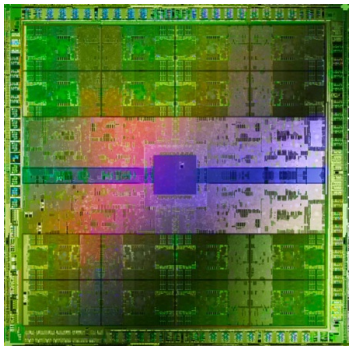
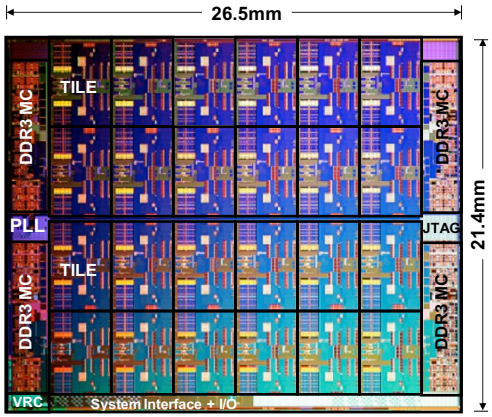
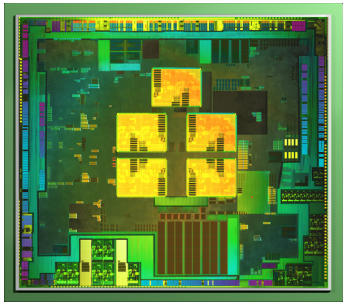


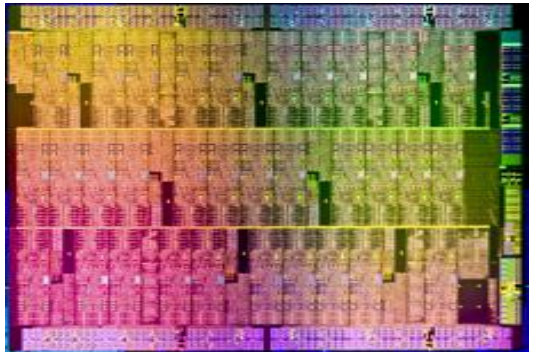
NVIDIA GTX 480 processor



Intel labs 48 core SCC processor



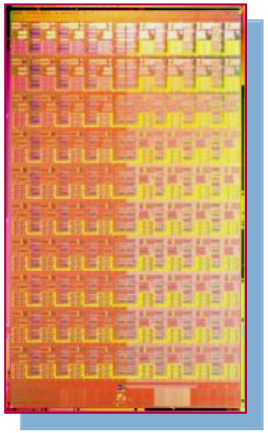
NVIDIA Tegra 3 (quad Arm Corex A9 cores + GPU)



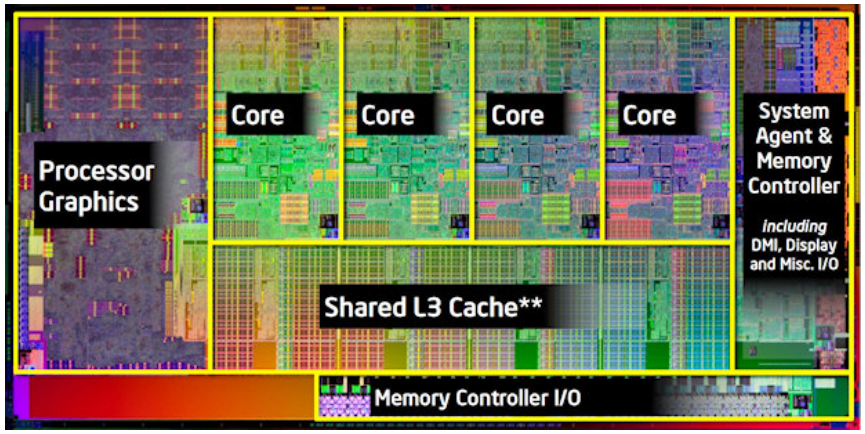
An Intel MIC processor

GPUs and the Heterogeneous programming problem

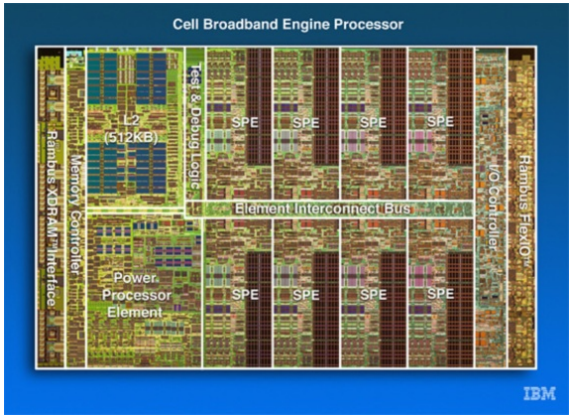
Tim Mattson (Intel Labs)



