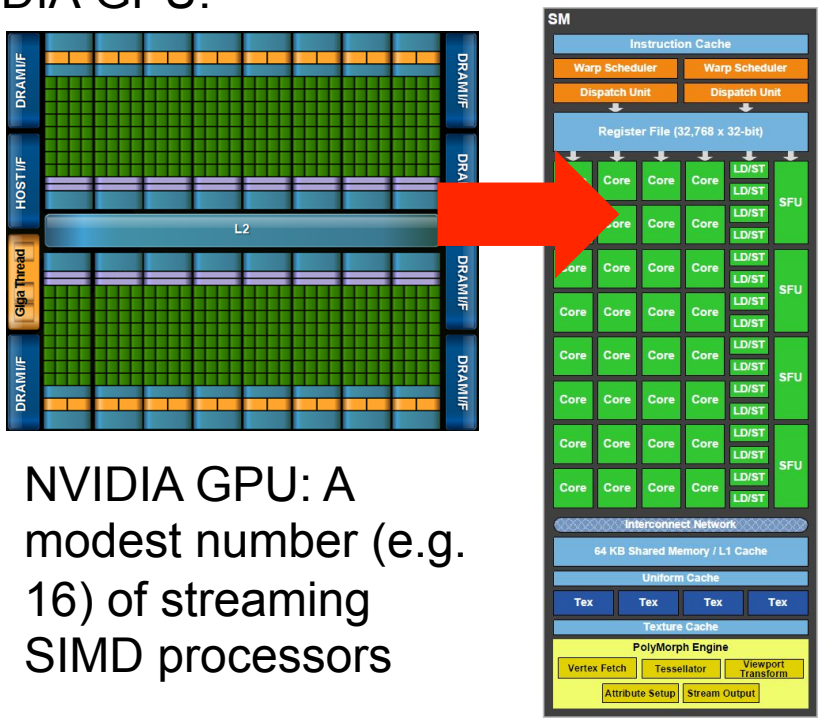
- A group of 32 work-items that execute simultaneously
  - Execution hardware is most efficiently utilized when all work-items in a warp execute instructions from the same PC.
  - Identifiable uniquely by dividing the work-item ID by 32

- In practical terms think or a warp as the minimum granularity of efficient SIMD execution, and the maximum hardware SIMD width in an NVIDIA GPU.
  - Note: the corresponding concept on an AMD GPU is a wavefront .. With 64 work-items in a wavefront.
  - The corresponding concept on a Cpu's vector unit is the number of lanes on the SIMD unit.

Mapping OpenCL to Nvidia GPUs

- OpenCL is designed to be functionally forgiving
  - First priority: make things work. Second: get performance.
- However, to get good performance, one must understand how OpenCL maps onto the hardware. For example, consider an NVIDIA GPU.

- Work Groups:
  - Each Work Group is scheduled onto a Streaming SIMD processor
  - Peak efficiency requires multiple work groups per Streaming SIMD processor
- Warps:
  - A work group is broken down into warps that execute together.
  - A SIMD instruction acts on a "warp"
  - The NVIDIA Warp width is 32 elements: ***LOGICAL*** SIMD width
- Work-items:
  - each work-item is a SIMD vector lane and runs on the processing element within a Streaming SIMD processor



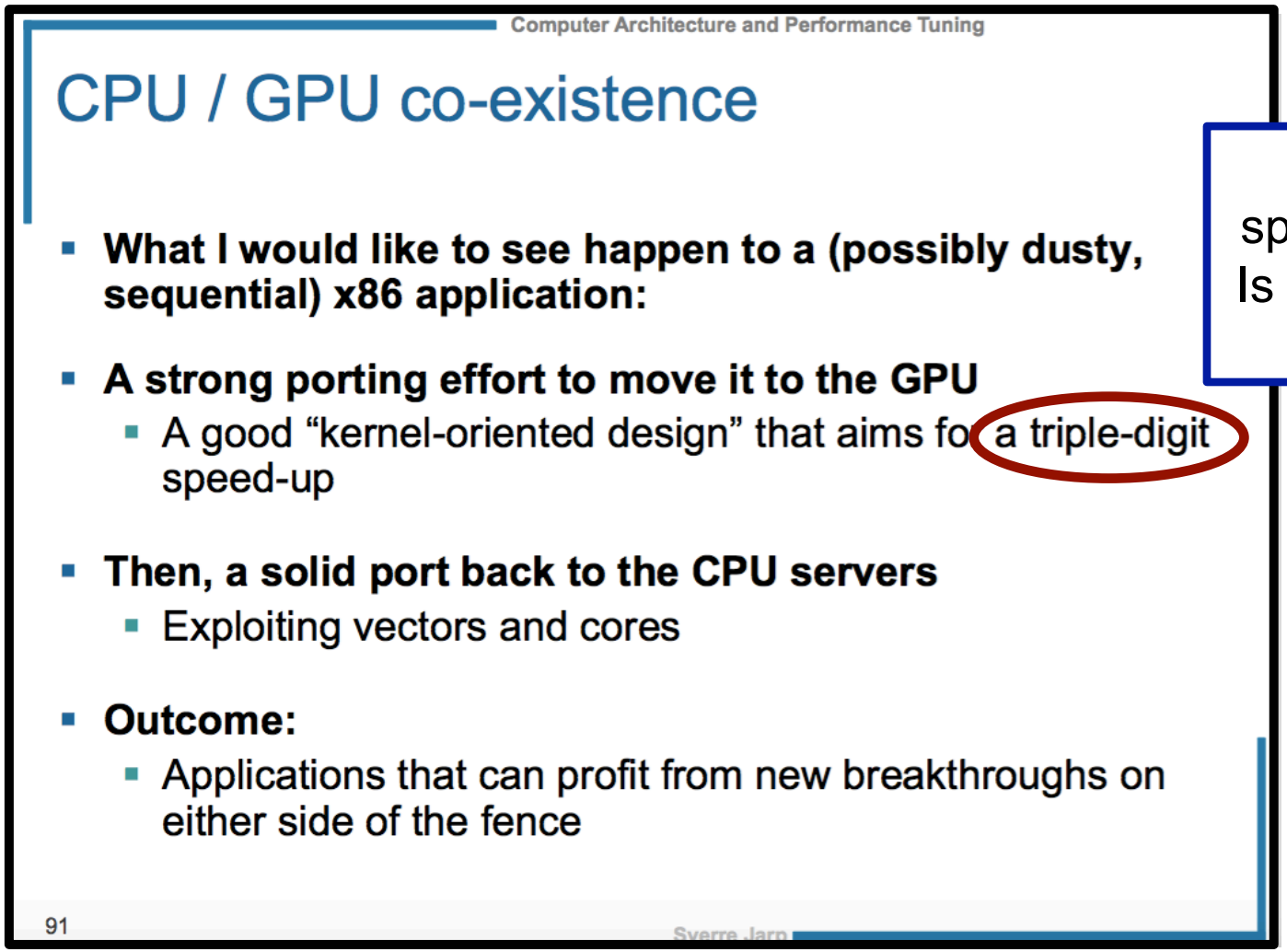
NVIDIA GPU: A modest number (e.g. 16) of streaming SIMD processors

