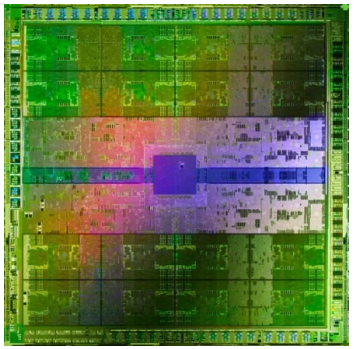
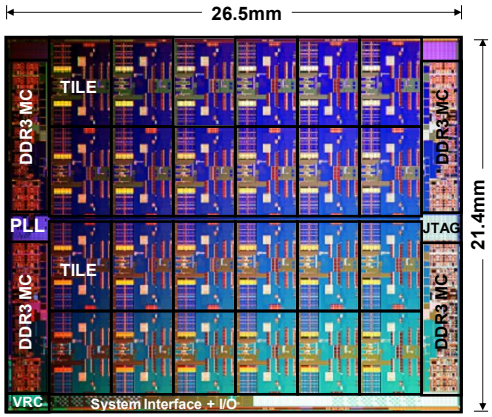
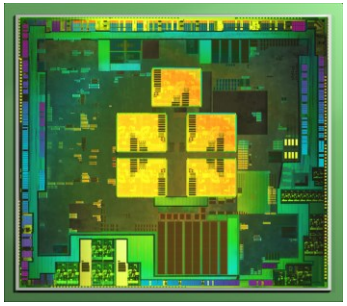


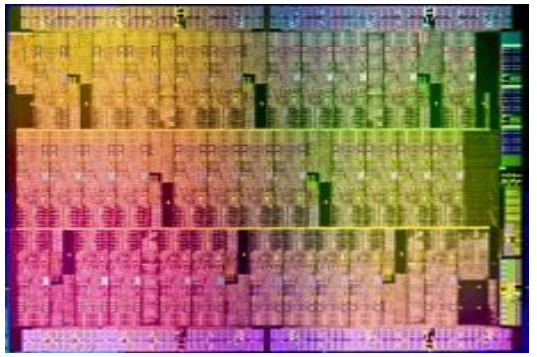
NVIDIA GTX 480 processor



Intel labs 48 core SCC processor



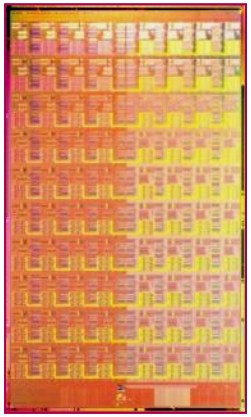
NVIDIA Tegra 3 (quad Arm Corex A9 cores + GPU)



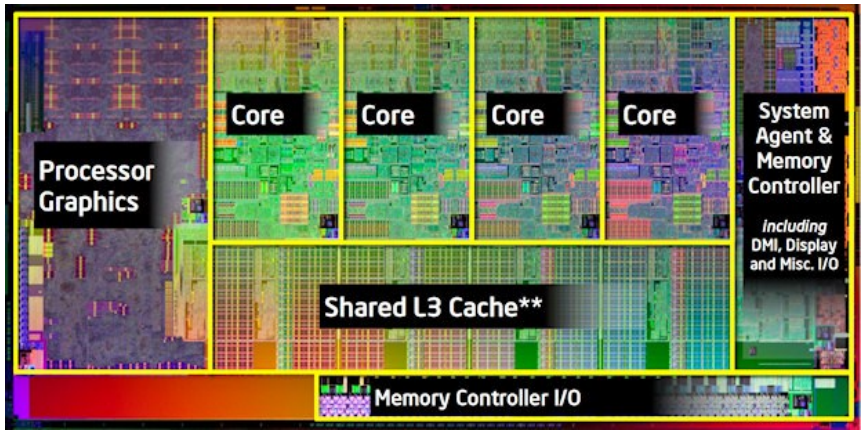
An Intel MIC processor

GPUs and the Heterogeneous programming problem

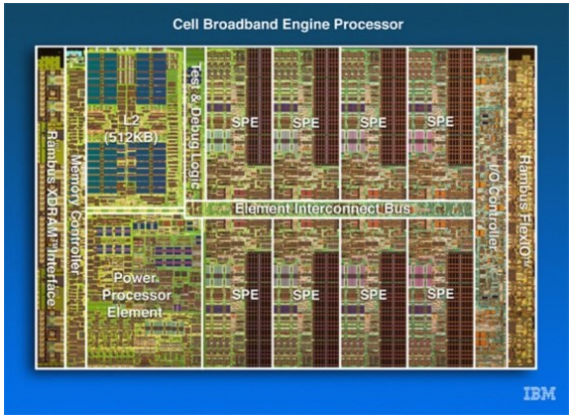
Tim Mattson (Intel Labs)



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

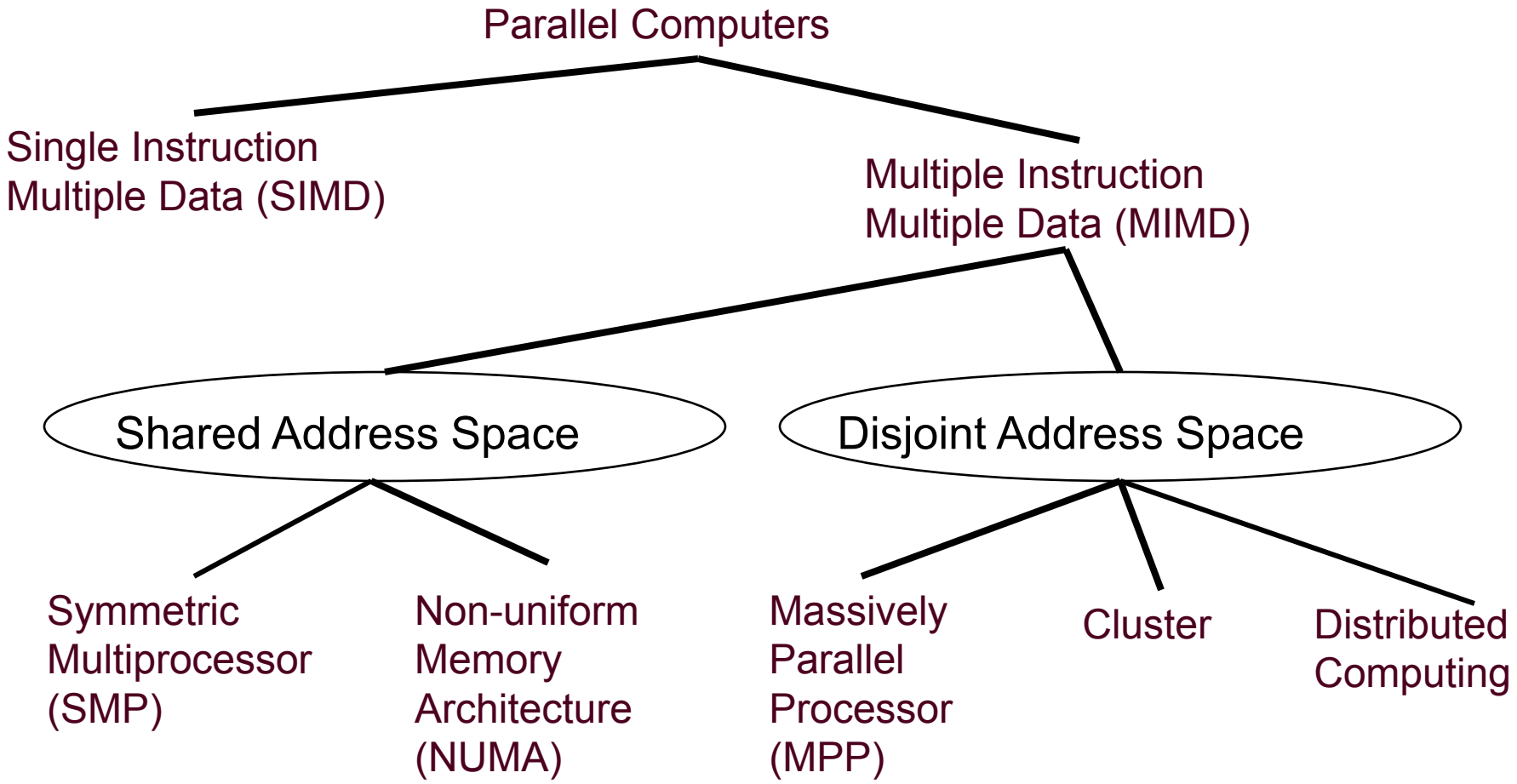
READ THIS … its very important

- 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.

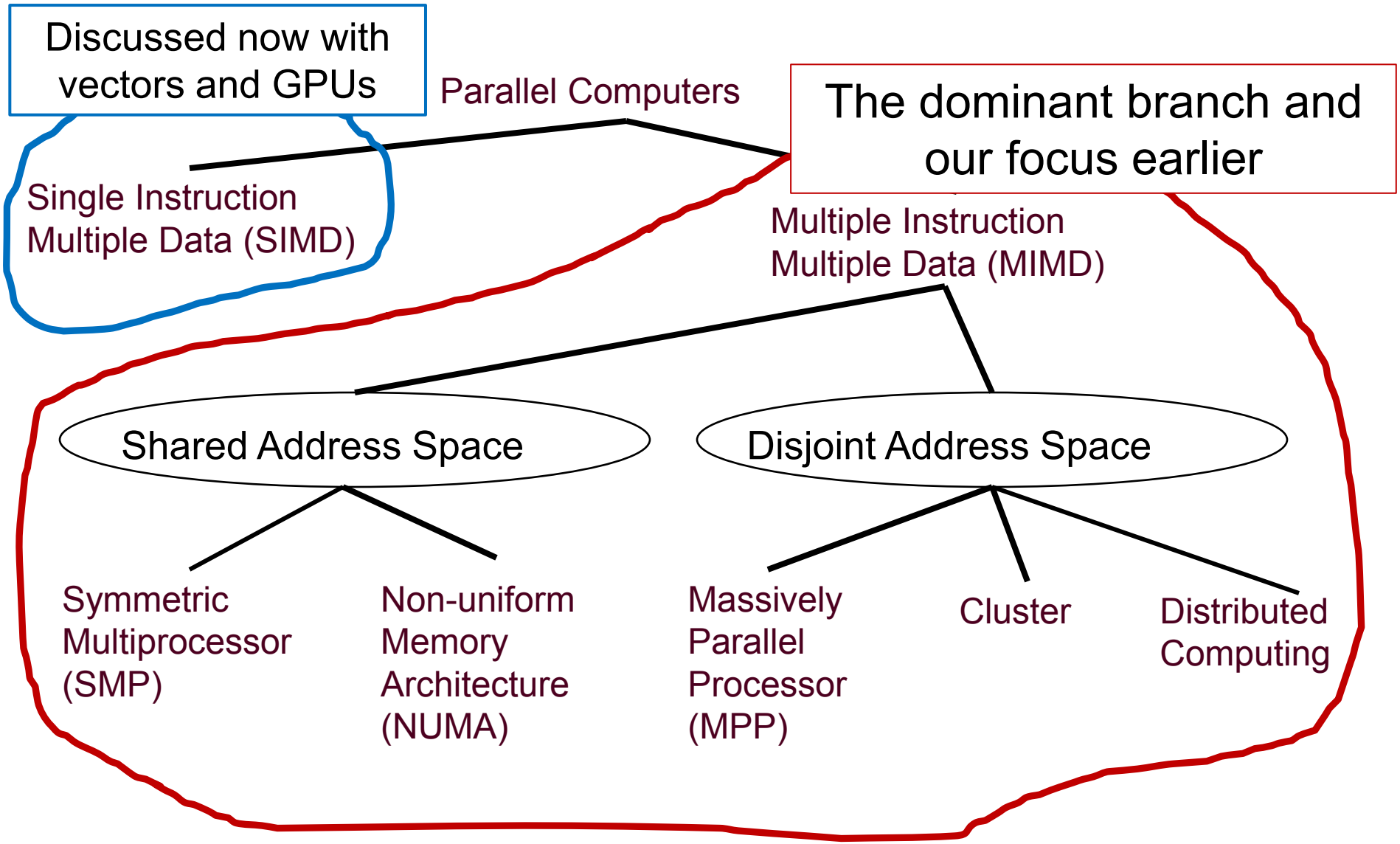
Slides marked with this symbol were produced-with Kurt Keutzer and his team for CS194 … A UC Berkeley course on Architecting parallel applications with Design Patterns.