Intel Labs 80 core Research processor



Intel "Sandybridge" processor



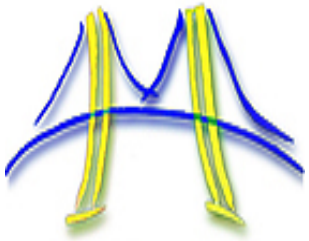
IBM Cell Broadband engine processor

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

Disclaimer

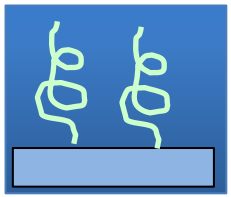
READ THIS … its very important

- 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.

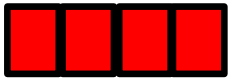


Slides marked with this symbol were produced-with Kurt Keutzer and his team for CS194 … A UC Berkeley course on Architecting parallel applications with Design Patterns.

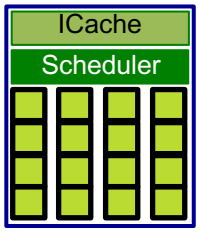
Hardware Diversity: Basic Building Blocks



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

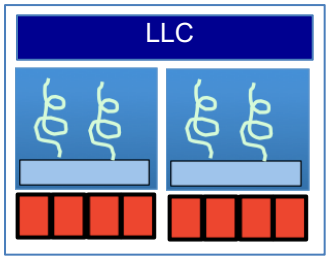


SIMD: Single Instruction Multiple Data.
Vector registers/instructions with 128 to 512 bits so a single stream of instructions drives multiple data elements.

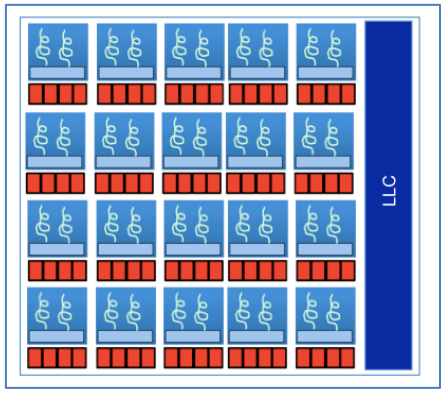


SIMT: Single Instruction Multiple Threads.
A single stream of instructions drives many threads. More threads than functional units. Over subscription to hide latencies. Optimized for throughput.

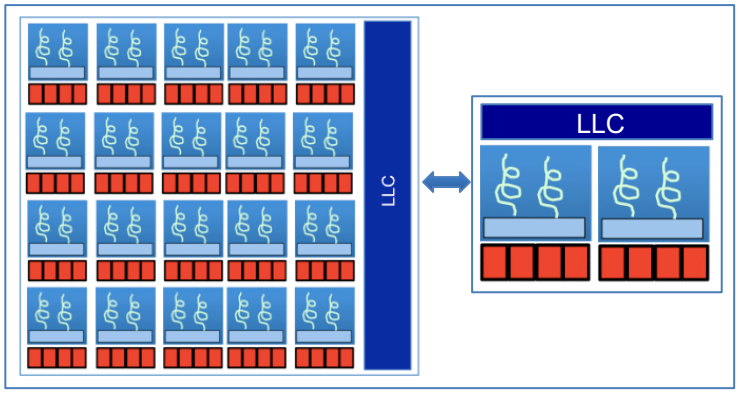
Hardware Diversity: Combining building blocks to construct nodes



