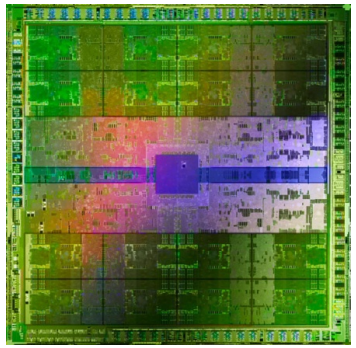
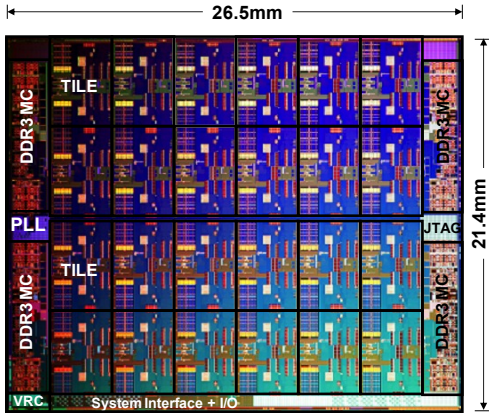
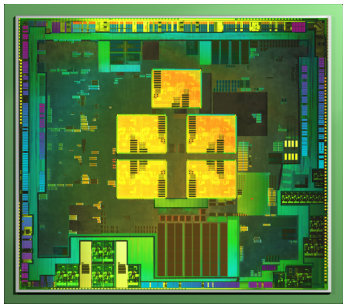


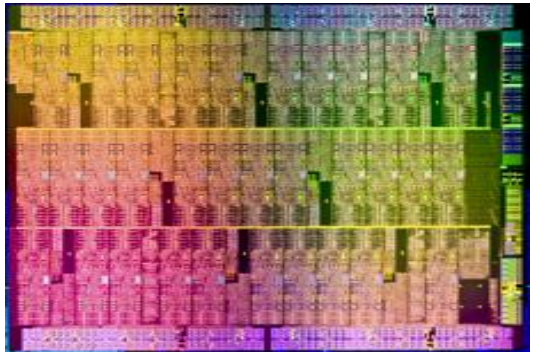
NVIDIA GTX 480 processor



Intel labs 48 core SCC processor



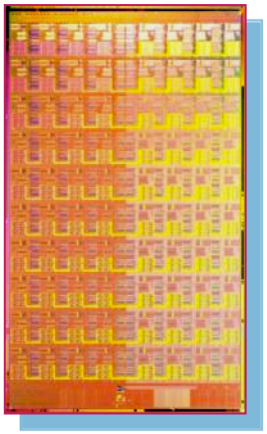
NVIDIA Tegra 3 (quad Arm Corex A9 cores + GPU)



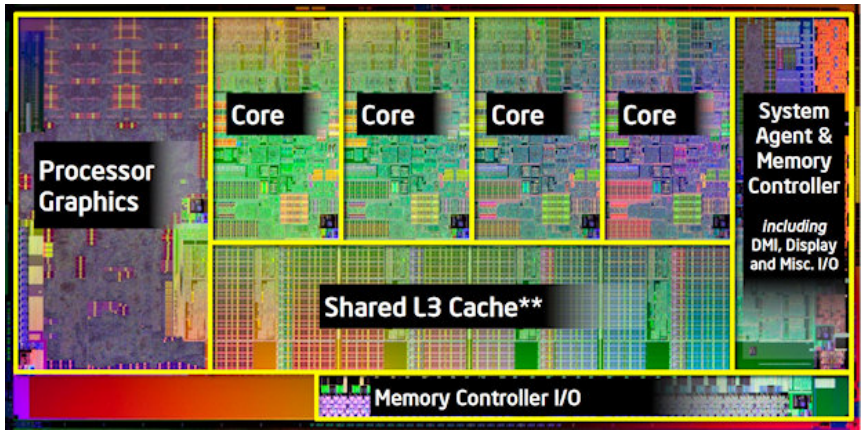
An Intel MIC processor

GPUs and the Heterogeneous programming problem

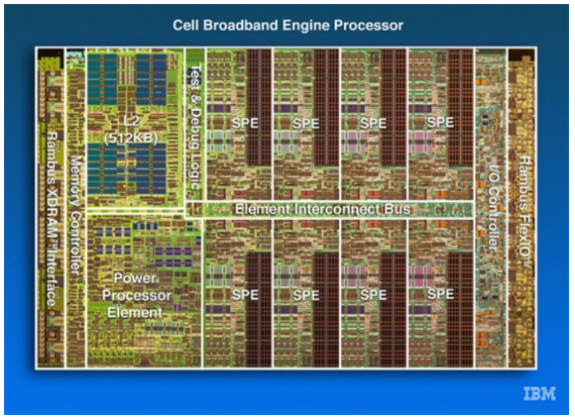
Tim Mattson (Intel Labs)



Intel Labs 80 core Research processor



Intel "Sandybridge" processor



IBM Cell Broadband engine processor

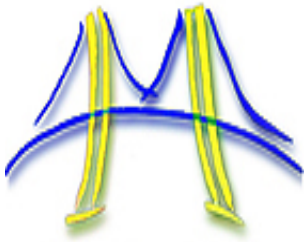
Other than the Intel lab's research processors. Die photos from UC Berkeley CS194 lecture notes

Third party names are the property of their owners

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.

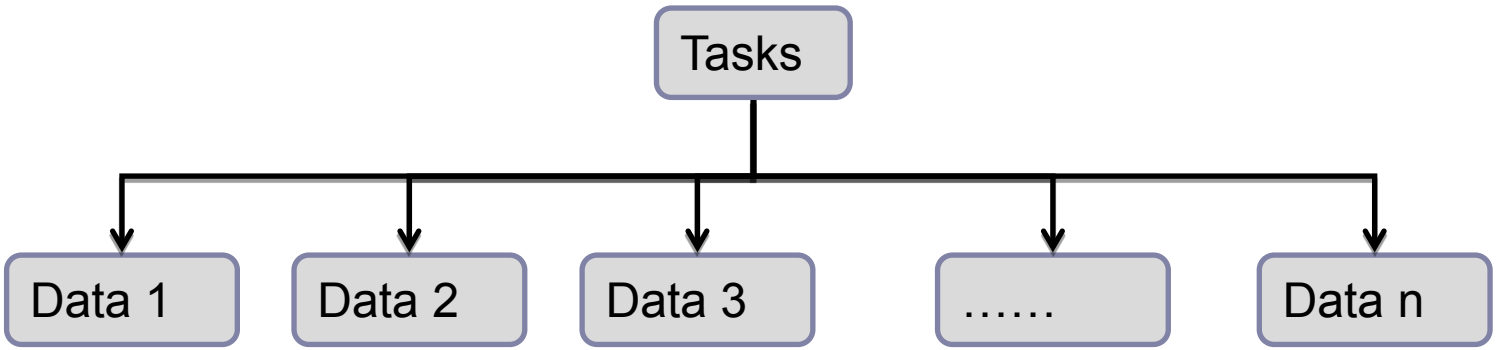


Slides marked with this symbol were produced-with Kurt Keutzer and his team for CS194 … A UC Berkeley course on Architecting parallel applications with Design Patterns.