A Streaming SIMD processor with 32 PE (hence, warp size is 32)

Outline

- The SIMT platform
- Understanding the GPU and GPGPU programming
- The 100X GPU/CPU speedup Myth
- The dream of performance portability
- The future of the GPU

100X speedups from GPUS: a common myth



Triple digit speedups? Really? Is this a reasonable goal?

Source: a great ESC'15 lecture by a smart person who made a mistake!!!

A high level view of performance

- Well optimized applications are either compute or bandwidth bounded

- **For bandwidth bound applications:**

  *Performance = Arch efficiency * Peak Bandwidth Capability*

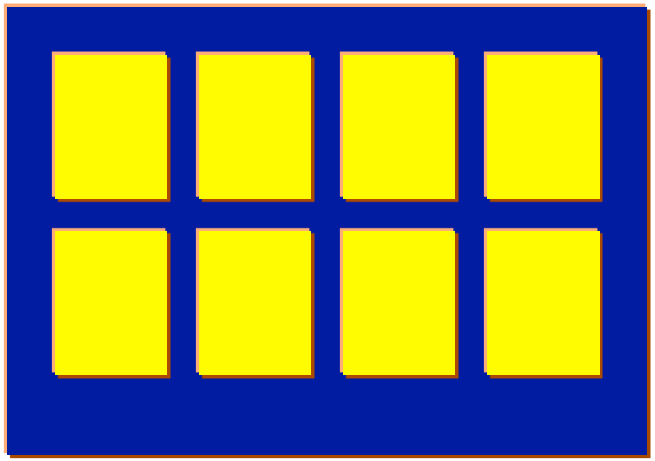
- **For compute bound applications:**

  *Performance = Arch efficiency * Peak Compute Capability*

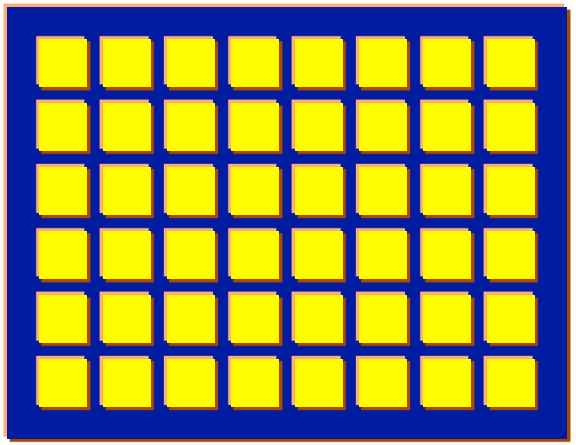
Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

Reasonable Speedup Expectations

- Chip A



- Chip B



$Perf_A = Eff_A * Peak_A(\text{Comp or BW})$

$Perf_B = Eff_B * Peak_B(\text{Comp or BW})$

$$Speedup\frac{B}{A} = \frac{Perf_B}{Perf_A} = \frac{Eff_B}{Eff_A} * \frac{Peak_A(Comp\_or\_BW)}{Peak_B(Comp\_or\_BW)}$$

Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

Speedup expectations for well optimized code: CPU vs. GPU

Core i7 960

- Four OoO Superscalar Cores, 3.2GHz
- Peak SP Flop: 102GF/s
- Peak BW: 30 GB/s

GTX 280

- 30 SMs (w/ 8 In-order SP each), 1.3GHz
- Peak SP Flop: 933GF/s*
- Peak BW: 141 GB/s

Assuming both Core i7 and GTX280 have the same efficiency:

	Max Speedup: GTX 280 over Core i7 960
Compute Bound Apps: (SP)	933/102 = 9.1x
Bandwidth Bound Apps:	141/30 = 4.7x