Alt intro to GPU hardware

Hardware Architectures for High Performance Computing (HPC)

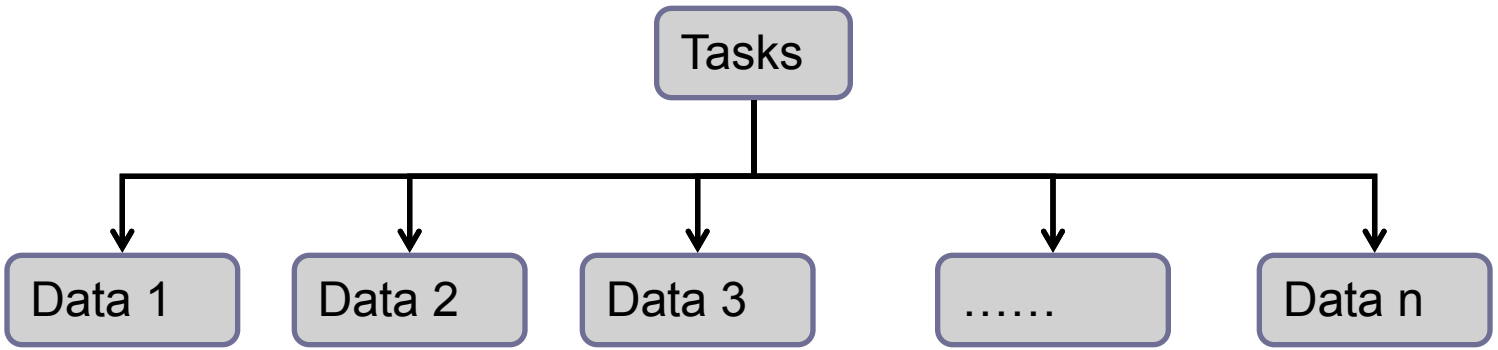


Hardware Architectures for High Performance Computing (HPC)

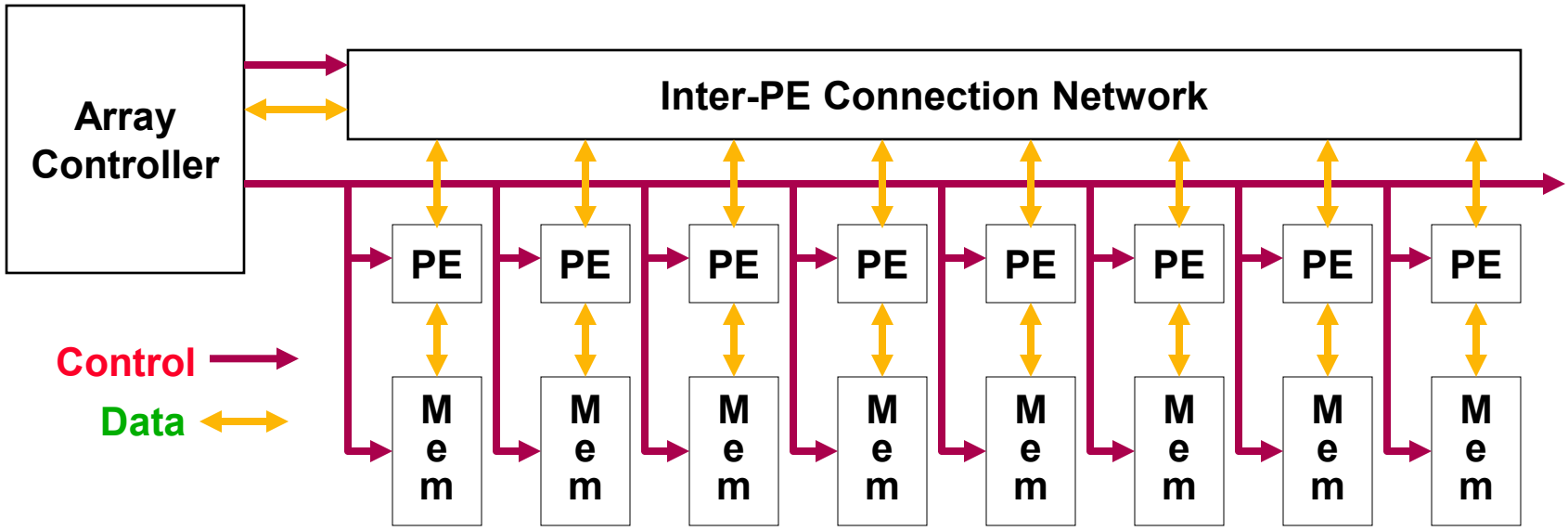


Data Parallelism Pattern

- Use when:
  - Your problem is defined in terms of collections of data elements operated on by a similar (if not identical) sequence of instructions; i.e. the concurrency is in the data.
- Solution
  - Define collections of data elements that can be updated in parallel.
  - Define computation as a sequence of collective operations applied together to each data element.



SIMD Architecture

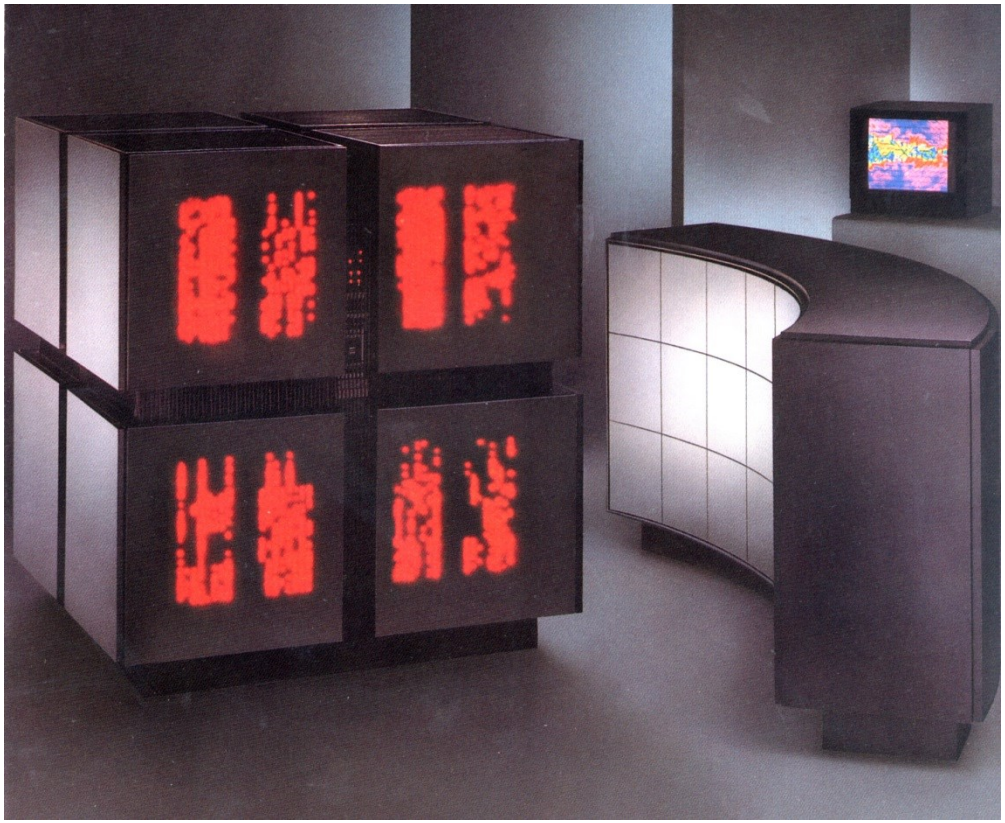


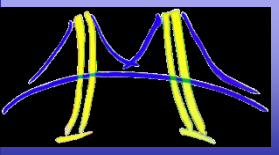
- Single Instruction Multiple Data (SIMD)
- Central controller broadcasts instructions to multiple processing elements (PEs)
  - Only requires one controller for whole array
  - Only requires storage for one copy of program
  - All computations fully synchronized

A classic SIMD Massively Parallel Processor: Thinking machines CM-200:

- **Connection machine CM200 … late 80's early 90's**

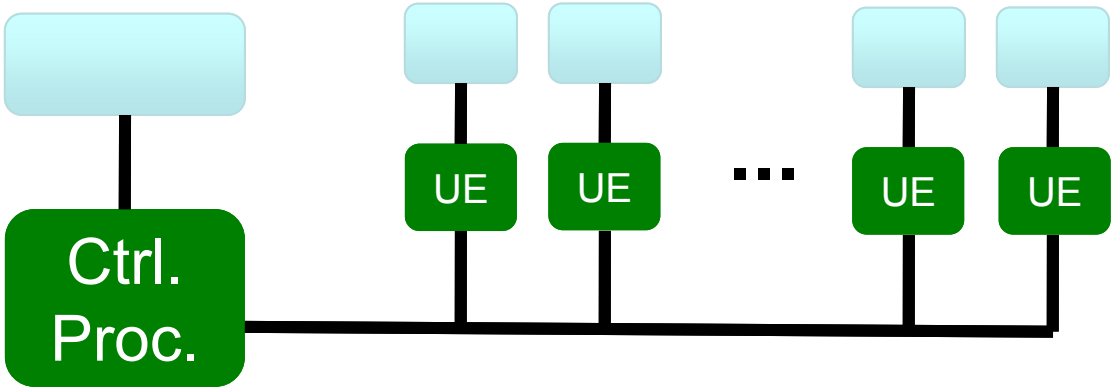
  - A Workstation hosted SIMD machine.
  - A node consists of a two processor chip pair (32 PEs) and an optional floating point accelerator.
  - Topology --- The nodes are connected as a hypercube.
  - Performance --- peak performance of 40 GFLOPS for the largest CM-200 (65536 PEs) with floating point accelerators.
  - Scalability --- 2K, 4K, 8K, 16K, 32K or 64K processors. Machines may be partitioned