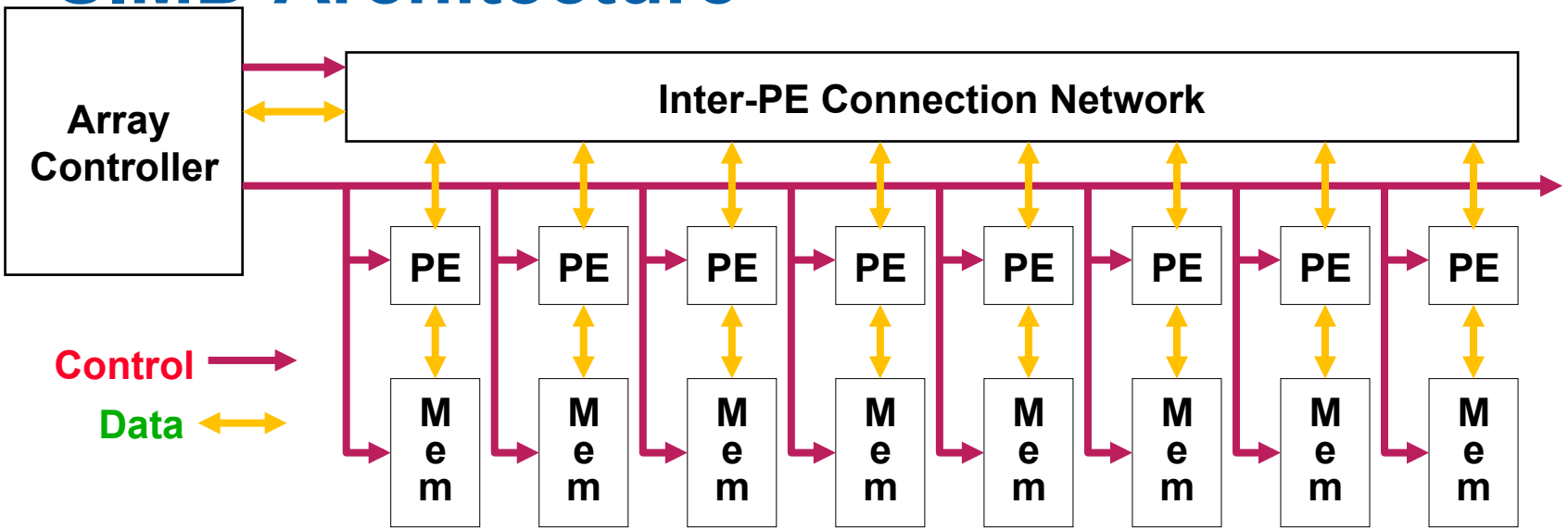
Alt intro to GPU hardware

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

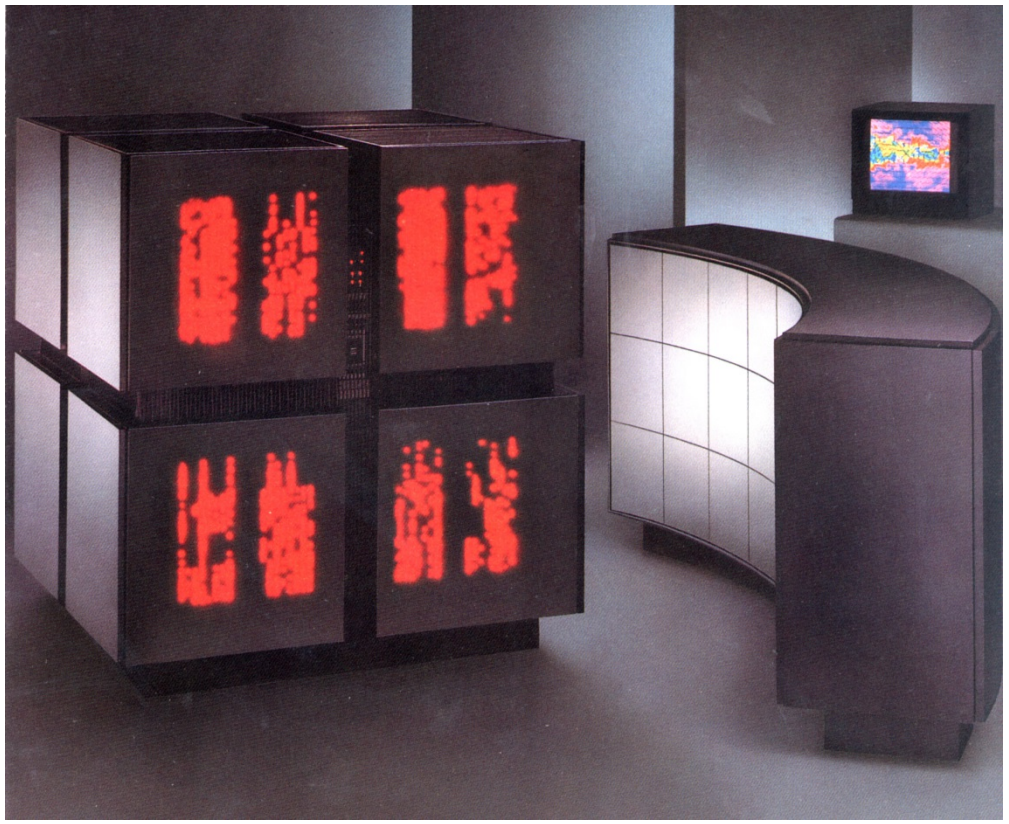


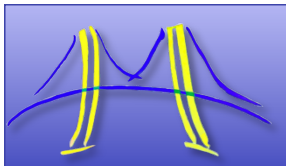
- Single Instruction Multiple Data (SIMD)
- Central controller broadcasts instructions to multiple processing elements (PEs)
  - Only requires one controller for whole array
  - Only requires storage for one copy of program
  - All computations fully synchronized

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

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

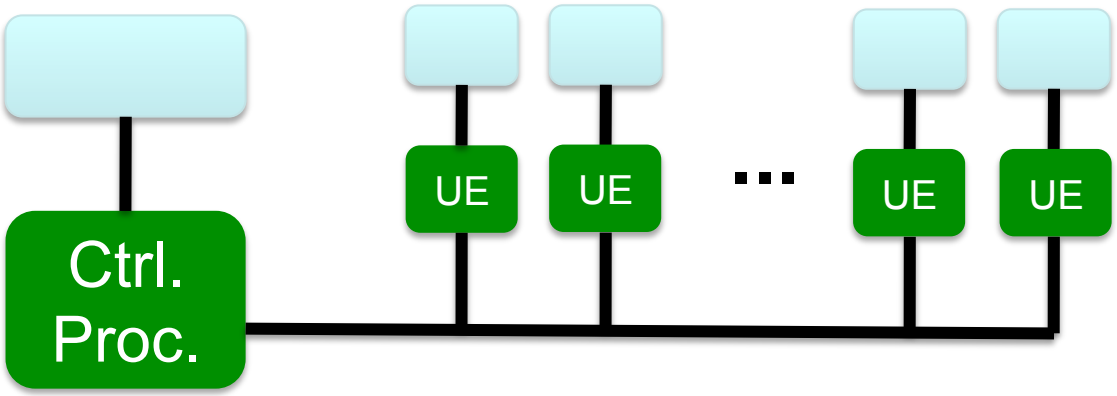
  - A Workstation hosted SIMD machine.
  - A node consists of a two processor chip pair (32 PEs) and an optional floating point accelerator.
  - Topology --- The nodes are connected as a hypercube.
  - Performance --- peak performance of 40 GFLOPS for the largest CM-200 (65536 PEs) with floating point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



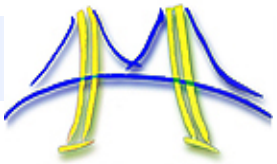


Modern Data-Parallel Machines

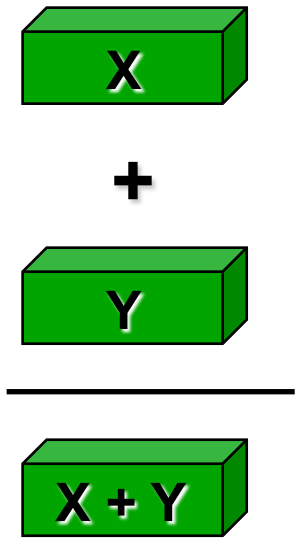
- SIMD in CPUs: The CPU pipeline is the "frontend", executing a sequential program and issuing commands to the SSE or AVX processor "array"
- SIMD in CPU-GPU systems: The CPU Host is the "frontend machine", issuing SIMD Kernel commands to the GPU "array" of Streaming Multiprocessors
- SIMD in GPUs: The 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)

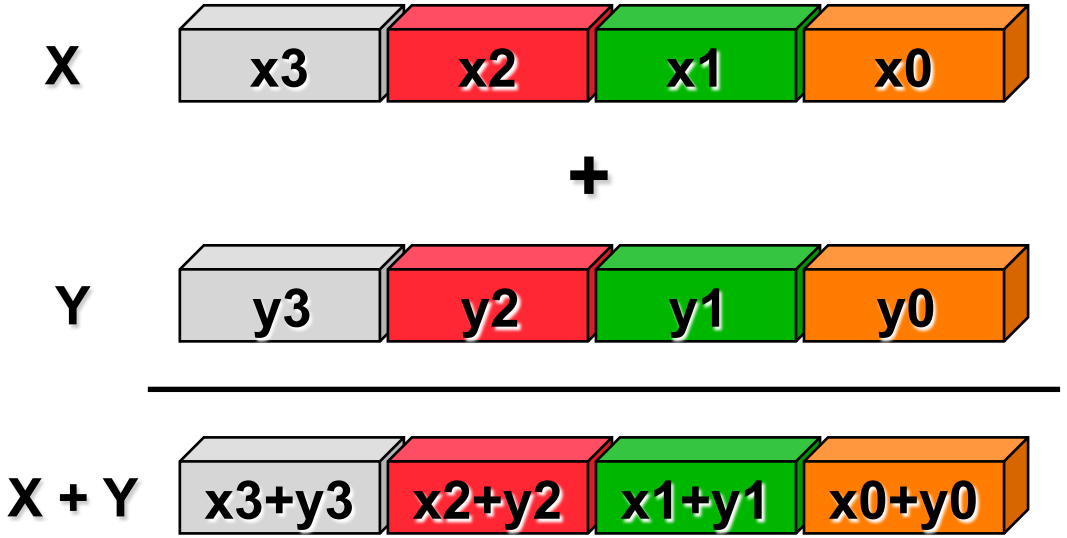


Pseudo SIMD: (Poor-Man's SIMD?)