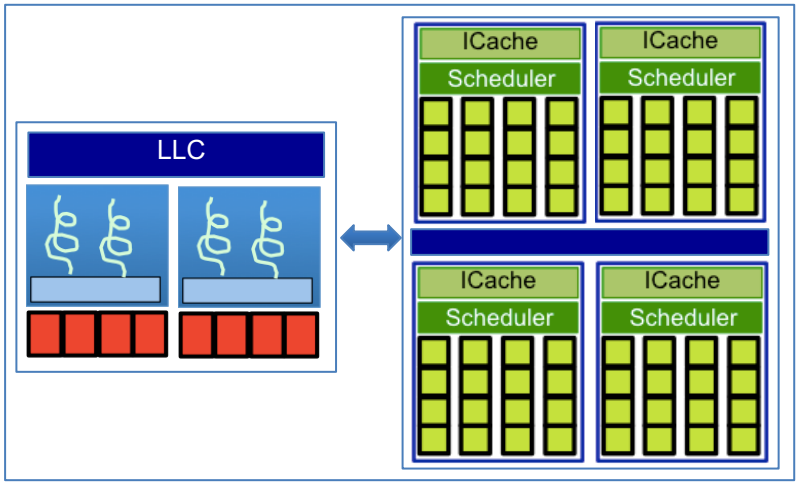
Multicore CPU



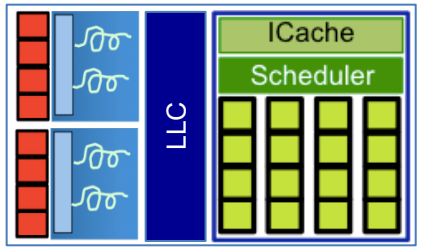
Manycore CPU



Heterogeneous:
CPU + manycore coprocessor



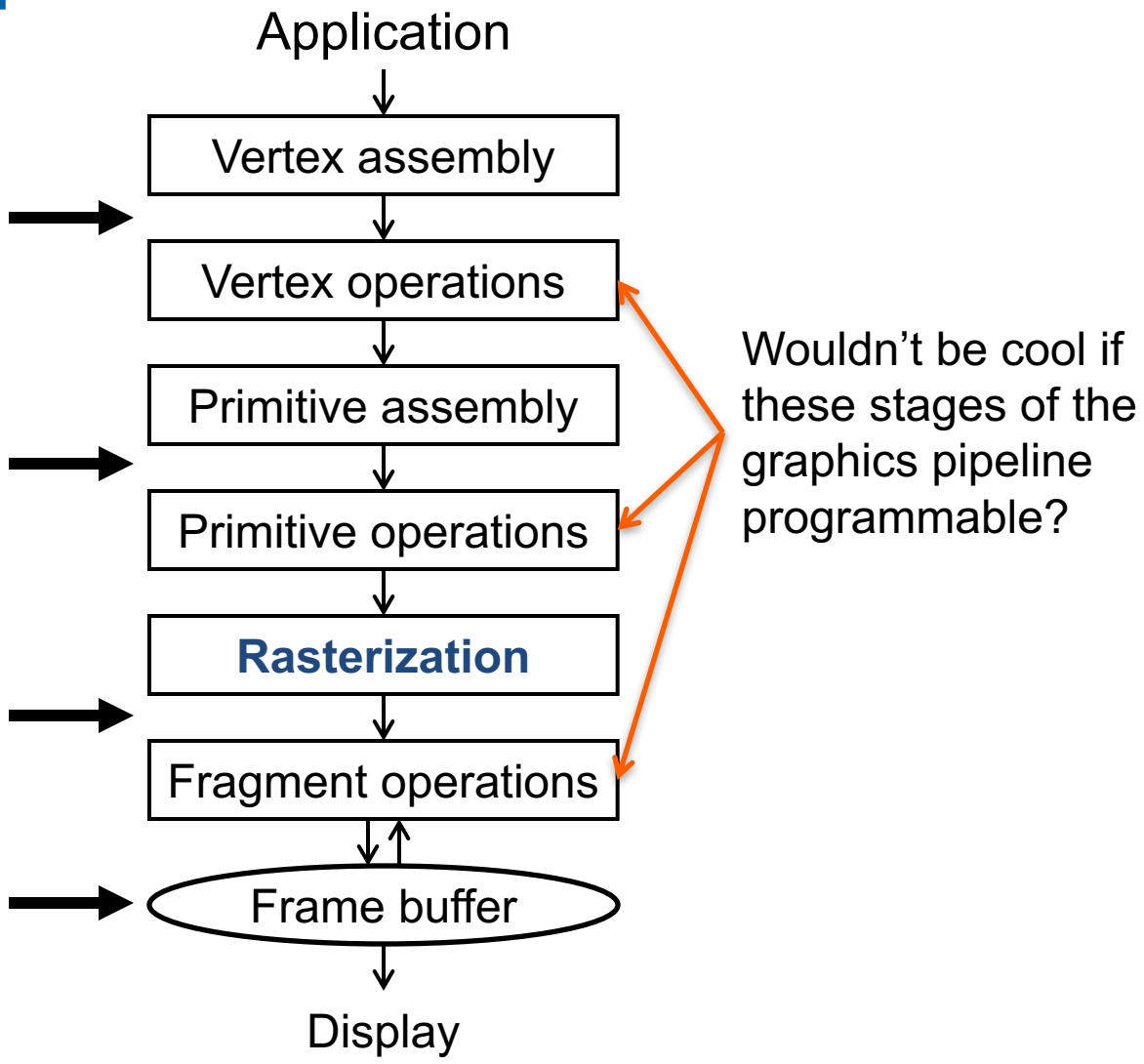