* 933GF/s assumes mul-add and the use of SFU every cycle on GPU

Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

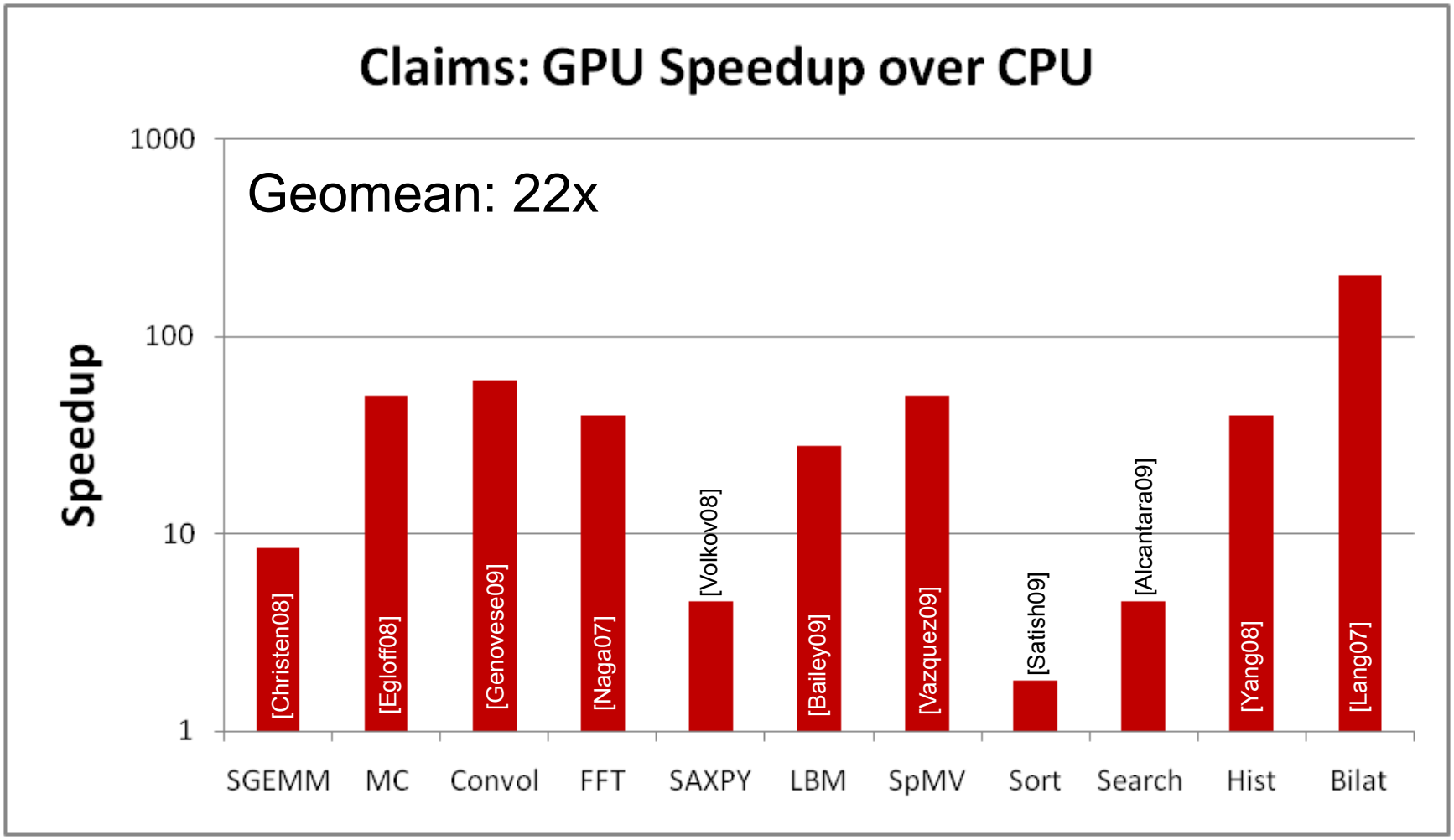
A fair comparison of CPUs and GPUs: Methodology

- Start with previously best published code / algorithm
- Validate claims by others
- Optimize BOTH CPU and GPU versions
- Collect and analysis performance data

Note: Only computation time on the CPU and GPU is measured. PCIe transfer time and host application time are not measured for GPU. Including such overhead will lower GPU performance