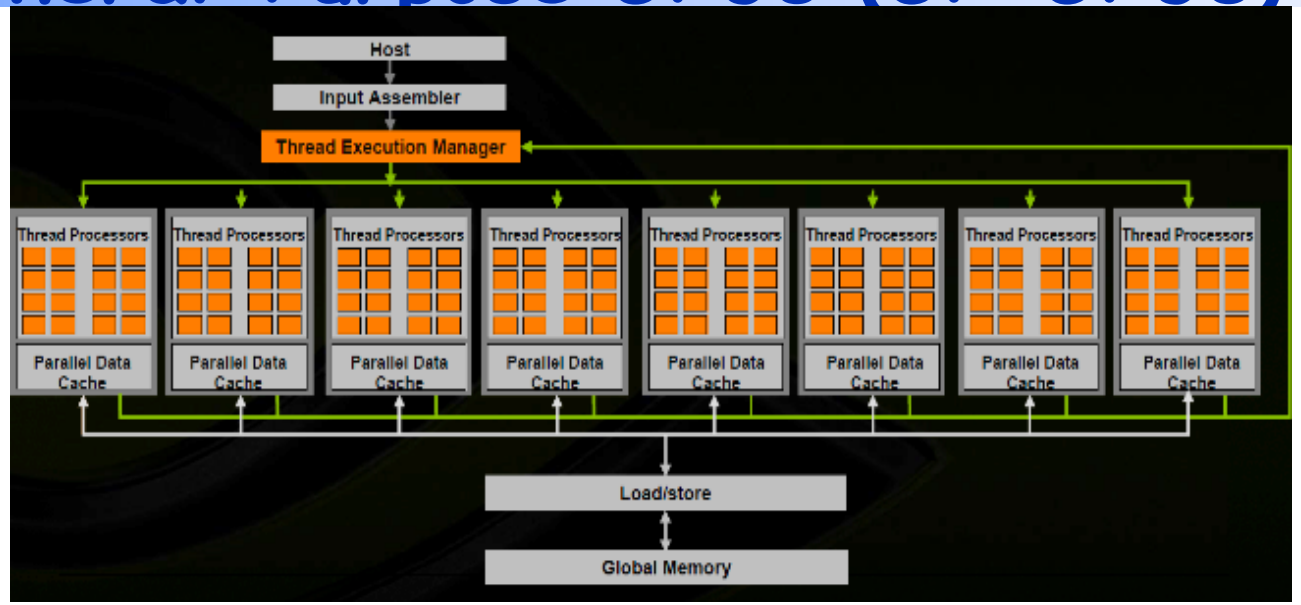
- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing (Intel)
  - with SSE / SSE2
  - one operation produces multiple results



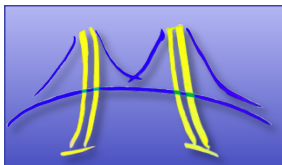


Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

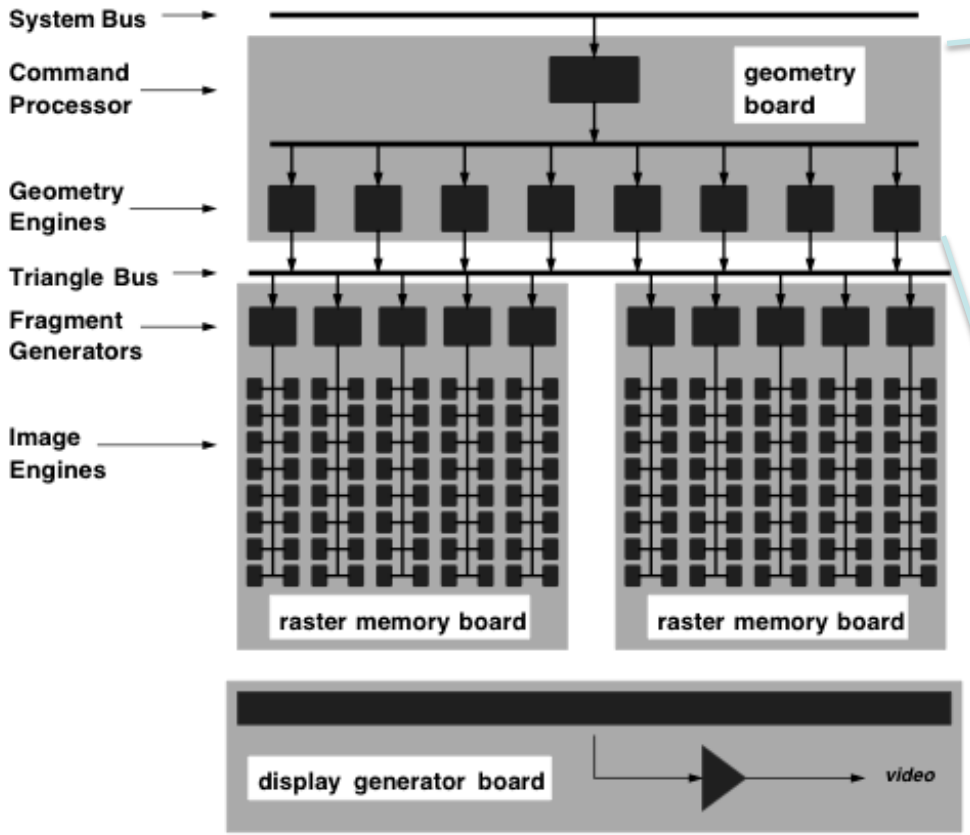
General-Purpose GPUs (GP-GPUs)



- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
  - "Compute Unified Device Architecture"
  - OpenCL is a vendor-neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution

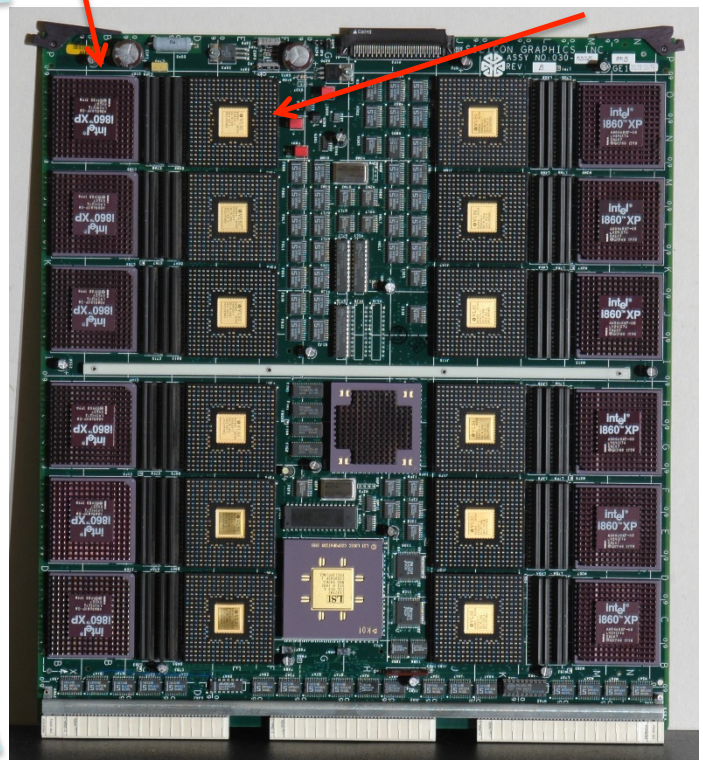


High-end GPUs have historically been programmable



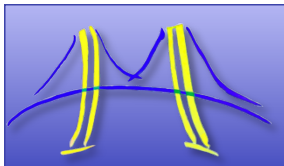
Intel i860 RISC CPU

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

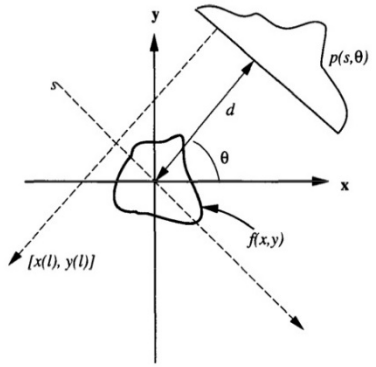


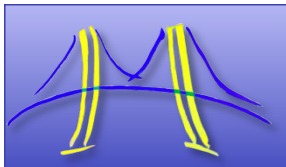
Figure 1: The Radon transform represents a generalized line integral projection of a 2-D (or 3-D) function $f(x, y, z)$ onto a line or plane.