Heterogeneous: CPU+GPU



Heterogeneous:
Integrated CPU+GPU

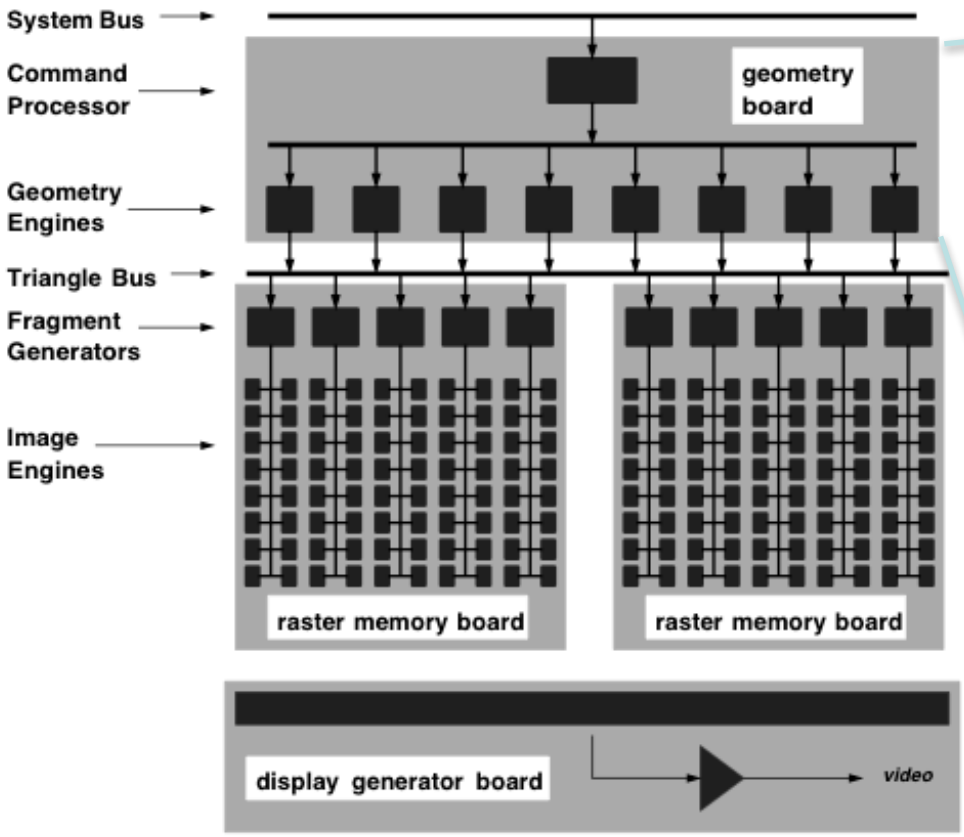
Let's take a deeper look at the GPU: The vertex pipeline

```
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;
```