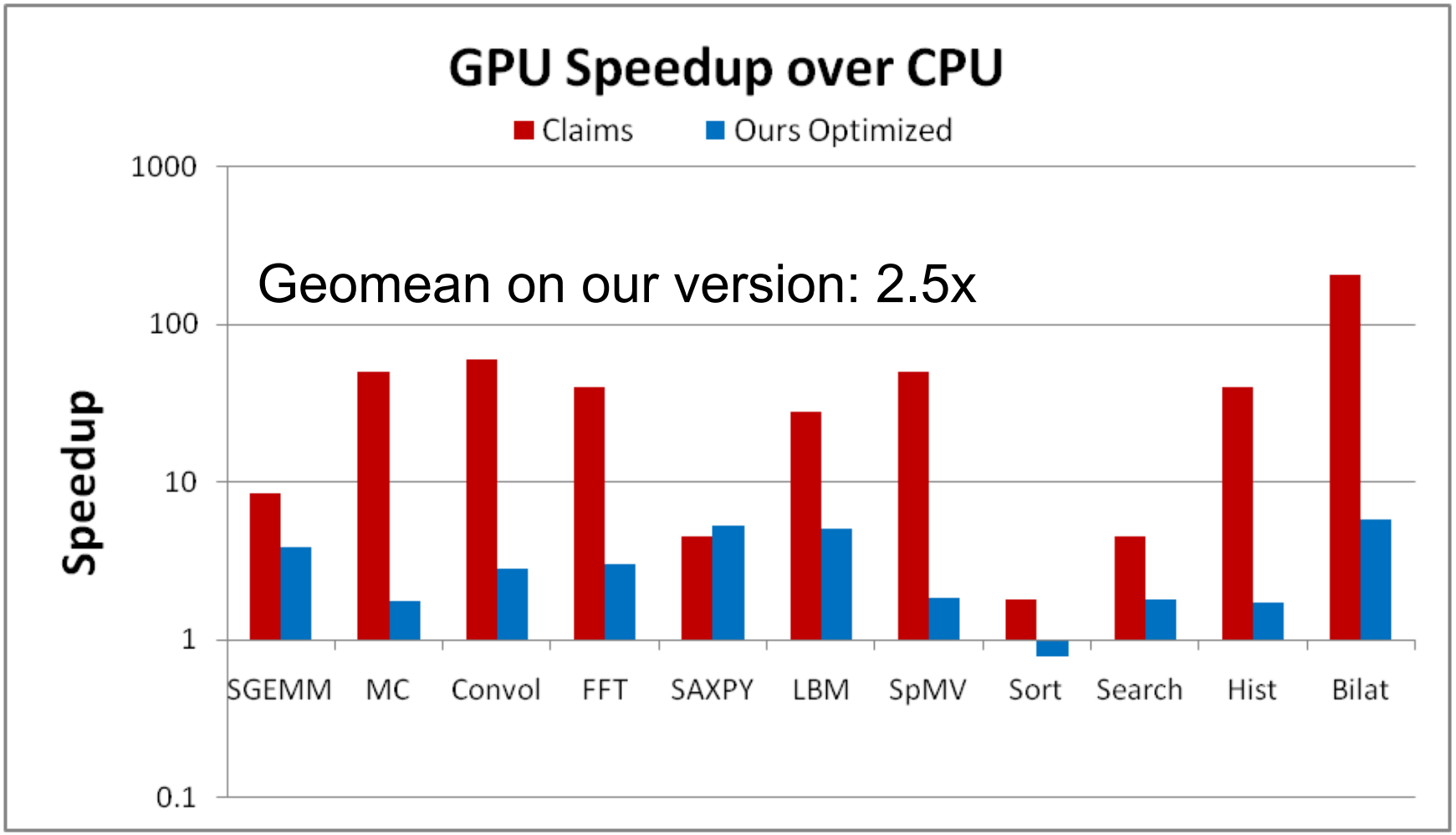
Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