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.

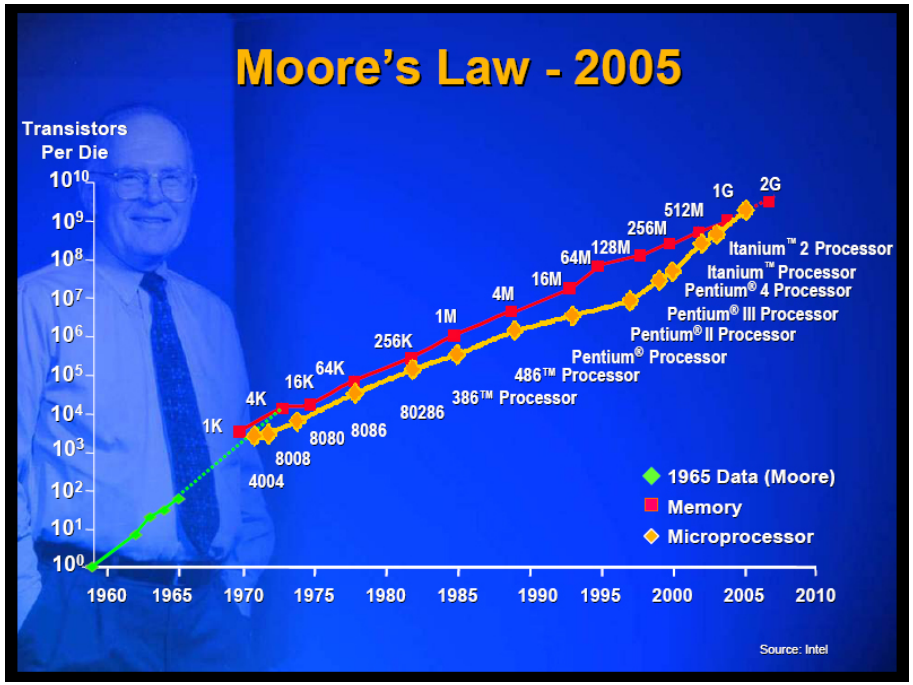
The Radon transform defines a mapping between the physical object space (x, y) and its projection space (s, θ), as illustrated in figure 1. The object is defined in a Cartesian coordinate system by $f(x, y)$, which describes the x-ray absorption or attenuation at the point (x, y) in the object at a fixed 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



From programmable workstation GPUs to programmable desktop GPUs

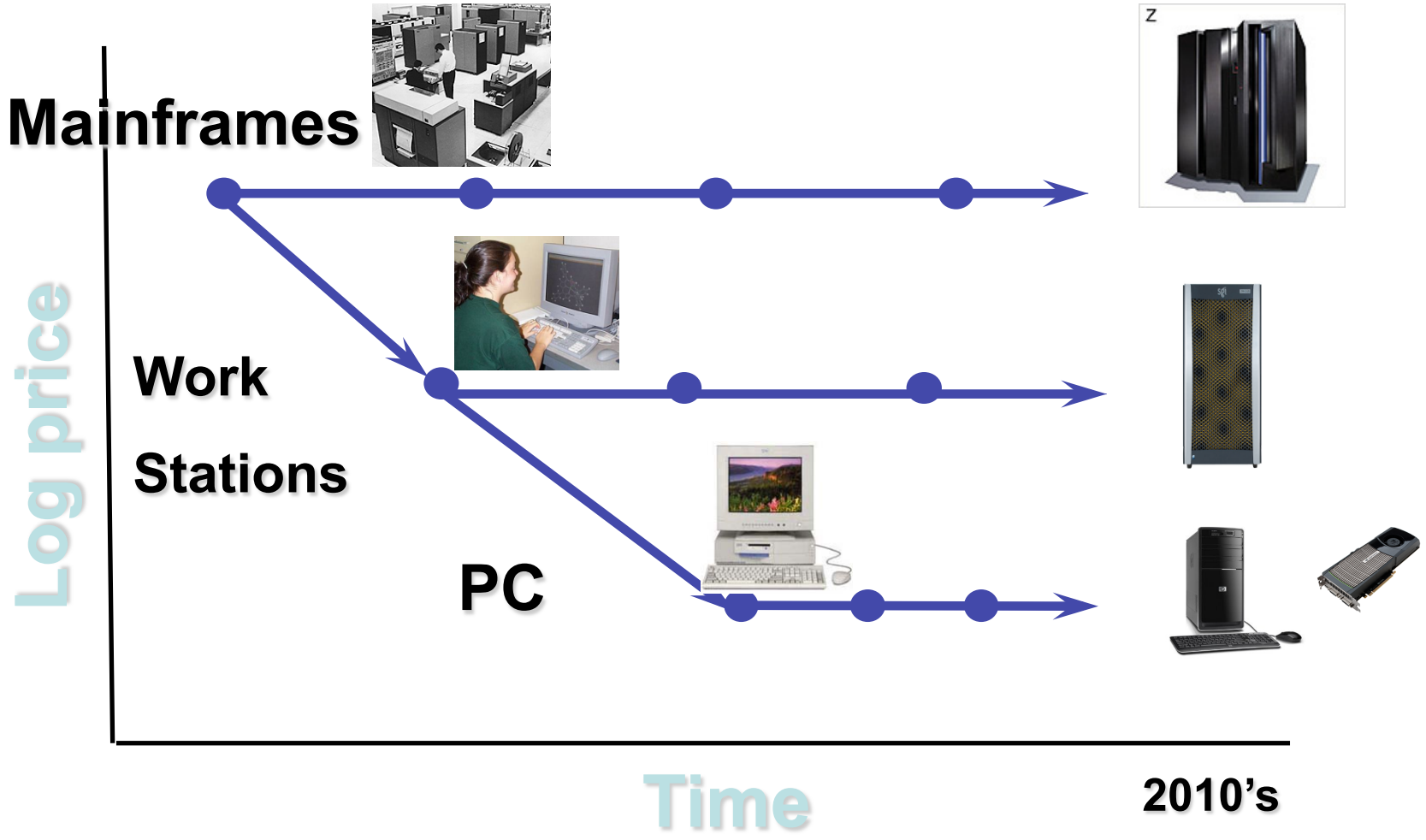
- Why did programmable graphics evolve on the desktop?
  1. Moore's Law made it possible
  2. Users demanded more compelling 3D graphics
- But maybe Bell's Law predicted this ahead of time…