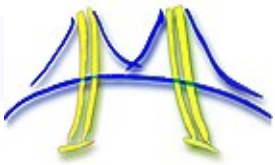


Modern Data-Parallel Machines

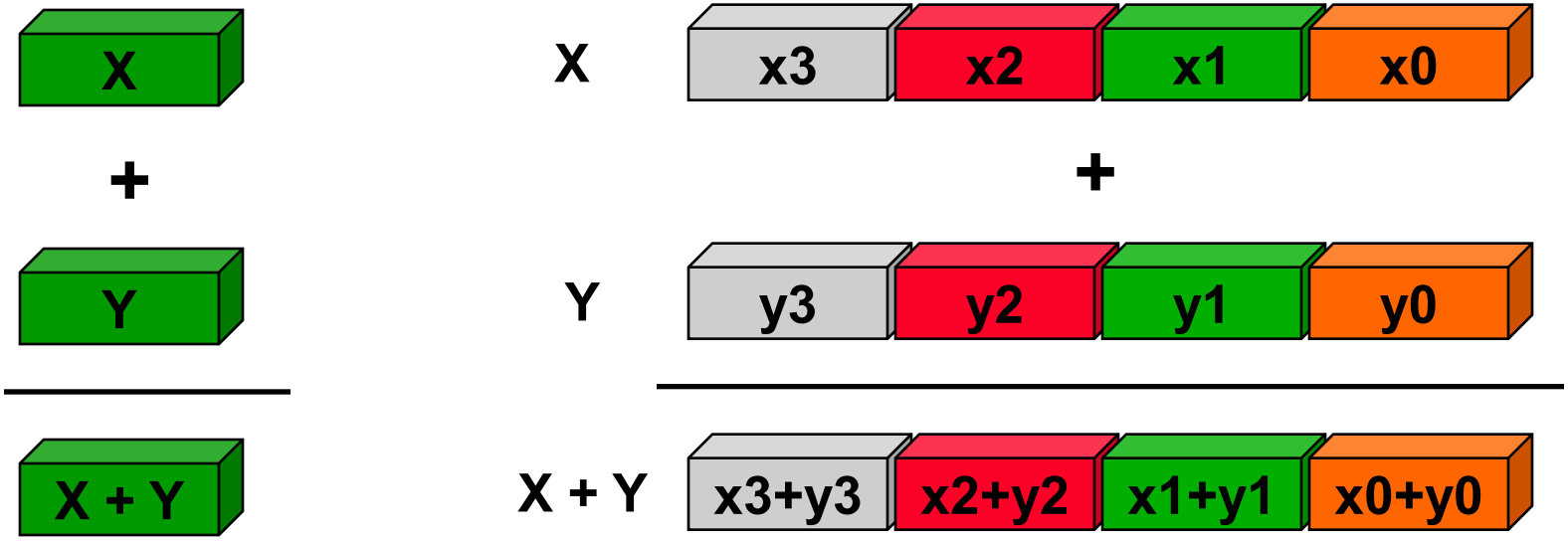
- SIMD in CPUs: The CPU pipeline is the "frontend", executing a sequential program and issuing commands to the SSE or AVX processor "array"
- SIMD in CPU-GPU systems: The CPU Host is the "frontend machine", issuing SIMD Kernel commands to the GPU "array" of Streaming Multiprocessors
- SIMD in GPUs: The Warp Scheduler issues commands to the SIMD arrays of Scalar Processors
- In none of these cases is the physical SIMD width (4-32) as large as the Connection machine (16K)



Pseudo SIMD: (Poor-Man's SIMD?)

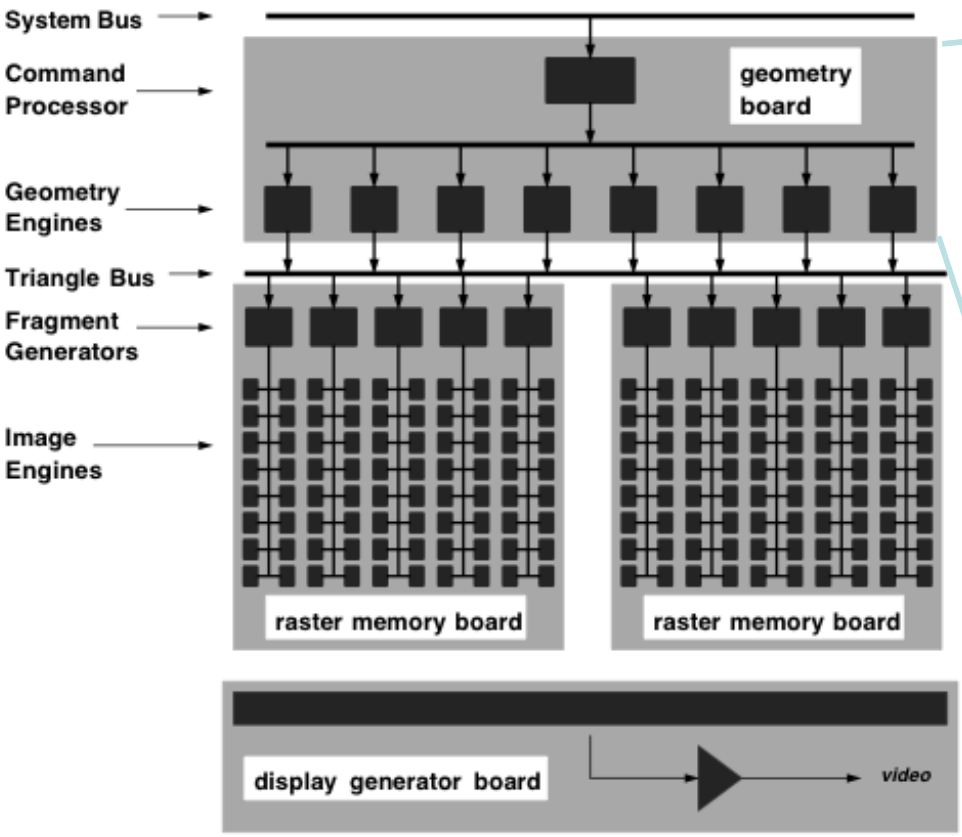


- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing (Intel)
  - with SSE / SSE2
  - one operation produces multiple results



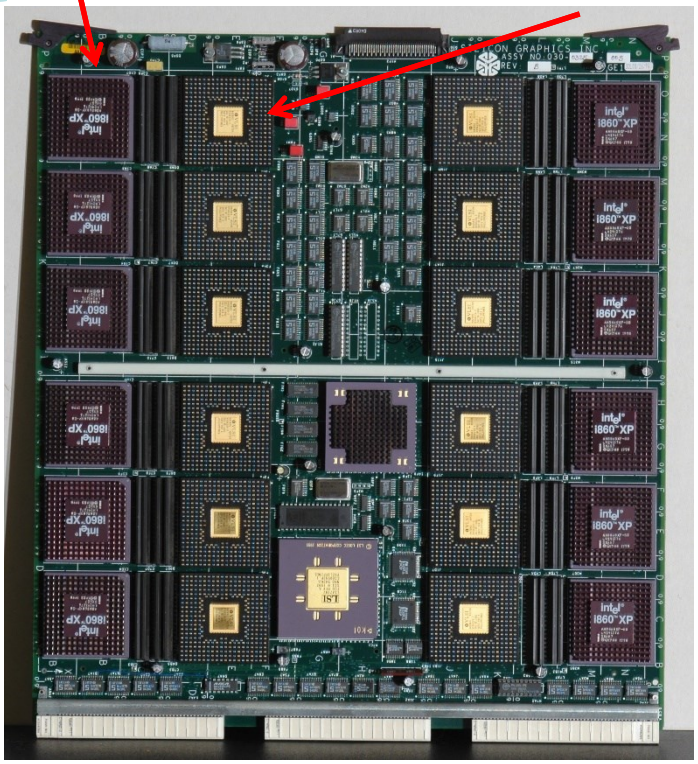
Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

High-end GPUs have historically been programmable



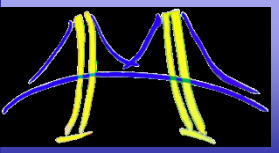
Intel i860 RISC CPU

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

- Graphics programming
  - OpenGL
  - DirectX
- General purpose applications on GPUs
  - It's been done since the mid-90s
  - Why hot now?
    1. Reasonable programming models
    2. Devices cost $300 instead of $3M

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.

The mathematical and algorithmic similarity of these two operations, when reformulated in terms of texture mapping and accumulation, is significant. It means that existing high performance computer graphics and imaging computers can be used to both render and reconstruct volumes at rates of 100 to 1000 times faster than CPU based techniques.

*2011 N. Shoreline Blvd., Mountain View, CA 94043

[1] The term tomographic reconstruction or Computed Tomography (CT)[12] is used to differentiate it from signal reconstruction: the rebuilding of a continuous function (signal) from a discrete sampling of that function.

0-8186-7067-3/95 $4.00 © 1995 IEEE

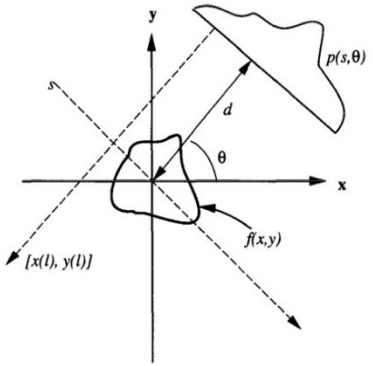


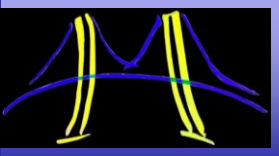
Figure 1: The Radon transform represents a generalized line integral projection of a 2-D (or 3-D) function $f(x, y, z)$ onto a line or plane.

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 transform and its inverse. We will show that volume rendering, as described in [5, 13, 15, 16, 17], is a generalized form of the Radon transform. Finally, we will demonstrate efficient hardware texture mapping based implementations of both volume rendering and it's inverse: volume reconstruction.

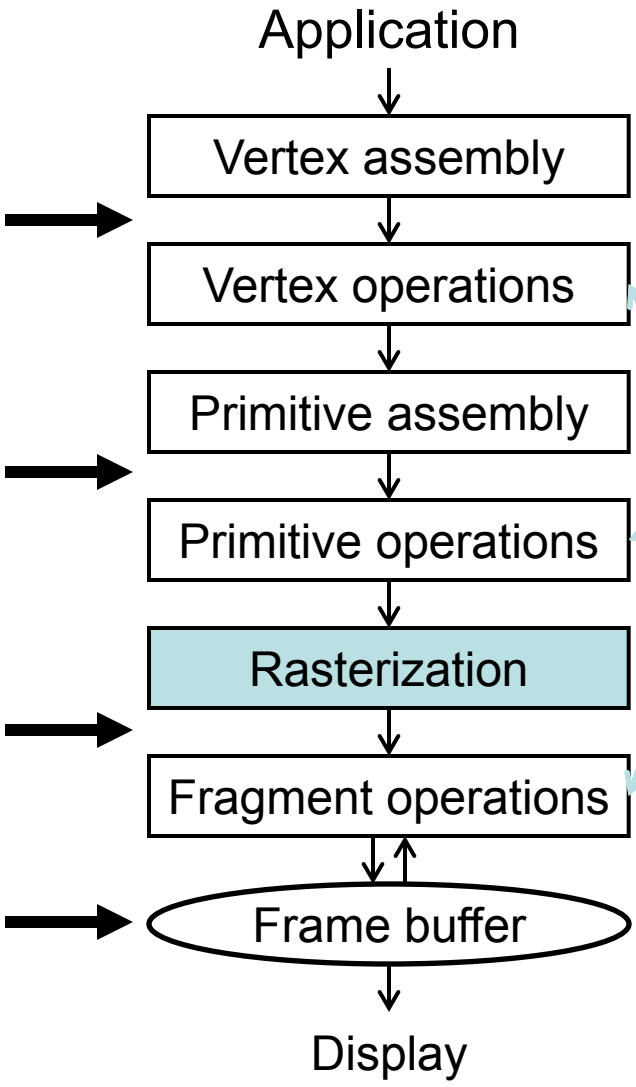
The Radon transform defines a mapping between the physical object space (x, y) and its projection space (s, θ), as illustrated in figure 1. The object is defined in a Cartesian coordinate system by $f(x, y)$, which describes the x-ray absorption or attenuation at the point (x, y) in the object at a fixed z-plane. Since the attenuation is directly proportional to the volumetric density of the object at that spatial position, a reconstructed image of $f(x, y)$ portrays a two dimensional non-negative density distribution.