What was claimed



Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

What we measured

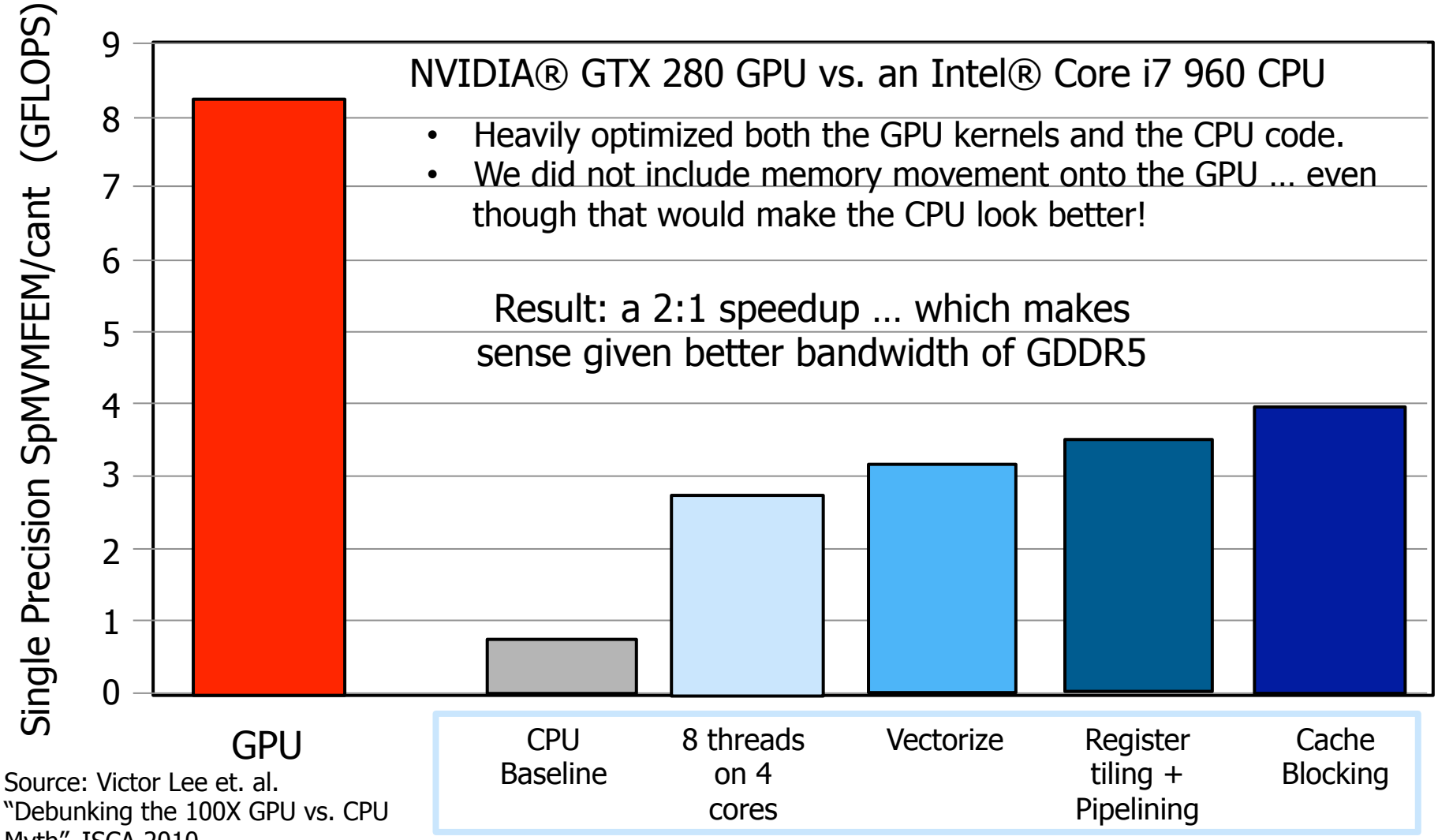


Apps.	SGEMM	MC	Conv	FFT	SAXPY	LBM	Solv	SpMV	GJK	Sort	RC	Search	Hist	Bilat
Core i7-960	94	0.8	1250	71.4	16.8	85	103	4.9	67	250	5	50	1517	83
GTX280	364	1.4	3500	213	88.8	426	52	9.1	1020	198	8.1	90	2583	475

Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

Sparse matrix vector product: GPU vs. CPU

- [Vazquez09]: reported a 51X speedup for an NVIDIA® GTX295 vs. a Core 2 Duo E8400 CPU … but they used an old CPU with unoptimized code



Source: Victor Lee et. al. "Debunking the 100X GPU vs. CPU Myth", ISCA 2010

*third party names are the property of their owners

Common Mistakes when comparing a CPU and a GPU

- Compare the latest GPU against an old CPU
- Highly optimized GPU code vs. unoptimized CPU code
  - I've seen numerous papers compare optimized CUDA vs. Matlab or python
- Parallel GPU code vs. serial, unvectorized CPU code.
- Ignore the GPU penalty of moving data across the PCI bus from the CPU to the GPU

GPUs are great and depending on the algorithm can show two to four fold speedups. But not 100+ … that's just irresponsible and should not be tolerated!!

Outline

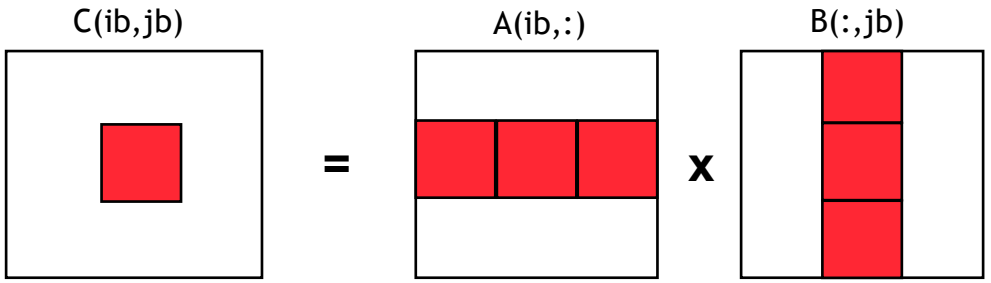
- The SIMT platform
- Understanding the GPU and GPGPU programming
- The 100X GPU/CPU speedup Myth
- The dream of performance portability
- The future of the GPU

Whining about performance Portability

- Do we have performance portability today?
  - NO: Even in the “serial world” programs routinely deliver single digit efficiencies.
  - If the goal is a large fraction of peak performance, you will need to specialize code for the platform.
- However there is a pretty darn good performance portable language. It’s called OpenCL

Portable performance: dense matrix multiplication

```
void mat_mul(int N, float *A, float *B, float *C)
{
 int i, j, k;
 int NB=N/block_size; // assume N%block_size=0
 for (ib = 0; ib < NB; ib++)
   for (jb = 0; jb < NB; jb++)
     for (kb = 0; kb < NB; kb++)
       sgemm(C, A, B, …)    // C_ib,jb = A_ib,kb * B_kb,jb
}
```



Transform the basic serial matrix multiply into multiplication over blocks

Note: sgemm is the name of the level three BLAS routine to multiply two matrices

Blocked matrix multiply: kernel

```
#define blksz 16
__kernel void mmul(
            const unsigned int N,
            __global float* A,
            __global float* B,
            __global float* C,
            __local  float* Awrk,
            __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;

  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
      //Each work-item loads a single element of the two
      //blocks which are shared with the entire work-group

      Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
      Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      #pragma unroll
      for(kloc=0; kloc<blksz; kloc++)
   Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

      barrier(CLK_LOCAL_MEM_FENCE);

      Abase += Ainc;    Bbase += Binc;
   }
   C[j*N+i] = Ctmp;
}
```

Blocked matrix multiply: kernel

It's getting the indices right that makes this hard

```
#define blksz 16
__kernel void mmul(
            const unsigned int N,
            __global float* A,
            __global float* B,
            __global float* C,
            __local  float* Awrk,
            __local  float* Bwrk)
{
  int kloc, Kblk;
  float Ctmp=0.0f;

  //  compute element C(i,j)
  int i = get_global_id(0);
  int j = get_global_id(1);

  // Element C(i,j) is in block C(Iblk,Jblk)
  int Iblk = get_group_id(0);
  int Jblk = get_group_id(1);

  // C(i,j) is element C(iloc, jloc)
  //  of block C(Iblk, Jblk)
  int iloc = get_local_id(0);
  int jloc = get_local_id(1);
  int Num_BLK = N/blksz;
```