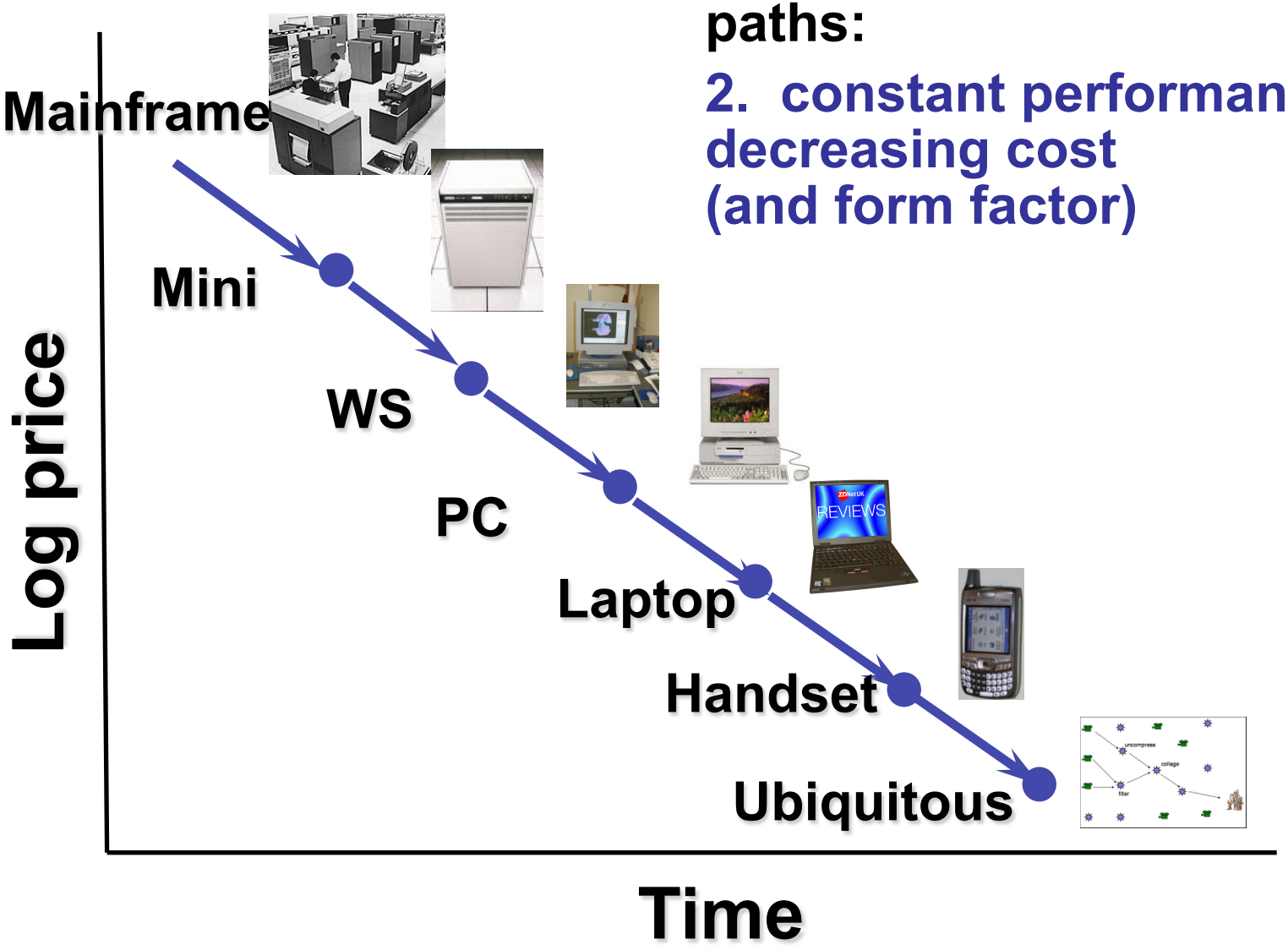
Bell's Law Of Computer Classes

Technology enables two paths:

1. constant price, increasing performance

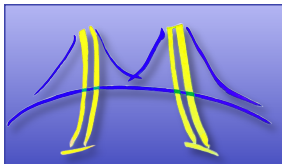


Bell's Evolution Of Computer Classes



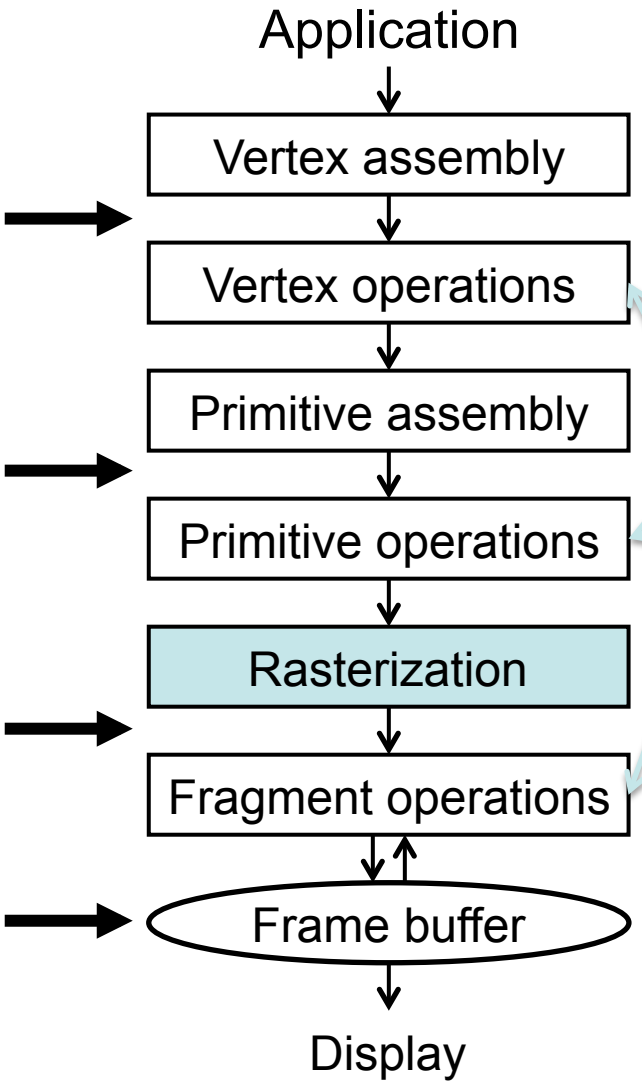
Technology enables two paths:

2. constant performance, decreasing cost (and form factor)



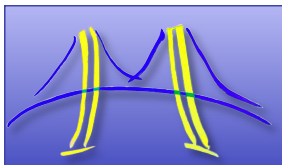
The Graphics vertex pipeline

```
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;
```



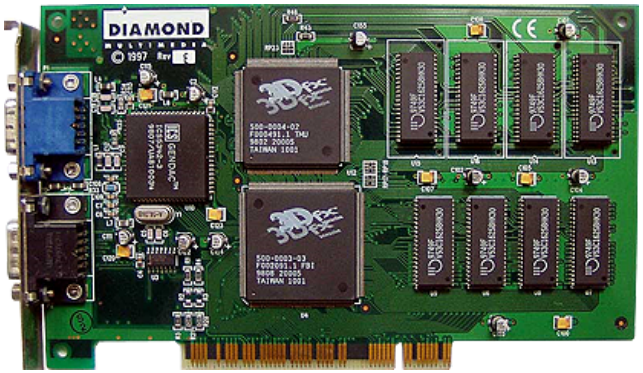
Wouldn't be cool to make these stages of the graphics pipeline programmable?

Thanks to Kurt Akeley

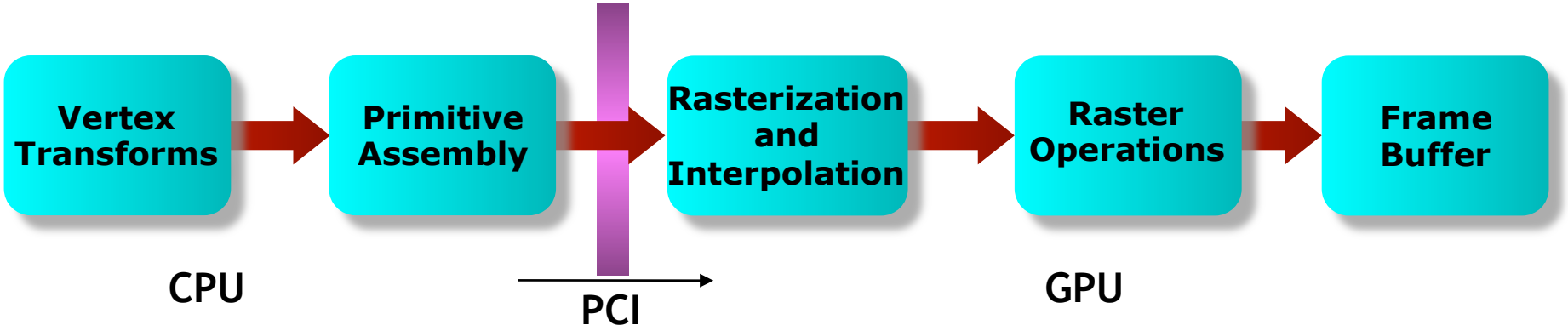


Generation I: 3dfx Voodoo (1996)

Diamond Multimedia
Monster3D

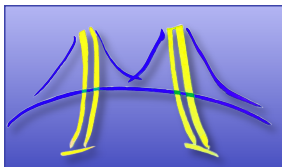


- Did not do vertex transformations: these were done in the CPU
- Did do texture mapping, z-buffering.
- 0.5 micron technology
- 1M transistors