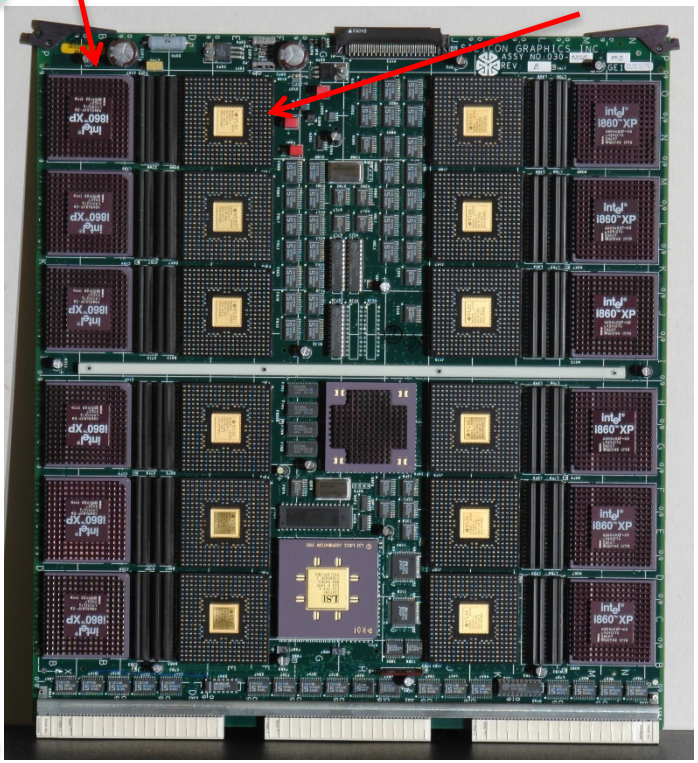
Thanks to Kurt Akeley

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.

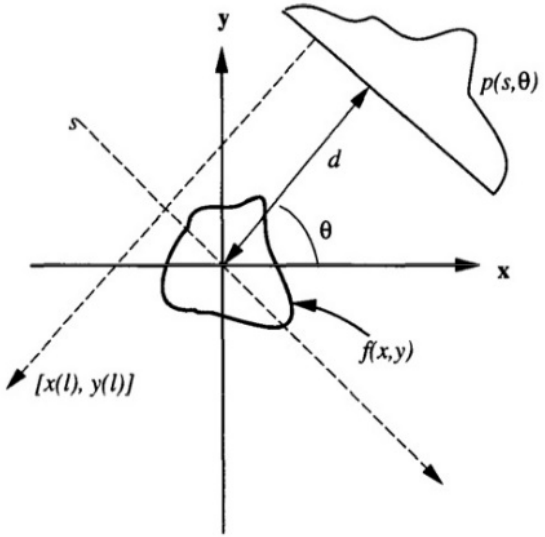


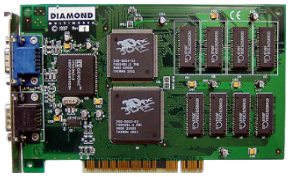
Figure 1: The Radon transform represents a generalized line integral projection of a 2-D (or 3-D) function $f(x, y, z)$ onto a line or plane.

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)

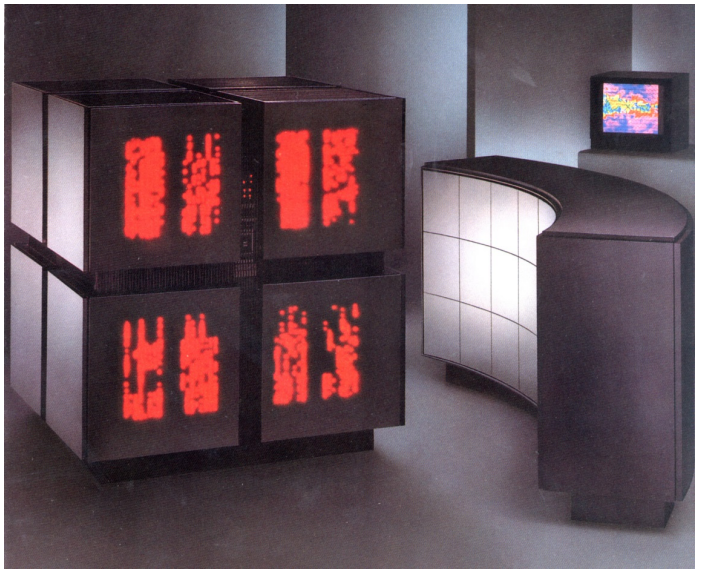
3rd Generation: GeForce3/Radeon 8500 (2001). The first GPU to allow a limited programmability in the vertex pipeline.