Load A and B blocks, wait for all work-items to finish

```
  // upper-left-corner and inc for A and B
  int Abase = Iblk*N*blksz;   int Ainc  = blksz;
  int Bbase = Jblk*blksz;     int Binc  = blksz*N;

 // C(Iblk,Jblk) = (sum over Kblk)
A(Iblk,Kblk)*B(Kblk,Jblk)
  for (Kblk = 0;  Kblk<Num_BLK;  Kblk++)
  {   //Load A(Iblk,Kblk) and B(Kblk,Jblk).
     //Each work-item loads a single element of the two
     //blocks which are shared with the entire work-group

     Awrk[jloc*blksz+iloc] = A[Abase+jloc*N+iloc];
     Bwrk[jloc*blksz+iloc] = B[Bbase+jloc*N+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);

     #pragma unroll
     for(kloc=0; kloc<blksz; kloc++)
  Ctmp+=Awrk[jloc*blksz+kloc]*Bwrk[kloc*blksz+iloc];

     barrier(CLK_LOCAL_MEM_FENCE);

     Abase += Ainc;     Bbase += Binc;
  }
  C[j*N+i] = Ctmp;
}
```

Wait for everyone to finish before going to next iteration of Kblk loop.

Matrix multiplication … Portable Performance (in MFLOPS)

- Single Precision matrix multiplication (order 1000 matrices)

Case	CPU	Xeon Phi	Core i7, HD Graphics	NVIDIA Tesla
Sequential C (compiled /O3)	224.4		1221.5	
C(i,j) per work-item, all global	841.5	13591		3721
C row per work-item, all global	869.1	4418		4196
C row per work-item, A row private	1038.4	24403		8584
C row per work-item, A private, B local	3984.2	5041		8182
Block oriented approach using local (blksz=16)	12271.3	74051 (126322*)	38348 (53687*)	119305
Block oriented approach using local (blksz=32)	16268.8			

Xeon Phi SE10P, CL_CONFIG_MIC_DEVICE_2MB_POOL_INIT_SIZE_MB = 4 MB
* The comp was run twice and only the second time is reported (hides cost of memory movement.

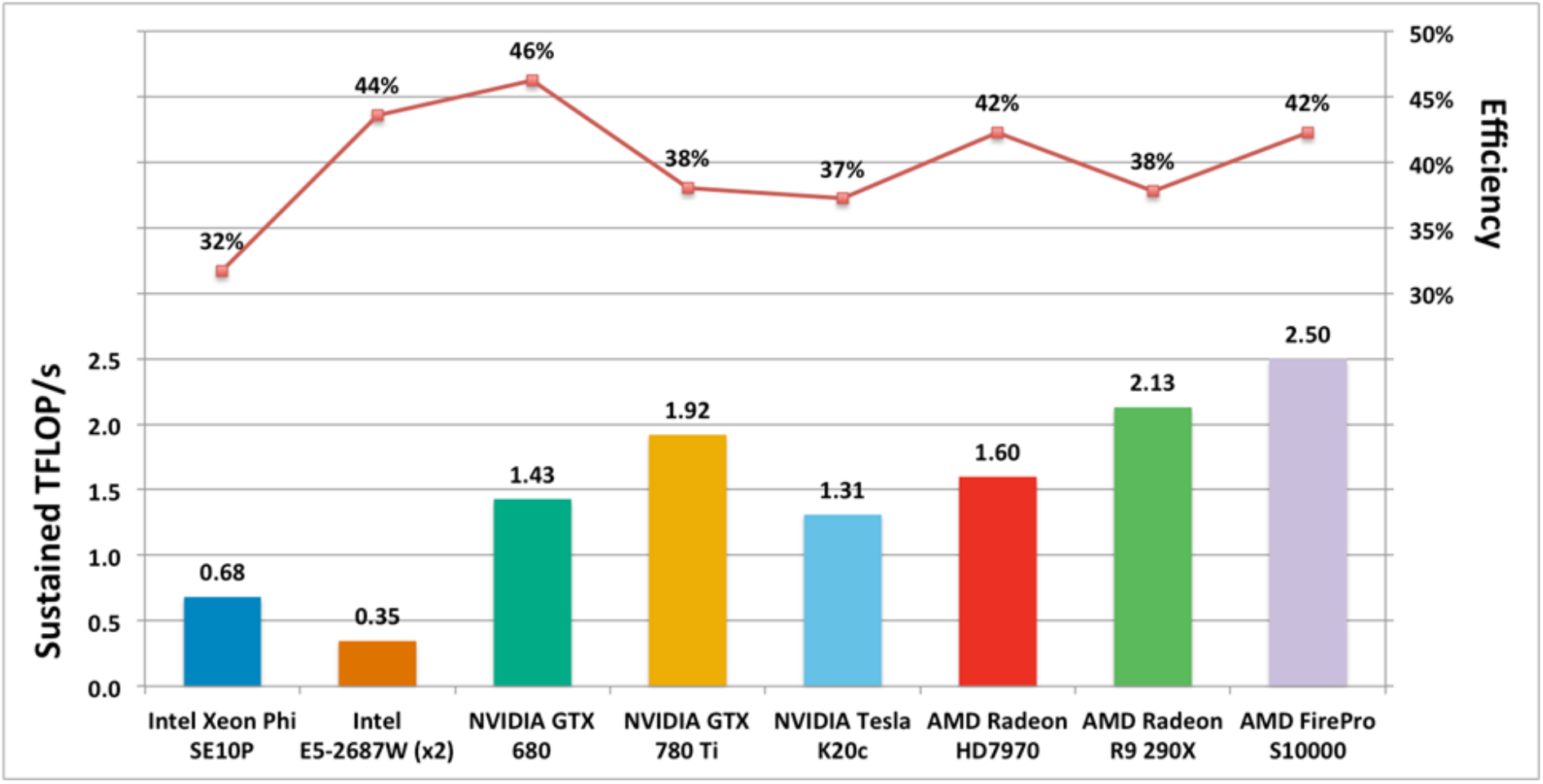
Intel® Core™ i5-2520M CPU @2.5 GHz (dual core) Windows 7 64 bit OS, Intel compiler 64 bit version 13.1.1.171, OpenCL SDK 2013, MKL 11.0 update 3.