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

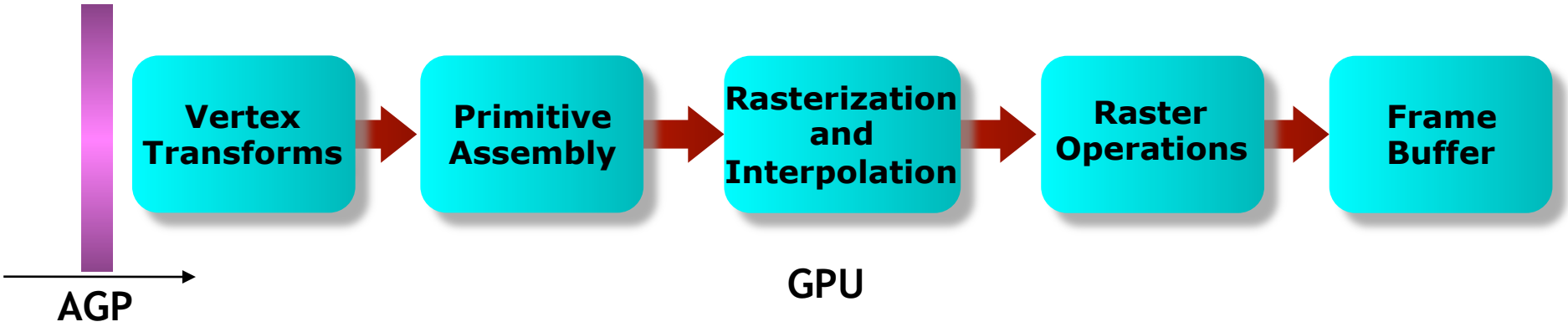


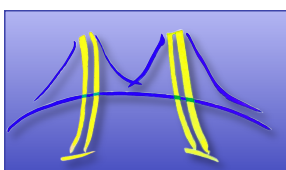
Generation II: GeForce 256/Radeon 7500 (1998)

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

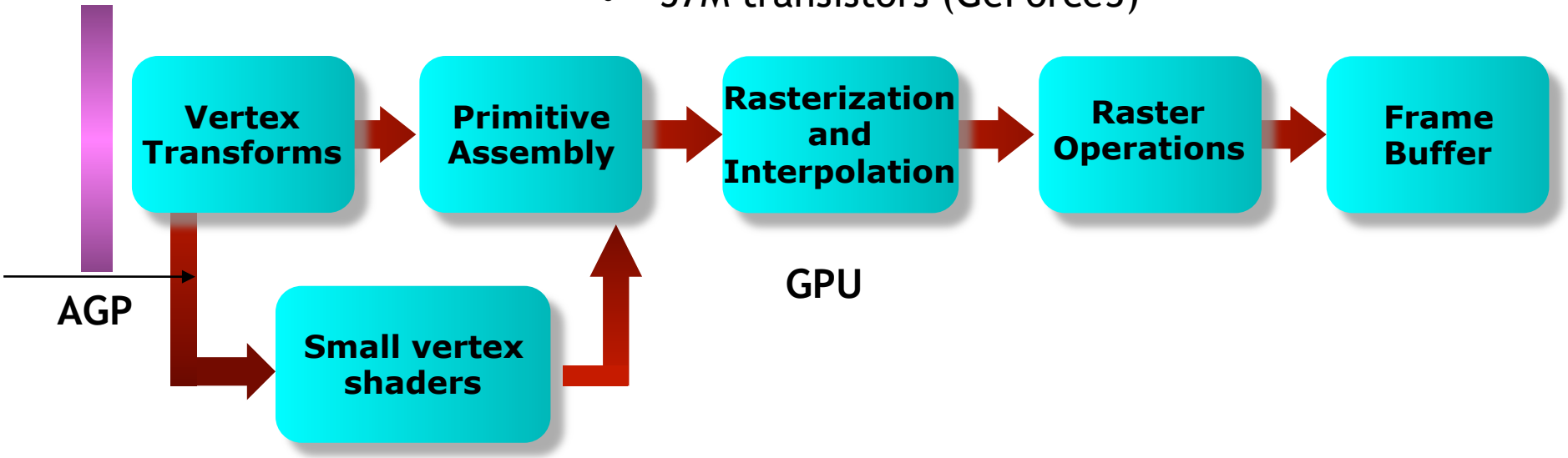


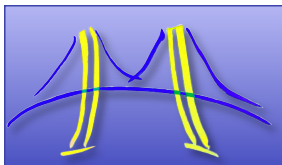


Generation III: GeForce3/Radeon 8500(2001)

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 IV: Radeon 9700/GeForce FX (2002)

GeForce FX

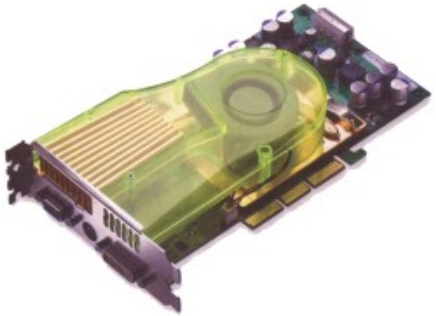
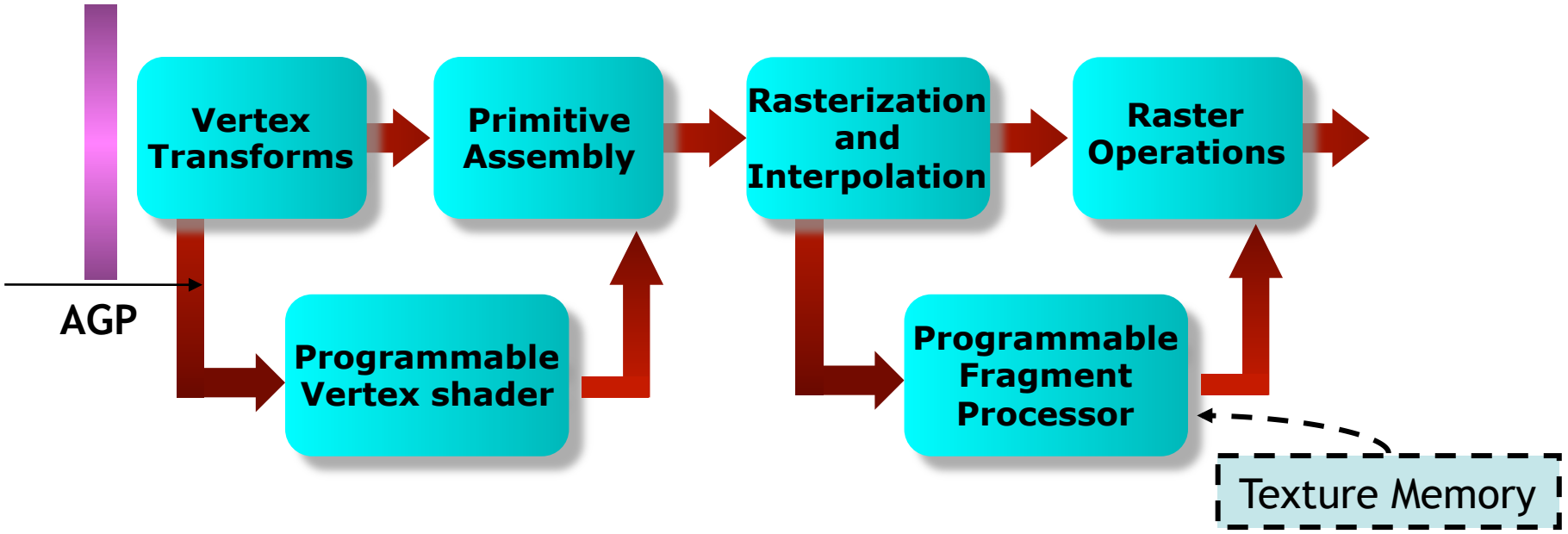


Image from "7 years of Graphics"