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

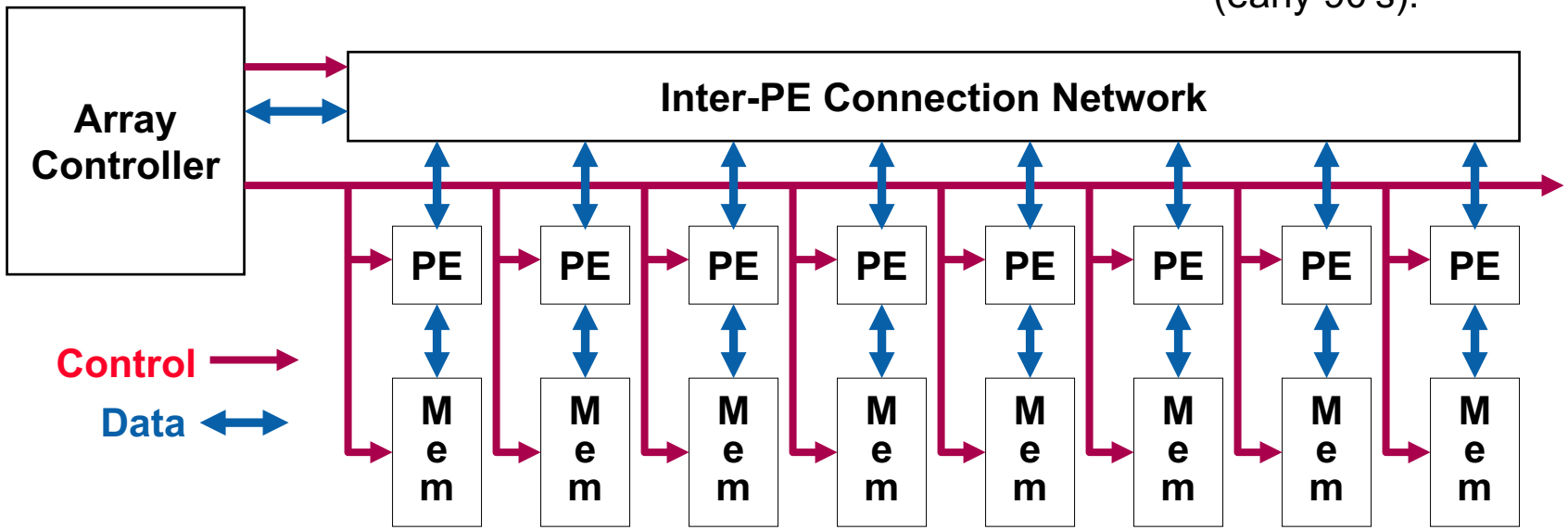
5th Generation: GeForce 8800/HD2900 (2006) and the birth of CUDA

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).

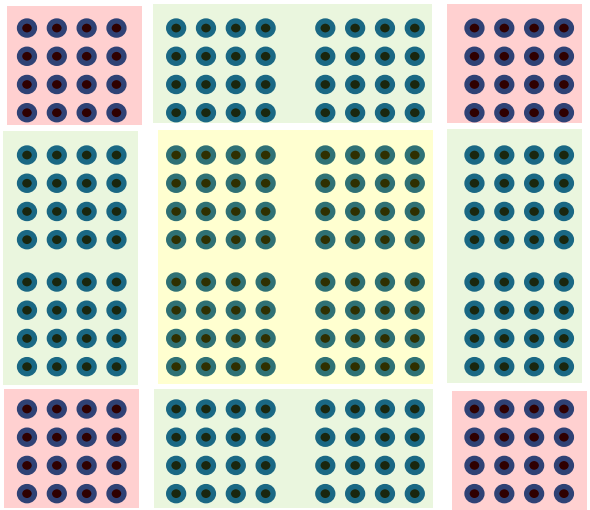


The SIMT model (single instruction Multiple thread)