Intel Core i7-4850HQ @ 2.3 GHz which has an Intel HD Graphics 5200 w/ high speed memory. ICC 2013 sp1 update 2.
Tesla® M2090 GPU from NVIDIA® with a max of 16 compute units, 512 PEs

These are not official benchmark results. You may observe completely different results should you run these tests on your own system.

BUDE: Bristol University Docking Engine

One program running well on a wide range of platforms



Whining about performance Portability

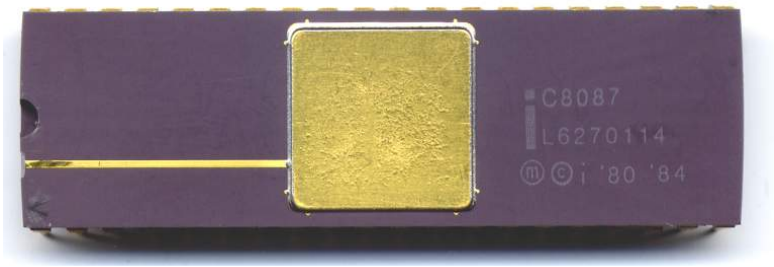
- Do we have performance portability today?
  - NO: Even in the “serial world” programs routinely deliver single digit efficiencies.
  - If the goal is a large fraction of peak performance, you will need to specialize code for the platform.
- However there is a pretty darn good performance portable language. It’s called OpenCL
- But this focus on mythical “Performance Portability” misses the point. The issue is “maintainability”.
  - You must be able maintain a body of code that will live for many years over many different systems.
  - Having a common code base using a portable programming environment … even if you must fill the code with if-defs or have architecture specific versions of key kernels … is the only way to support maintainability.

Outline

- The SIMT platform
- Understanding the GPU and GPGPU programming
- The 100X GPU/CPU speedup Myth
- The dream of performance portability
- The future of the GPU

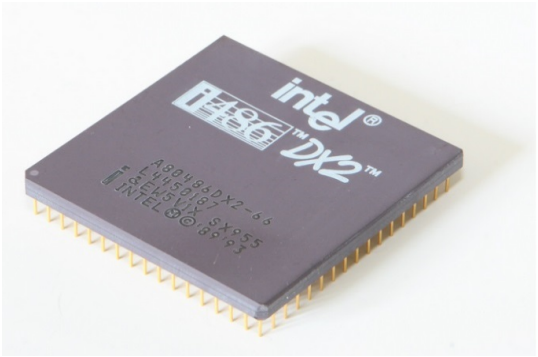
Coprocessors to accelerate flops

- The coprocessor: 8087 introduced in 1980.
  - The first x87 floating point coprocessor for the 8086 line of microprocessors.
  - Performance enhancements: 20% to 500%, depending on the workload.

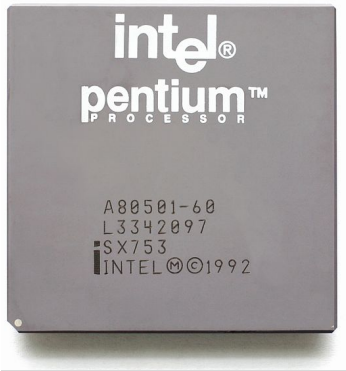


- Related Standards:
  - Partnership between industry and academia led to IEEE 754 …. The most important standard in the history of HPC. IEEE 754 first supported by x87.

- Intel® 80486DX, Pentium®, and later processors include floating-point functionality in the CPU … the end of the line for the X87 processors.



Intel® 486DX2™ processor, March 1992.



Intel® Pentium™ processor, Spring 1993.

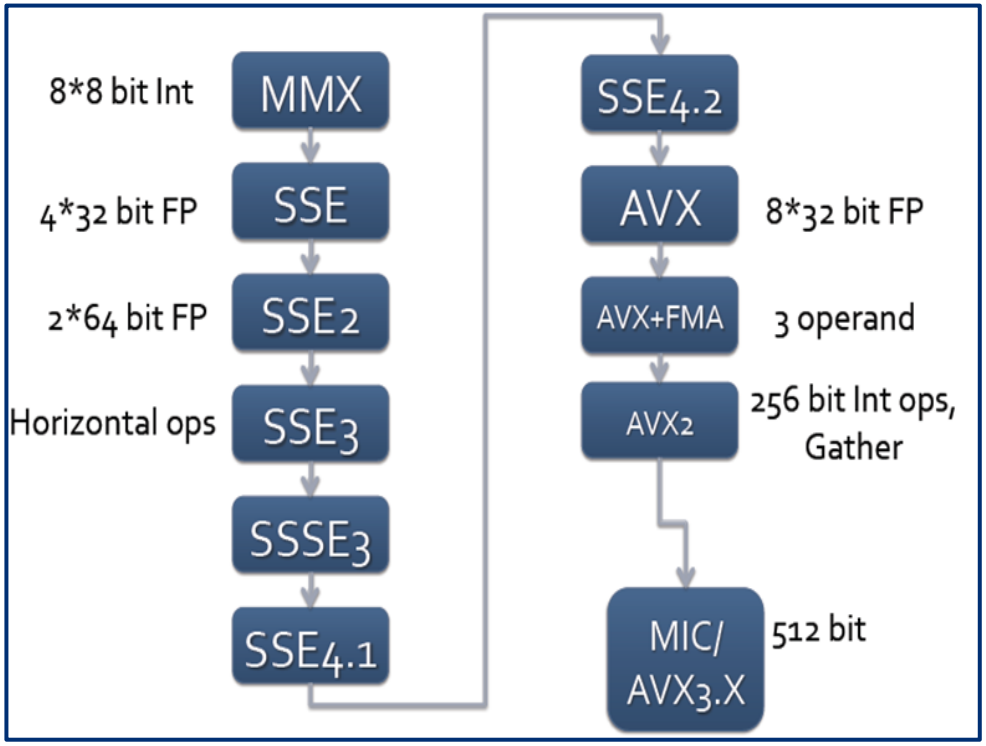
Intel C8087 Math Coprocessor. Source: http://www.cpu-collection.de/?l0=i&i=1698&sd=1