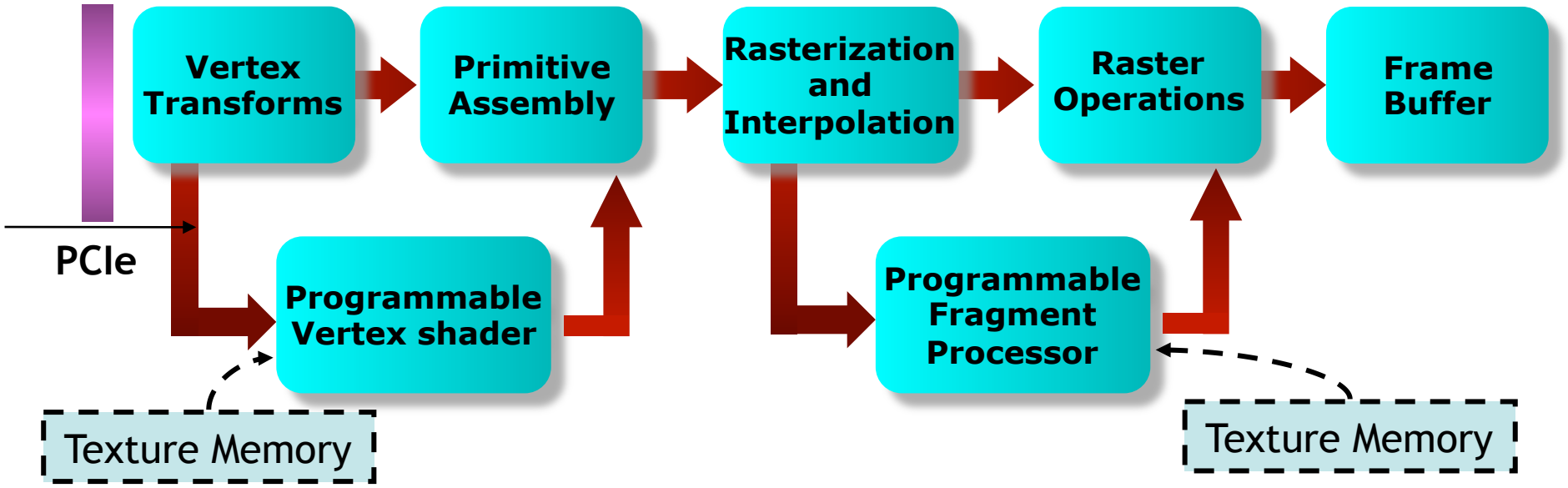
- This generation is the first generation of "fully-programmable" graphics cards
- DirectX 9 (shader model 2.0)
- Different versions have different resource limits on fragment/vertex programs
- 0.13 micron technology node
- 80M transistors

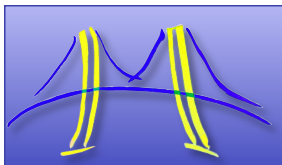


Slide from Suresh Venkatasubramanian and Joe Kider

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

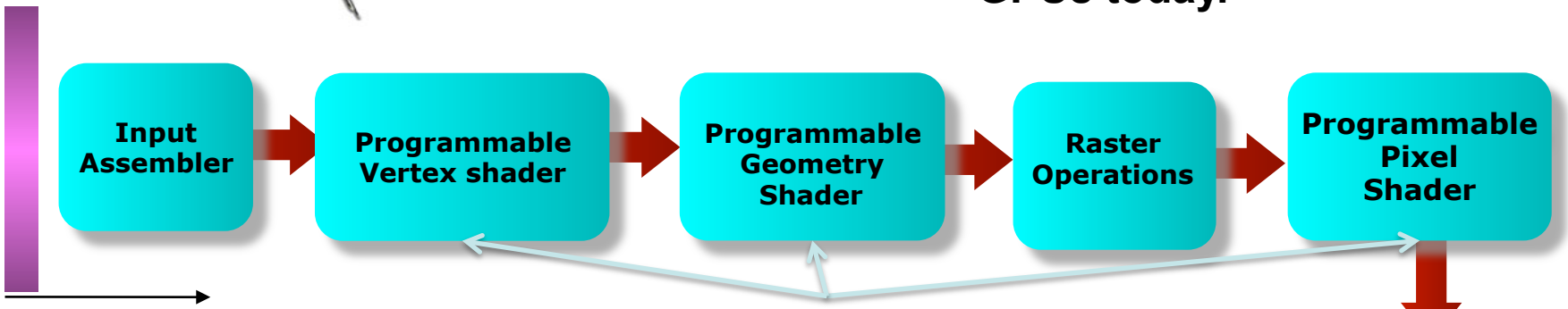




Generation V: GeForce8800/HD2900 (2006)

- 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***

We're still using generation 5 GPUs today.



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

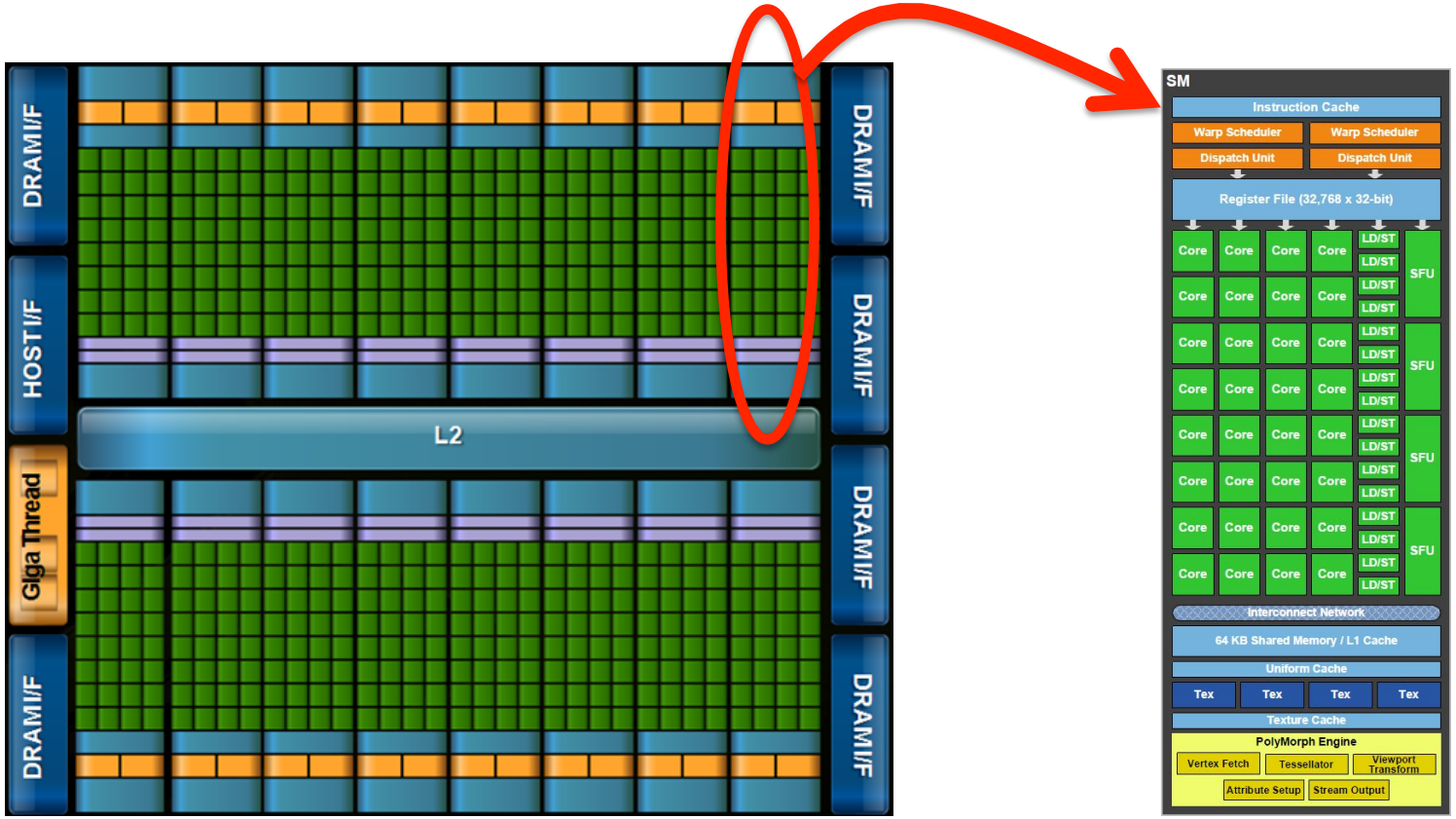
GPGPU arrives: 2006

- GeForce 8800/HD2900:
  - Ground-up GPU redesign
  - Support for Direct3D 10
  - Geometry Shaders
  - Stream out / transform-feedback
  - Unified shader processors
- ***Support for General GPU programming***

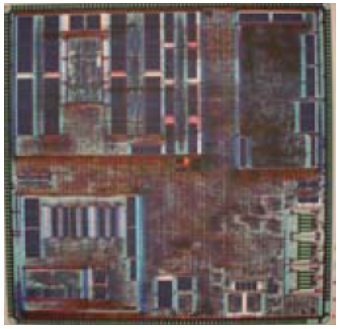
- Fortunately for NVIDIA, the academic community had been working on GPGPU programming for almost a decade.
- Ian Buck at Stanford was wrapping up his dissertation "Stream computing on Graphics Hardware" and the language "Brook".
- He moved over to NVIDIA and led the effort to create CUDA.
- CUDA was 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.