1. Turn source code into a scalar instruction stream

```
extern void reduce(   __local  float*,   __global float*);

__kernel void pi(  const int niters, float  step_size,
       __local  float*  l_sums,  __global float*  p_sums)
{
  int n_wrk_items  = get_local_size(0);
  int loc_id       = get_local_id(0);
  int grp_id   = get_group_id(0);
  float x, accum = 0.0f;    int i,istart,iend;

  istart =   (grp_id * n_wrk_items   + loc_id) * niters;
  iend   = istart+niters;

  for(i= istart; i<iend; i++){
     x = (i+0.5f)*step_size;    accum += 4.0f/(1.0f+x*x); }

  l_sums[local_id] = accum;
  barrier(CLK_LOCAL_MEM_FENCE);
  reduce(l_sums, p_sums);
}
```

2. Map instruction streams onto an an N dim index space

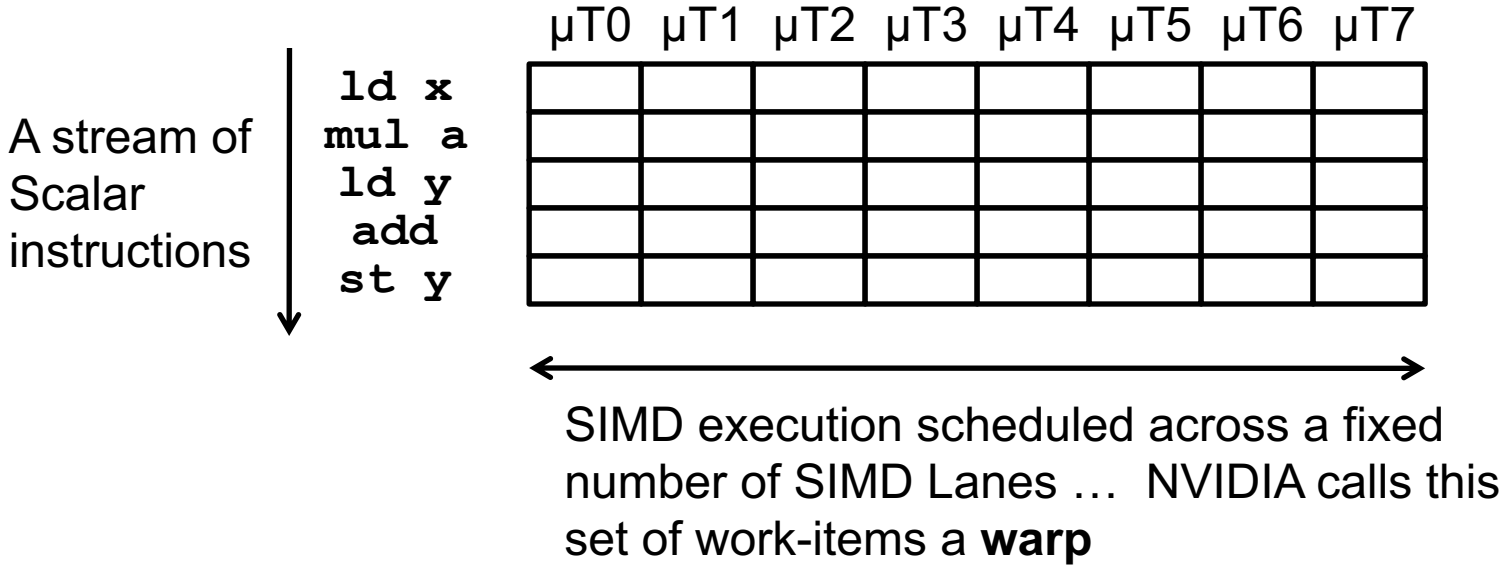


3. Map data structures onto the same index space

4. Run on hardware designed around the same SIMT execution model

SIMT execution strategy

- The underlying GPU hardware is still a vector processor … Individual scalar instruction streams are grouped together for SIMD execution on hardware



- But the programmer writes scalar code that runs in multiple vector lanes …. often much easier than traditional vectorization

Slide thanks to Krste Asanovic

Programming models for SIMT platforms

- Proprietary solutions based on CUDA and OpenACC.
- But there are Open Standard solutions (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!!!

*third party names are the property of their owners

Our Strategy

- We will start with OpenCL … the Open Standard
- We will explore the full range of issues required to program a GPU.
- We will then (time permitting) show how the same concepts map onto the other key GPGPU models
  - CUDA
  - OpenMP 4.5 and OpenACC
- We will close with high level observations on the field of GPGPU computing