Vector processing accelerators

- Vector Coprocessor:
  - Vector co-processor (from Sky computer) for Intel iPSC/2 MPP (~1987)
  - Floating point systems array processors (late 80s's)

Absorbed into the CPU

The Intel® i860 processor (early 90's) with integrated vector instructions.

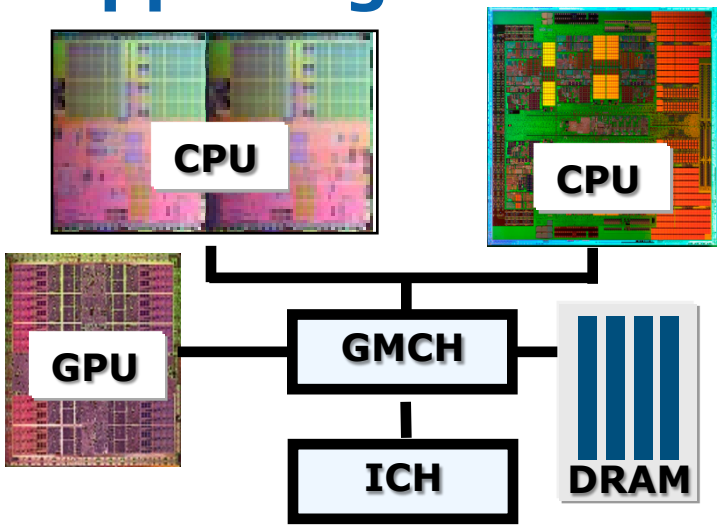


And now vector instructions are a ubiquitous element of CPUs.

*third party names are the property of their owners

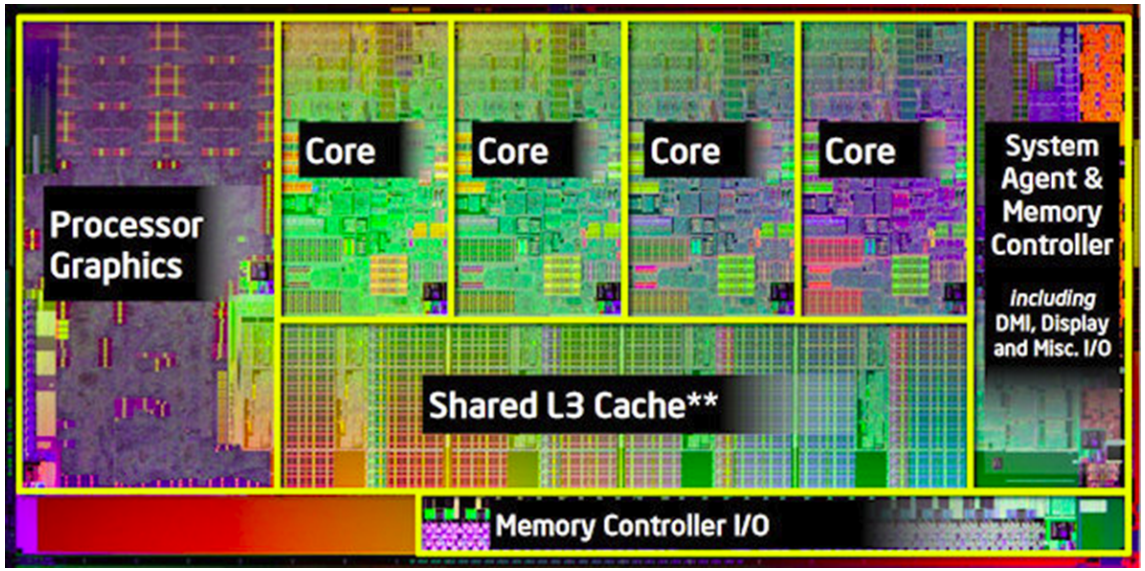
Source: http://www.cpu-collection.de/?l0=co&l1=Intel&l2=i860

The absorption of the GPU into the CPU is happening now



- A modern platform has:
  - CPU(s)
  - GPU(s)
  - DSP processors
  - … other?

- Current designs put this functionality onto a single chip … mitigates the PCIe bottleneck in GPGPU computing!



Intel® Core™ i5-2500K Desktop Processor (Sandy Bridge) Intel HD Graphics 3000 *(2011)*

GMCH = graphics memory control hub,
ICH = Input/output control hub

Conclusion

- The SIMT platform is here to stay … though the GPU is likely to move from a discrete card to an IP block on the CPU die.
- Performance benefits are significant (two to four times) but not silly (100+).
- The more interesting question …
  - Is the SIMT platform as a software abstraction better than multi-threading + vectorization?
  - The corporate CPU world says “no” but they are not the final judges of this fundamental programmability question. It’s up to applications programmers to decide.
  - So try both and see what you think … threads+vectorization or SIMT?