The Radon transform can be thought of as an operator on the



The Graphics 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;
```



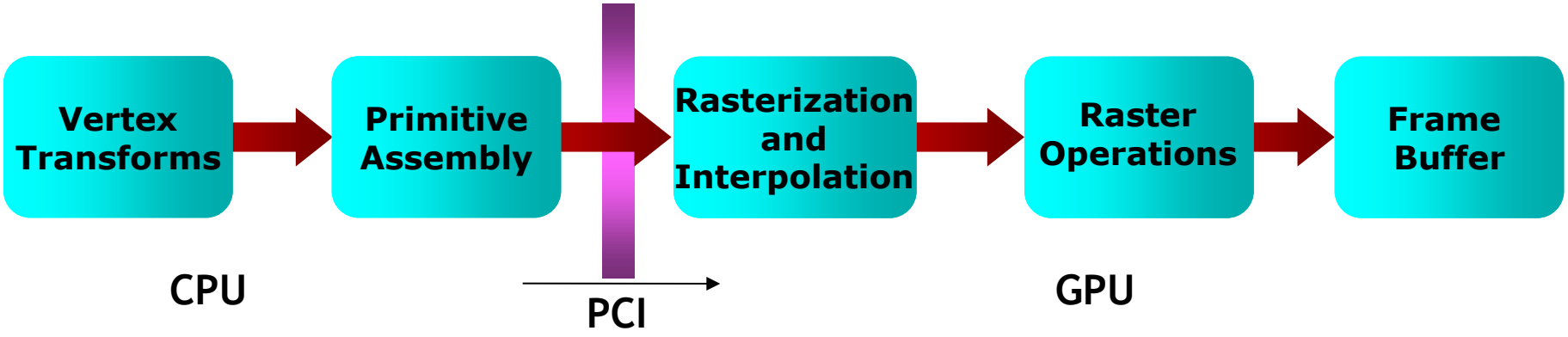
Wouldn't be cool to make these stages of the graphics pipeline programmable?

Thanks to Kurt Akeley

Generation I: 3dfx Voodoo (1996)

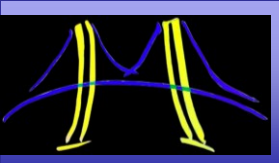
Diamond Multimedia
Monster3D

- Did not do vertex transformations: these were done in the CPU
- Did do texture mapping, z-buffering.
- 0.5 micron technology
- 1M transistors



Transistor counts and technology node information from: www.maximumpc.com/article/features/graphics_extravaganza_ultimate_gpu_retrospective

Slide adapted from Suresh Venkatasubramanian and Joe Kider



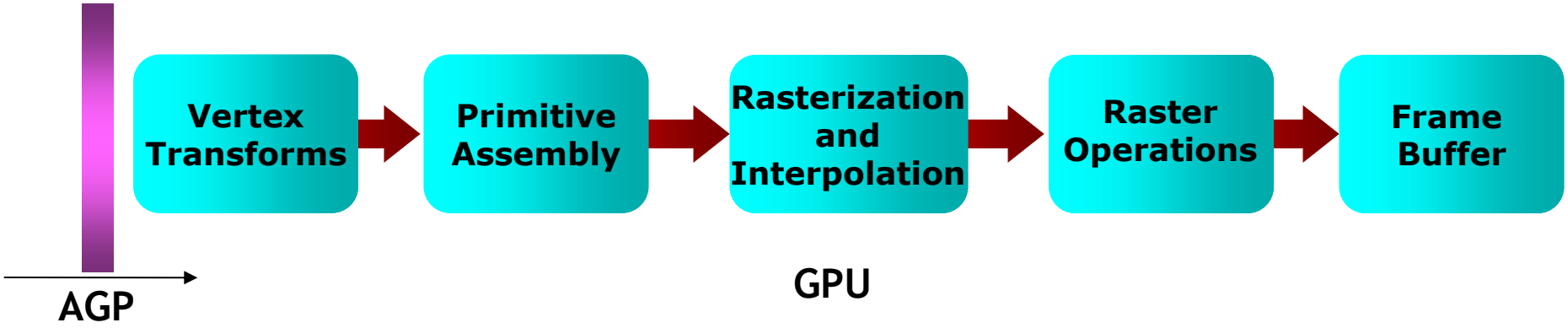
Generation I game: Quake2 (1997)

Generation II: GeForce 256/Radeon 7500 (1998)

GeForce 256

Image from “7 years of Graphics”

- Main innovation: shifting the transformation and lighting calculations to the GPU
- DirectX 7
- Allowed multi-texturing: giving bump maps, light maps, and others..
- Faster AGP bus instead of PCI
- 0.22 micron technology (GeForce 256)
- 23M transistors



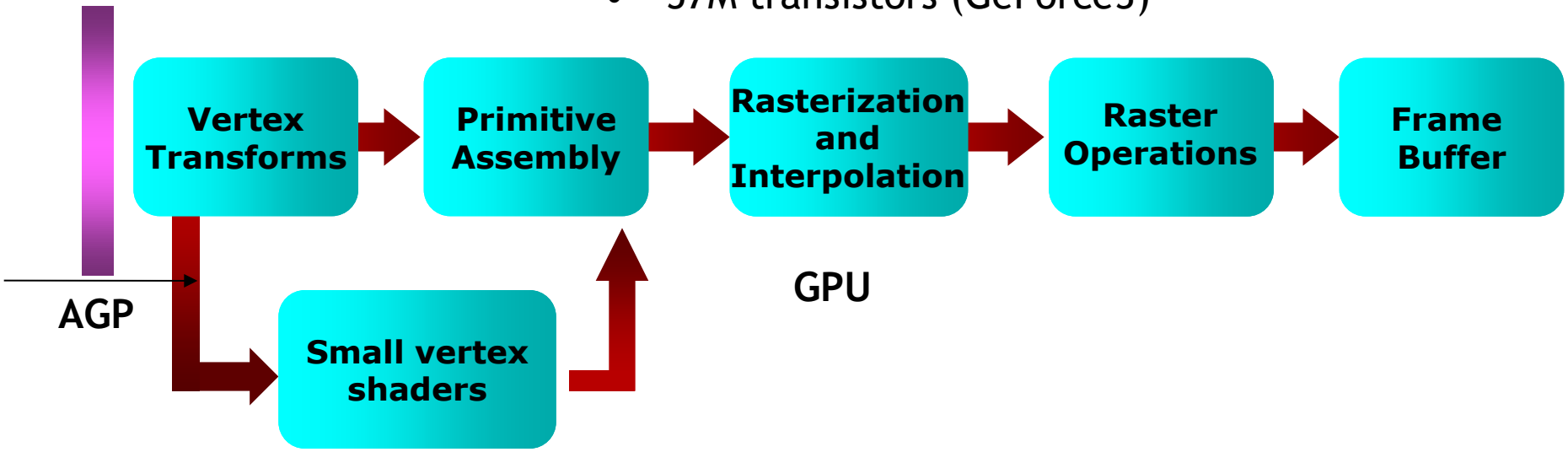
Generation II game: Quake3 (1999)

Quake3 now runs on mobile devices

Generation III: GeForce3/Radeon 8500(2001)

Image from "7 years of Graphics"

- For the first time, allowed limited amount of programmability in the vertex pipeline
- DirectX 8
- Also allowed volume texturing and multi-sampling (for antialiasing)
- 0.15 micron technology (GeForce3)
- 57M transistors (GeForce3)

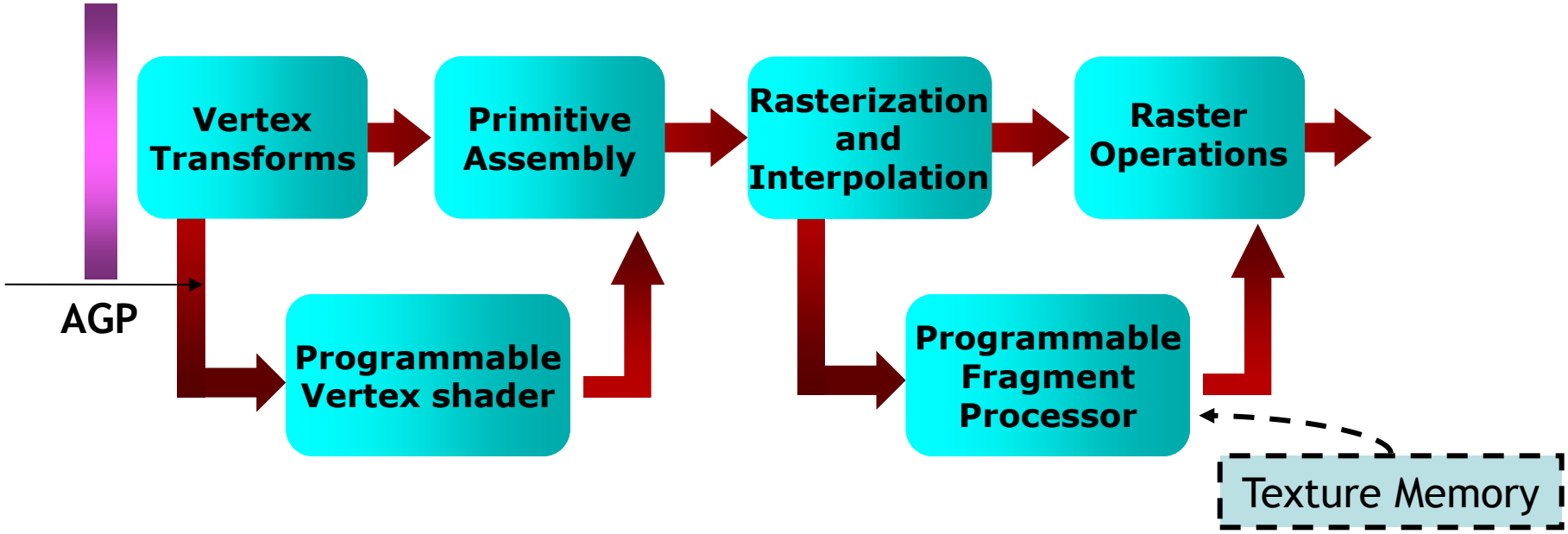


Generation III game: UT2004 (2003)

Generation IV: Radeon 9700/GeForce FX (2002)

Image from "7 years of Graphics"

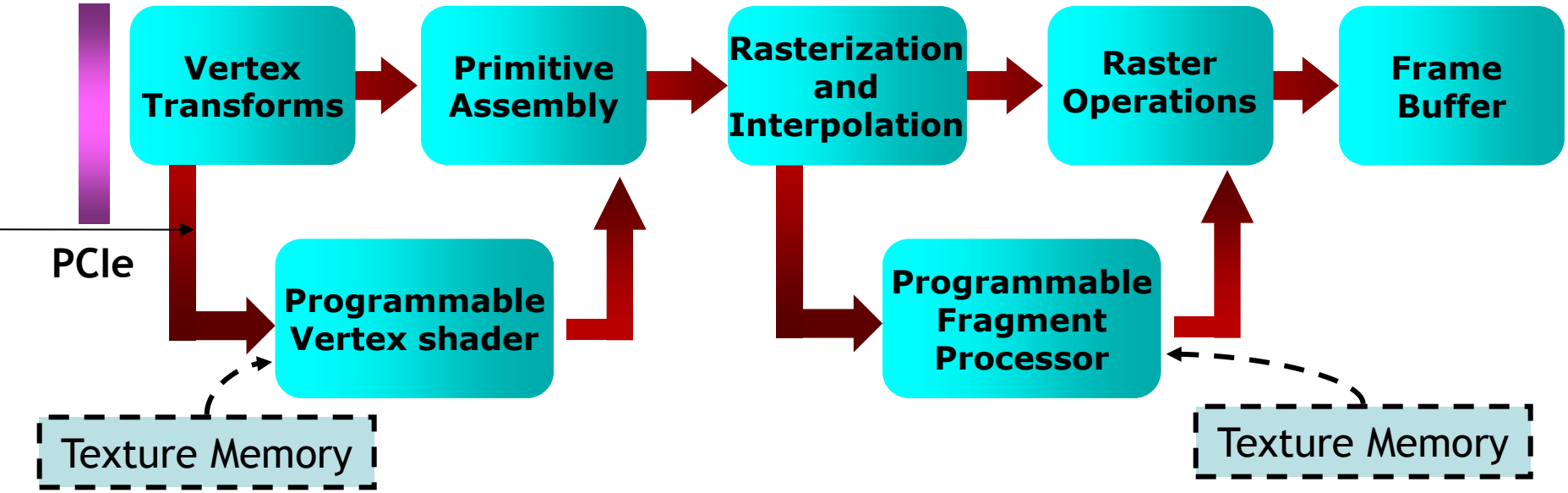
- This generation is the first generation of "fully-programmable" graphics cards
- DirectX 9 (shader model 2.0)
- Different versions have different resource limits on fragment/vertex programs
- 0.13 micron technology node
- 80M transistors



Generation III game: Half-Life 2

Generation IV: GeForce6/X800 (2004)

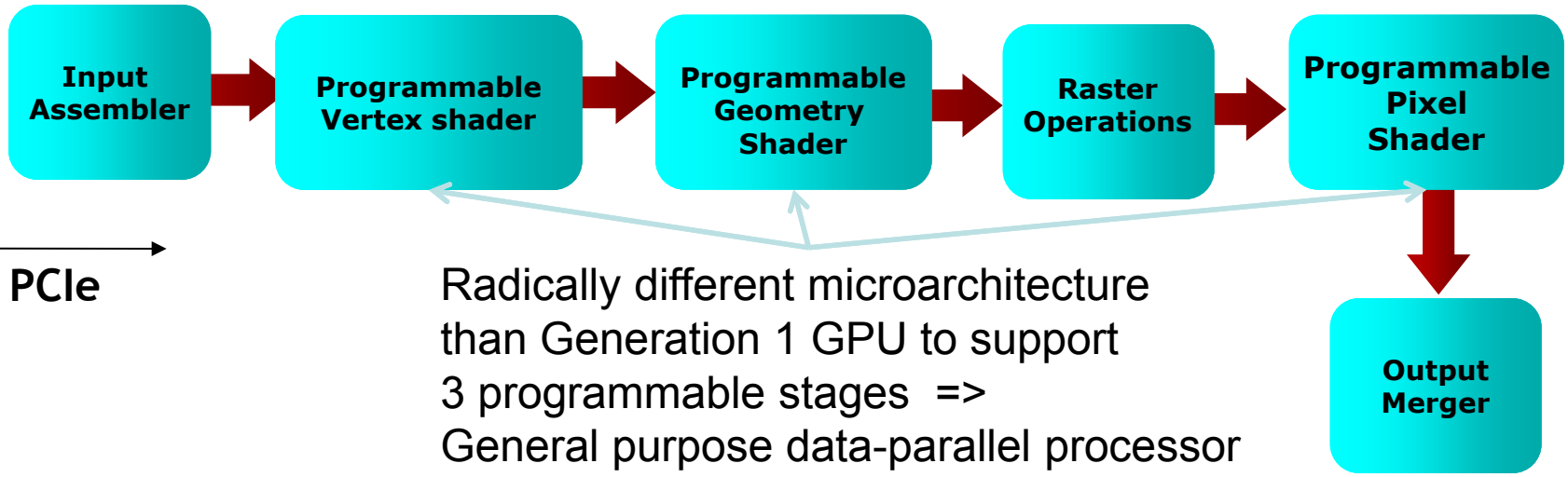
- Simultaneous rendering to multiple buffers
- DirectX 9 (shader model 3.0)
- True conditionals and loops
- PCIe bus
- Vertex texture fetch
- 0.11 micron technology
- 146M transistors



Generation IV game: FarCry (2004)

Generation V: GeForce8800/HD2900 (2006)

- Ground-up GPU redesign
- Support for Direct3D 10
- Geometry Shaders
- Stream out / transform-feedback
- Unified shader processors
- 0.09 micron technology
- 681M transistors!
- ***Support for General GPU programming***



Radically different microarchitecture than Generation 1 GPU to support 3 programmable stages => General purpose data-parallel processor

Slide adapted from Suresh Venkatasubramanian and Joe Kider

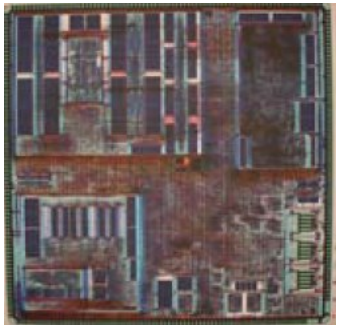
Generation V game: Crysis (2007)

GPGPU arrives: 2006

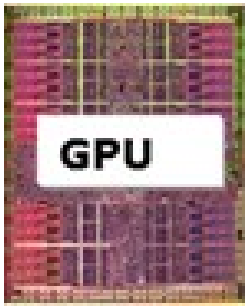
- ***Support for General GPU programming***
- ***But how will programmers write code for this GPU?***

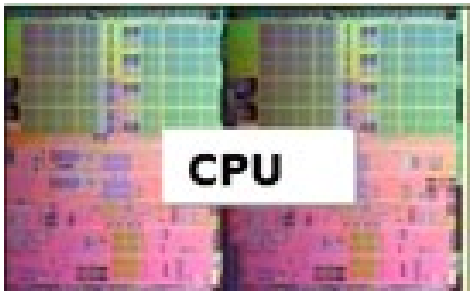
- Fortunately for NVIDIA, the academic community had been working on GPGPU programming for almost a decade.
- Ian Buck at Stanford was wrapping up his dissertation “Stream computing on Graphics Hardware” and the language “Brook”.
- He moved over to NVIDIA and led the effort to create CUDA.
- CUDA was extremely influential … Late in 2008 Apple, AMD, Intel, NVIDIA, Imagination Technologies and several other companies released a vendor-neutral, portable standard for stream computing called OpenCL.

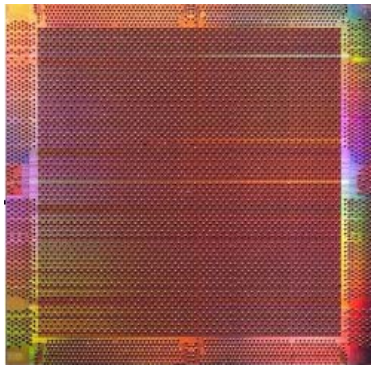
The heterogeneous platform: a Host (CPU) + a huge range of devices



DSP



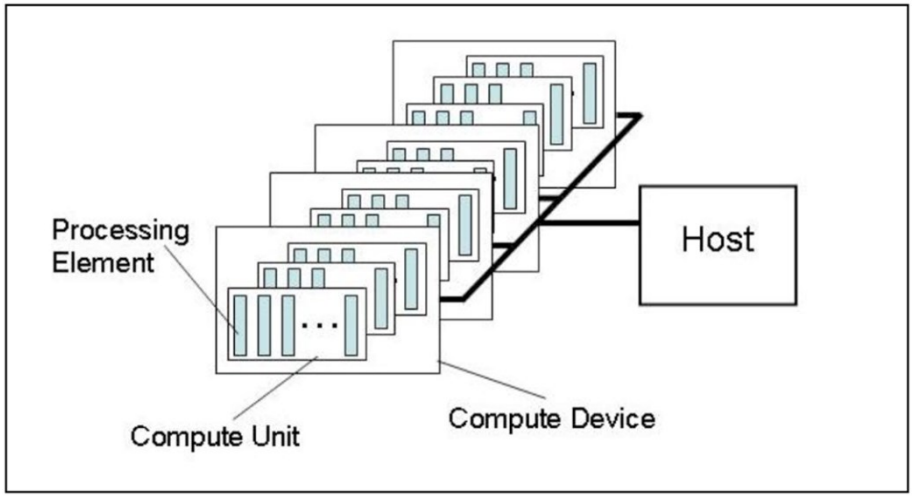




FPGA

Integrated CPU+ FPGA

(Intel® Atom™ Processor E6x5C Series)

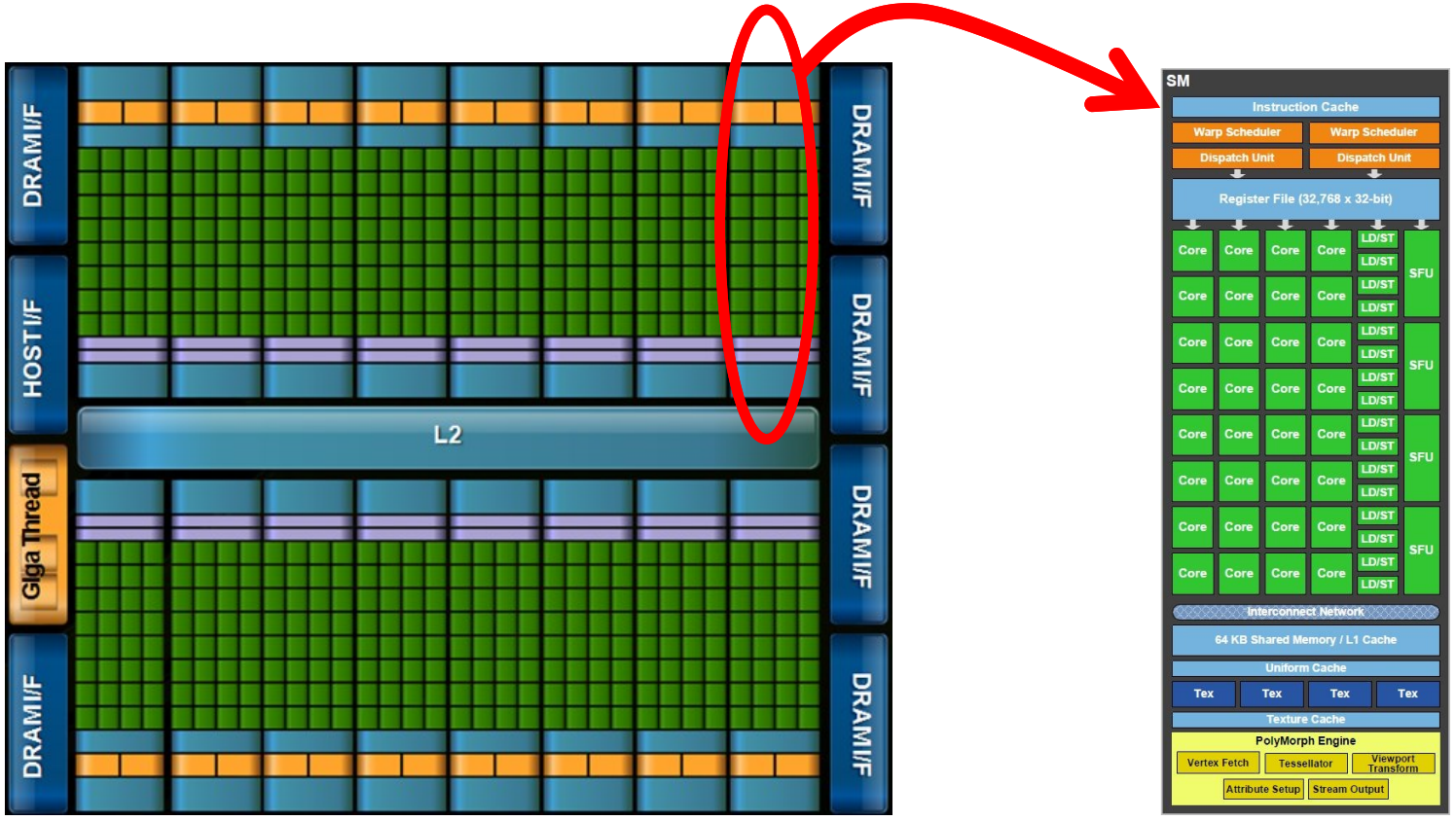


… and who knows what the future will bring?

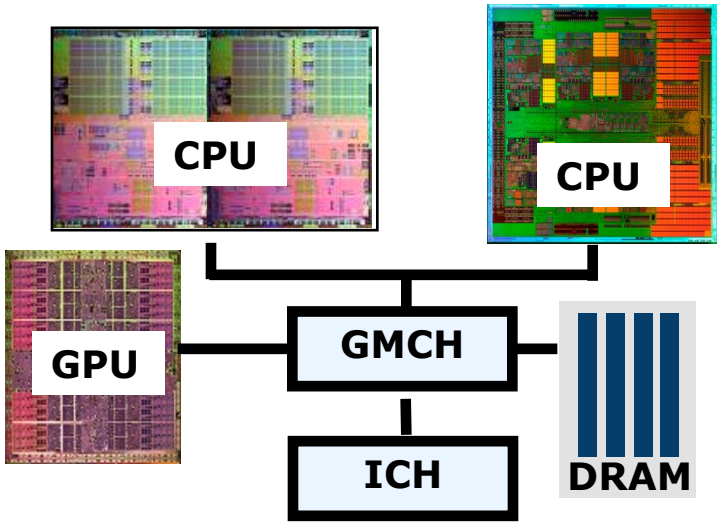
Many-core CPU (MIC & Xeon Phi™)

Nvidia GPU Architecture

- Nvidia GPUs are a collection of “Streaming Multiprocessors”
  - Each SM is analogous to a core of a Multi-Core CPU
- Each SM is a collection of SIMD execution pipelines that share control logic, register file, and L1 Cache

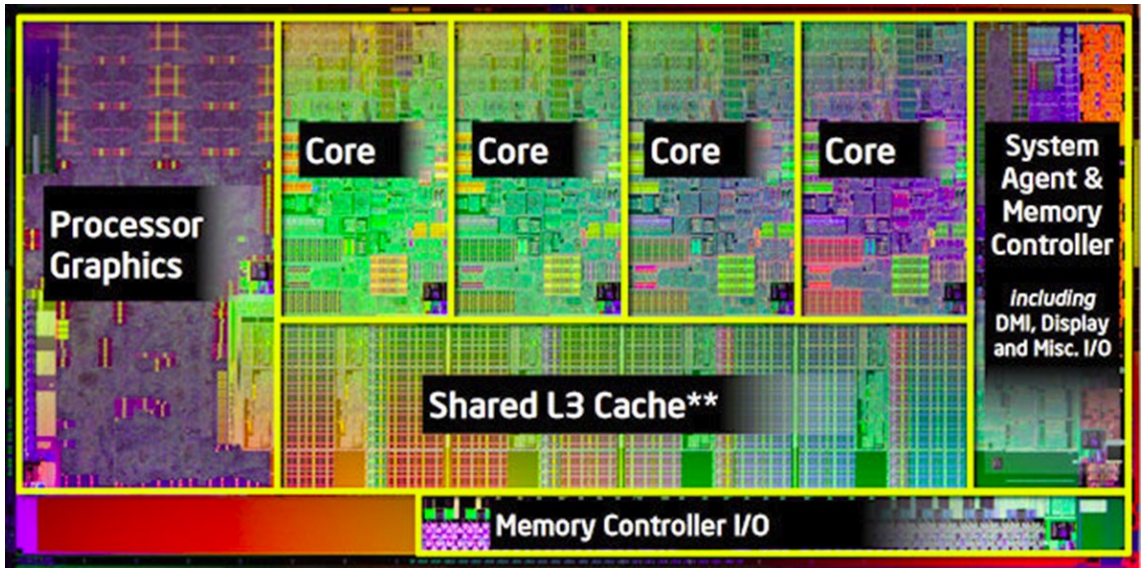


Our HW future is clear:



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

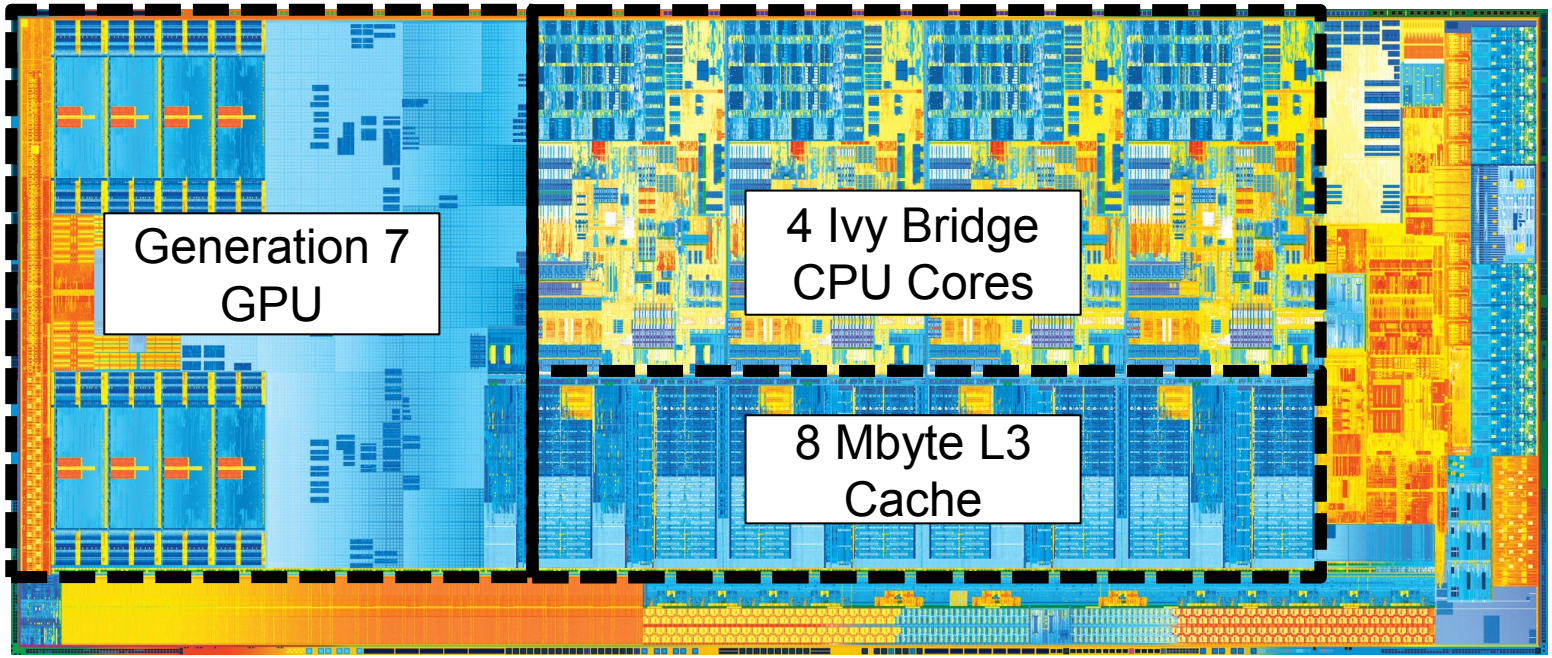
- Current designs put this functionality onto a single chip … mitigates the PCIe bottleneck in GPGPU computing!



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

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

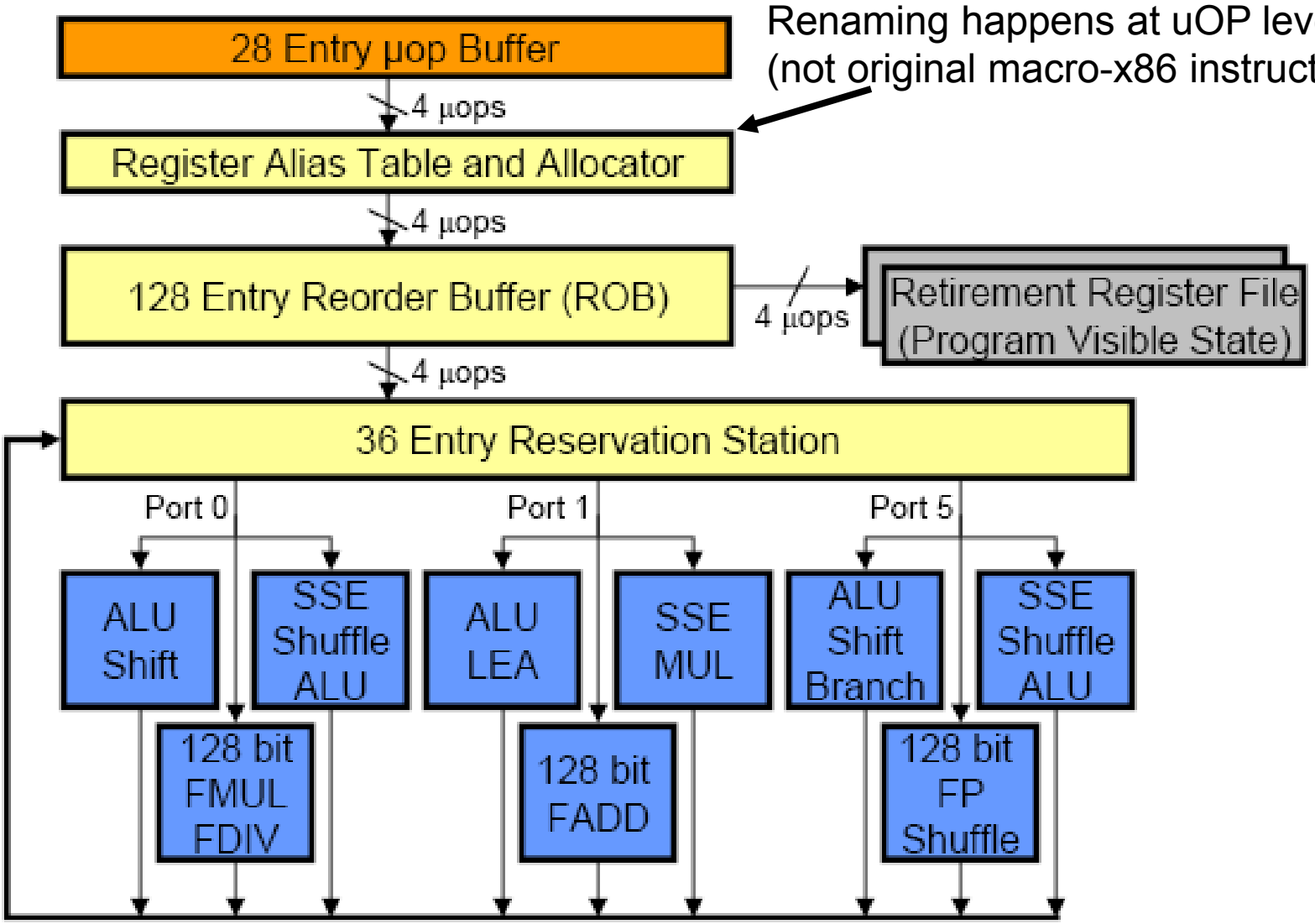
Intel's Ivy Bridge GPU



- 4 CPU cores + GPU
  - All integrated on the same die
  - GPU and aggregate CPUs have about the same peak performance
    - 256 single-precision Gflops/sec
- GPU is fully programmable with OpenCL
  - DirectX 11 too

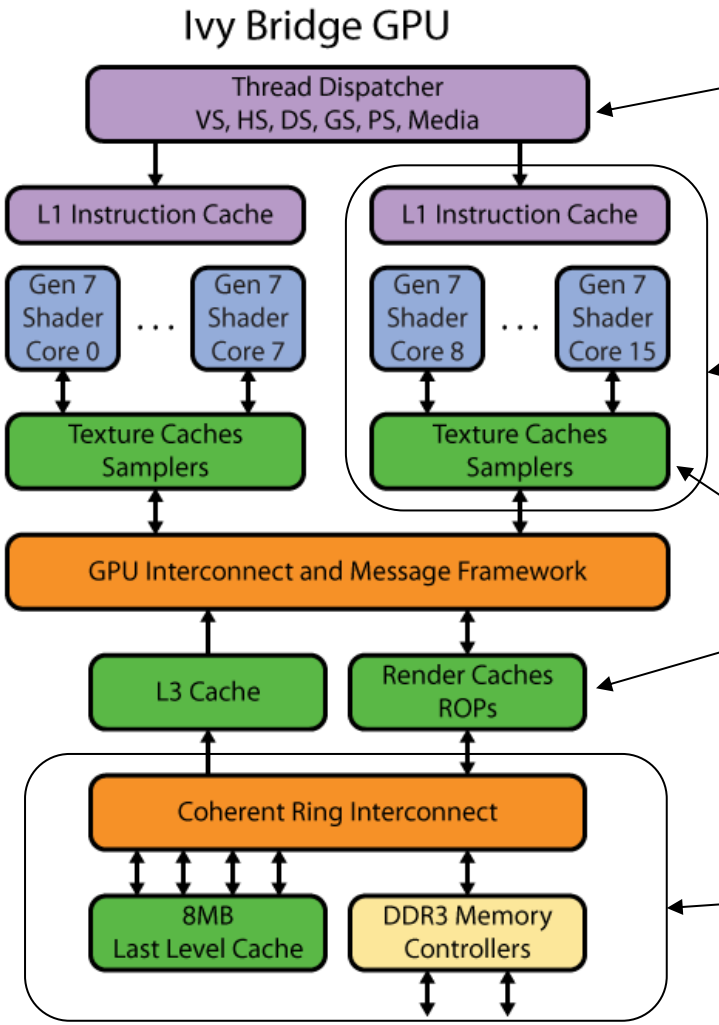
Thanks to David Kanter of Real World Tech and Intel

Out-of-Order Execution Engine



Thanks to David Kanter of Real World Tech, Intel, and Krste Asanovic of UCB

Ivy Bridge System Architecture



Hardware load balancing across GPU Cores

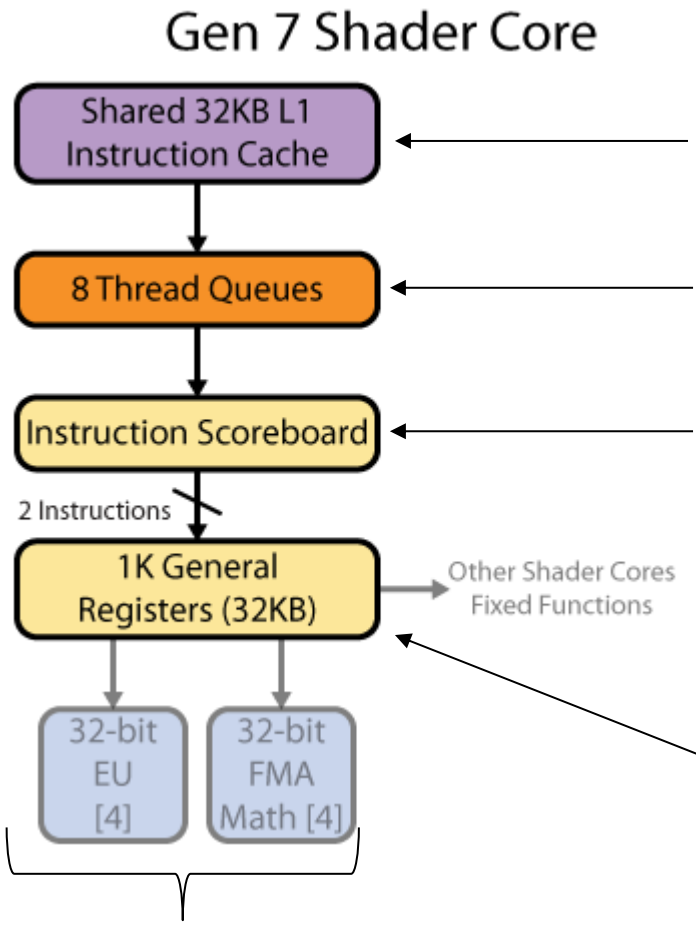
GPU Core
Equivalent structure on a NVIDIA GPU is called a "streaming multiprocessor"

GPUs include special memories for manipulating textures (images)

Shared with CPU cores

Thanks to David Kanter of Real World Tech and Intel

Ivy Bridge GPU Core Microarchitecture



Cache shared with 8 other processing lanes

8 active threads per processing lane to tolerate memory latency

Pipeline hazards detected and respected using a register scoreboard

128 32-bit physical registers per thread (about the same as the Ivy Bridge CPU)

Thanks to David Kanter of Real World Tech and Intel

So how do we program GPUs?

CUDA or OpenACC: if you are willing to restrict yourself to a single vendor's product (NVIDIA)

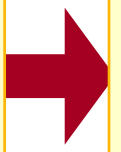
OpenCL or OpenMP 4.0: if you want portability across devices and vendors

The BIG idea behind OpenCL

- OpenCL execution model … execute a kernel at each point in a problem domain.
  - E.g., process a 1024 x 1024 image with one kernel invocation per pixel or 1024 x 1024 = 1,048,576 kernel executions

Traditional loops

```
void
trad_vadd(int n,
          const float *a,
          const float *b,
          float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
}
```



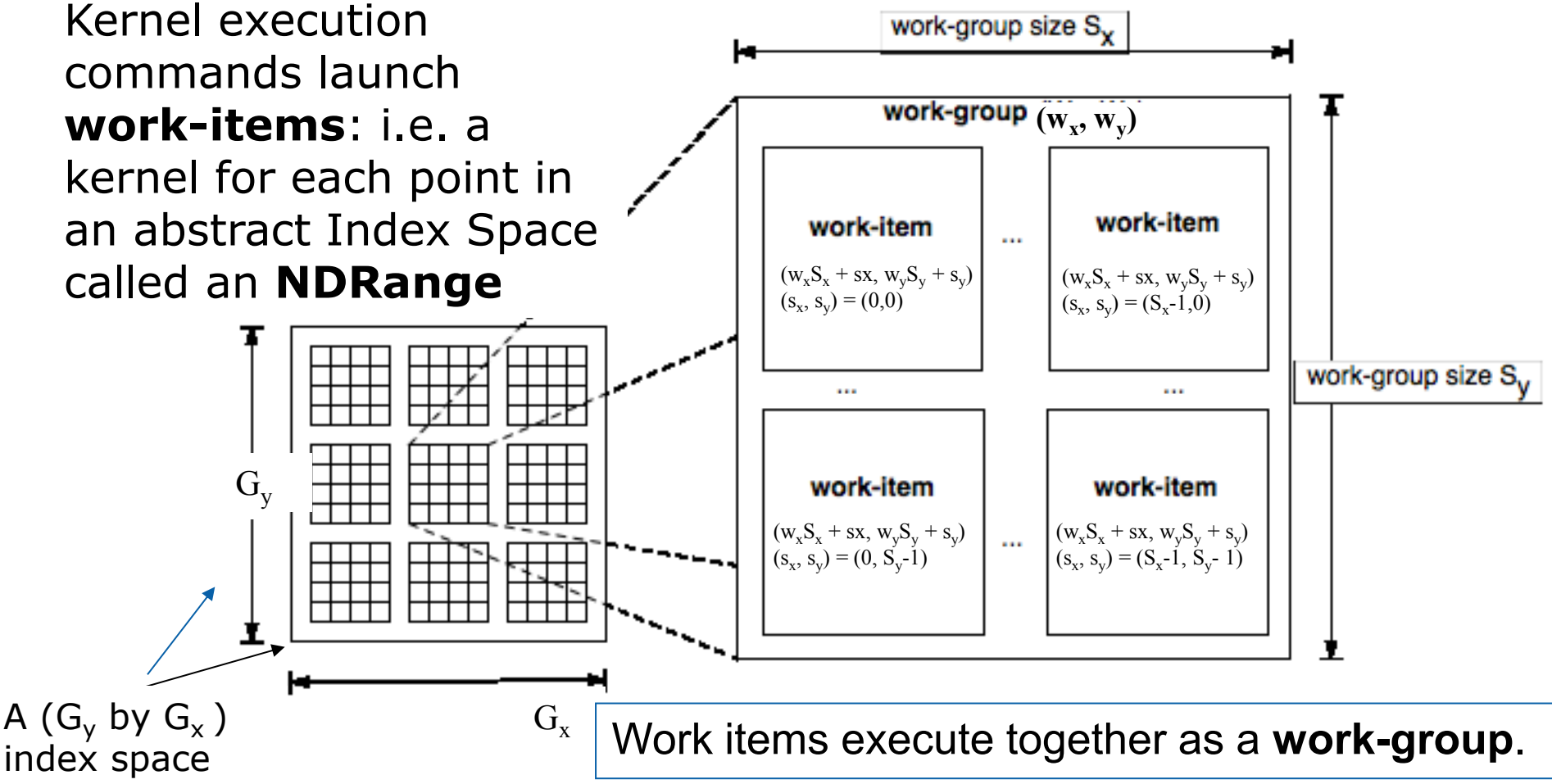
Kernel Parallelism OpenCL

```
kernel void
vec_add(global const float *a,
        global const float *b,
        global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] + b[id];

} // execute over "n" work-items
```

Execution Model

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

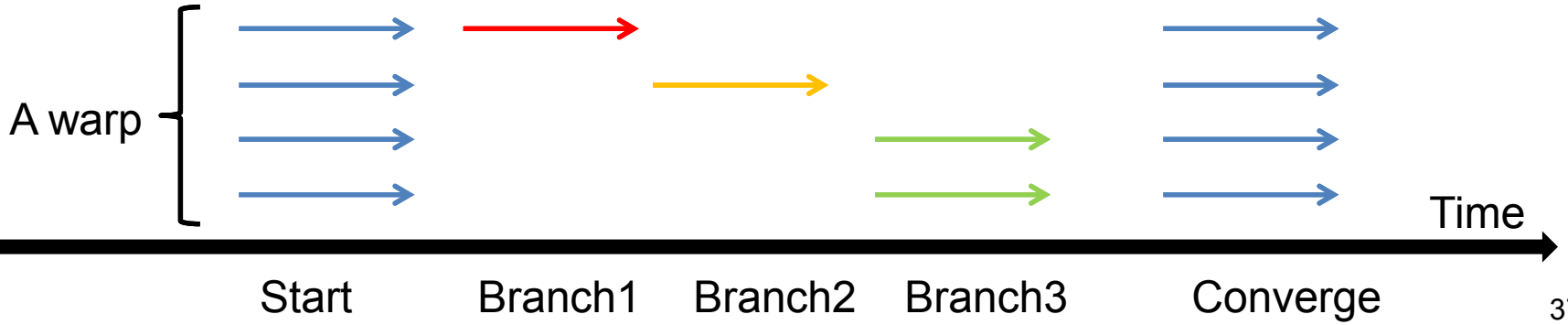
Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**



Work items execute together as a **work-group**.

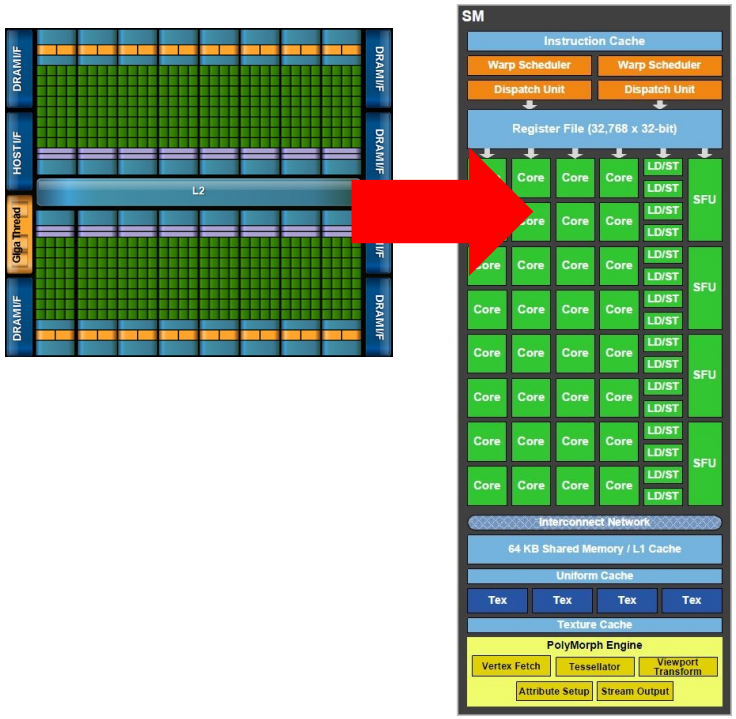
Single Instruction Multiple Data

- Individual threads of a warp start together at the same program address
- Each thread has its own instruction address counter and register state
  - Each thread is free to branch and execute independently
  - Provide the MIMD abstraction
- Branch behavior
  - Each branch will be executed serially
  - Threads not following the current branch will be disabled



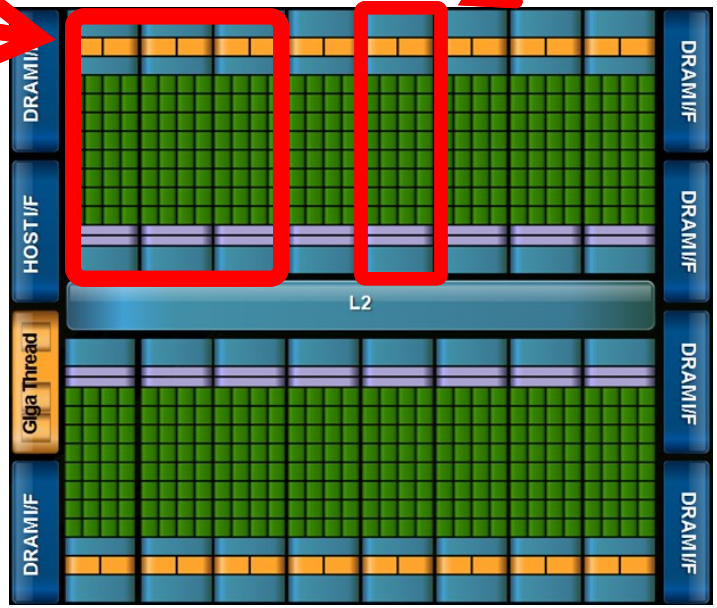
Mapping OpenCL to Nvidia GPUs

- 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 is mapped to Nvidia GPUs



Mapping OpenCL to Nvidia GPUs

- Work item ⇔ SIMD Lane ⇔CUDA Thread
- Warp ⇔ SIMD execution granularity
- Work Group ⇔ Streaming Multiprocessor (SM)
- Index Space ⇔ Multiple SMs

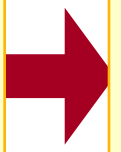


The BIG idea behind OpenCL

- OpenCL execution model … execute a kernel at each point in a problem domain.
  - E.g., process a 1024 x 1024 image with one kernel invocation per pixel or 1024 x 1024 = 1,048,576 kernel executions

Traditional loops

```
void
trad_vadd(int n,
          const float *a,
          const float *b,
          float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
}
```



Kernel Parallelism OpenCL

```
kernel void
vec_add(global const float *a,
        global const float *b,
        global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] + b[id];

} // execute over "n" work-items
```

The BIG idea behind CUDA

- CUDA execution model … execute a kernel at each point in a problem domain.
  - E.g., process a 1024 x 1024 image with one kernel invocation per pixel or 1024 x 1024 = 1,048,576 kernel executions

Traditional loops

```
void
trad_vadd(int n,
          const float *a,
          const float *b,
          float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
}
```

Kernel Parallelism CUDA

```
void __global__
vec_add(float *a,
        float *b,
        float *c)
{
  int id = blockIdx.x * blockDim.x
           + threadIdx.x;

 c[id] = a[id] + b[id];

} // execute over "n" work-items
```

OpenCL vs. CUDA Terminology

- Host defines a **command queue** and associates it with a context (devices, kernels, memory, etc).
- Host enqueues commands to the command queue

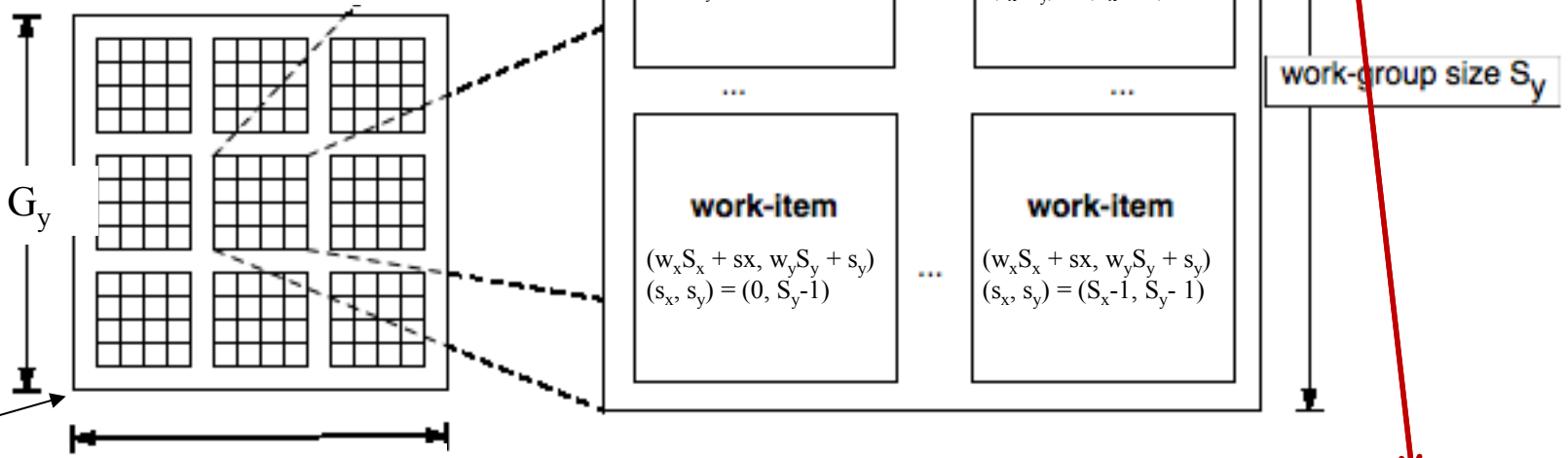
CUDA Stream

Thread Block

Threads

Kernel execution commands launch **work-items**: i.e. a kernel for each point in an abstract Index Space called an **NDRange**

Grid



A (G_y by G_x) index space

Work items execute together as a **work-group**.

OpenCL or CUDA … they require massive changes to existing code. There has got to be a better way.

- OpenMP allows most Application programmers to ignore pthreads (or C++'11 threads). Dramatically simplifies their life.
- Can we use the same directive-oriented scheme we used for OpenMP? Yes.
  - OpenMP 4.0 (spec. released Nov'2013) includes constructs for directive driven GPU programming
  - OpenACC (a proprietary spec from Cray, PGI and Nvidia released in June'2012) took an early version of the OpenMP 4.0 work and released it as their own (with proper attribution… they aren't bad people)

The big idea behind OpenACC

- Let's add two vectors together …. C = A + B

```
void vadd(int n,
          const float *a,
          const float *b,
          float *restrict c)
{
  int i;
 #pragma acc parallel loop
  for (i=0; i<n; i++)
    c[i] = a[i] + b[i];
}
int main(){
float *a, *b, *c;  int n = 10000;
// allocate and fill a and b

    vadd(n, a, b, c);

}
```

Assure the compiler that c is not aliased with other pointers

Turn the loop into a kernel, move data to a device, and launch the kernel.

Host waits here until the kernel is done. Then the output array c is copied back to the host.

A more complicated example:
Jacobi iteration: OpenACC (GPU)

Turn the loop into a kernel, move data to a device, and launch the kernel.

Host waits here until the kernel is done.

```
while (err>tol && iter < iter_masx){
   err = 0.0;
   #pragma acc parallel loop reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] – A[j][i]));
       }
    }
    #pragma acc parallel loop
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Source: based on Mark Harris of NVIDIA®, “Getting Started with OpenACC”, GPU technology Conf., 2012

A more complicated example:
Jacobi iteration: OpenACC (GPU)

A, and Anew copied between the host and the GPU on each iteration

```
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma acc parallel loop reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
      }
   }
   #pragma acc parallel loop
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
      }
   }
   iter ++;
}
```

Performance was poor due to excess memory movement overhead

Source: based on Mark Harris of NVIDIA®, “Getting Started with OpenACC”, GPU technology Conf., 2012

The OpenACC data environment

- Data is moved as needed by the compiler on entry and exit from a parallel or kernel region.
- Data copy overhead can kill performance.
- Solution?
  - A data region to explicitly control data movement.

    #pragma acc data
  - Data movement is explicit …. Compiler no longer moves data for you.
  - Key clauses
    - **Copy, copyin, copyout**: move indicated list of variables between host and device on entry/exit form data region
    - **Create**: create the data on the accelerator.
    - **Private, firstprivate**: same meaning as with OpenMP …. Scalars are made private by default.

A more complicated example:
Jacobi iteration: OpenACC (GPU)

```
#pragma acc data copy(A), create(Anew)
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma acc parallel loop reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
       }
    }
    #pragma acc parallel loop
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Create a data region on the GPU. Copy A once onto the GPU, and create Anew on the device (no copy from host)

Copy A back out to host … but only once

Source: based on Mark Harris of NVIDIA®, "Getting Started with OpenACC", GPU technology Conf., 2012

OpenMP 4.0

Jacobi iteration: OpenMP accelerator directives

```
#pragma omp target data map(A, Anew)
while (err>tol && iter < iter_max){
   err = 0.0;
   #pragma target
   #pragma omp parallel for reduction(max:err)
   for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         Anew[j][i] = 0.25* (A[j][i+1] + A[j][i-1]+
                             A[j-1][i] + A[j+1][i]);
         err = max(err,abs(Anew[j][i] - A[j][i]));
       }
    }
    #pragma omp target
    #pragma omp parallel for
    for(int j=1; j< n-1; j++){
      for(int i=1; i<M-1; i++){
         A[j][i] = Anew[j]i];
       }
    }
    iter ++;
}
```

Create a data region on the GPU. Map A and Anew onto the target device

Uses existing OpenMP constructs such as parallel and for

Copy A back out to host … but only once

Conclusion

- The hardware trends are clear: Throughput optimized processors integrated with CPUs are here to stay.
- Software is evolving:
  - OpenCL and CUDA today.
  - Directive driven approaches (OpenACC and OpenMP 4.0) tommorow.
- Advice:
  - Stick to industry standards to force vendors to “play nice”