Nvidia GPU Architecture

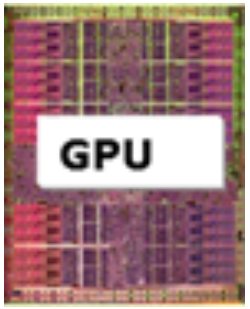
- Nvidia GPUs are a collection of “Streaming Multiprocessors”
  - Each SM is analogous to a core of a Multi-Core CPU
- Each SM is a collection of SIMD execution pipelines that share control logic, register file, and L1 Cache

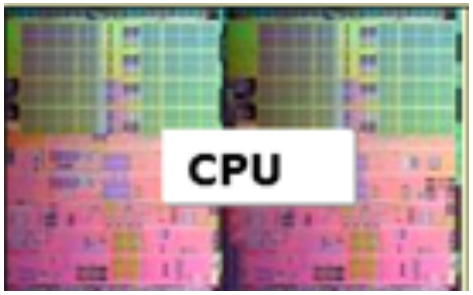


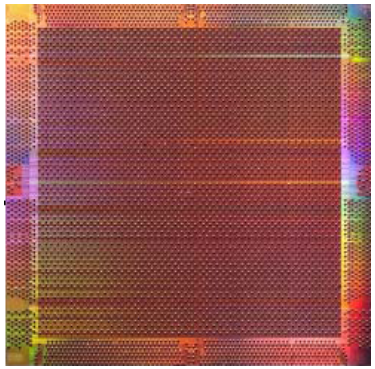
The heterogeneous platform: a Host (CPU) + a huge range of devices



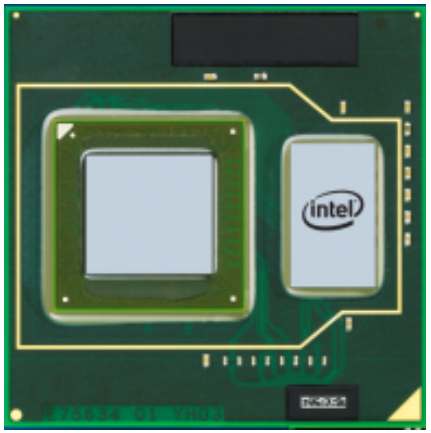
DSP





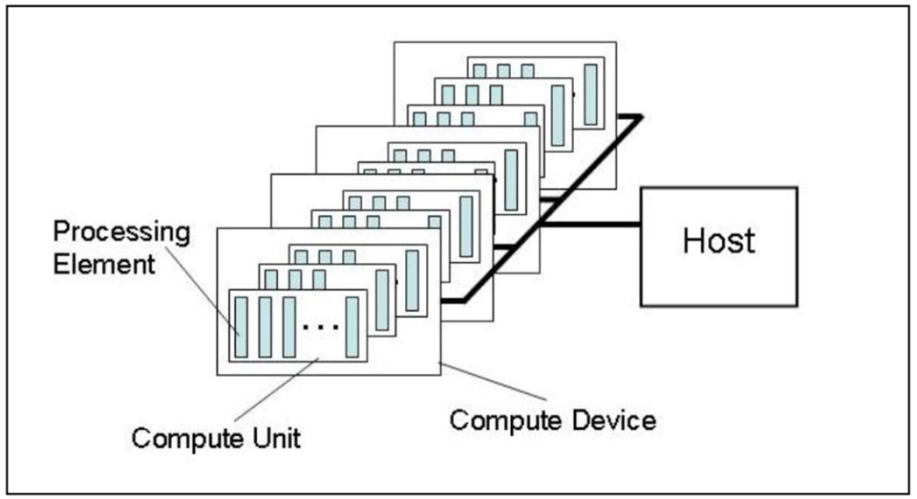


FPGA



Integrated CPU+ FPGA

(Intel® Atom™ Processor E6x5C Series)



… and who knows what the future will bring?

Many-core CPU (MIC & Xeon Phi™)

The BIG idea behind OpenCL
(and CUDA and the others)

- 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];
}
```

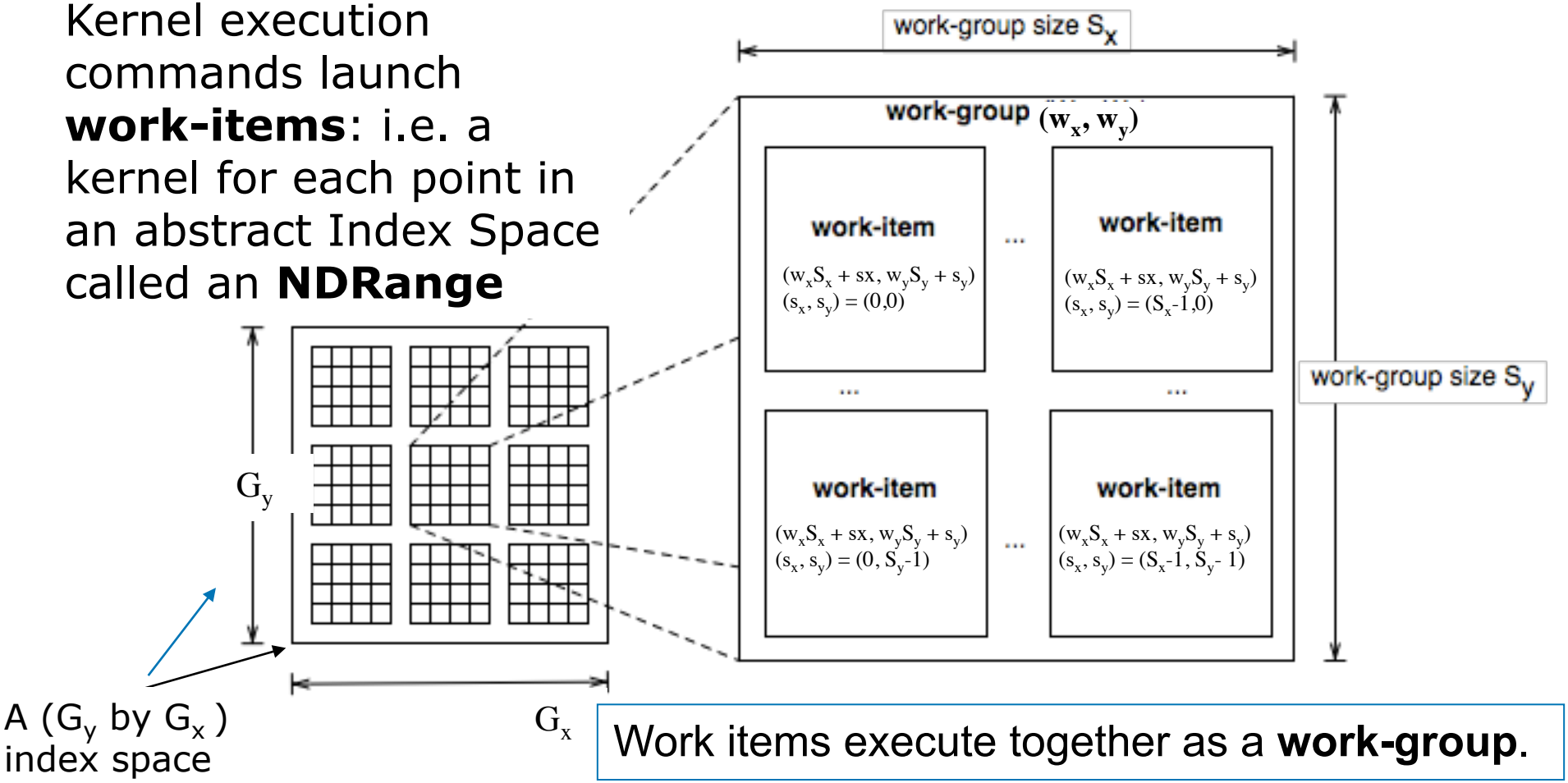
Kernel Parallelism OpenCL

```
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**.

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

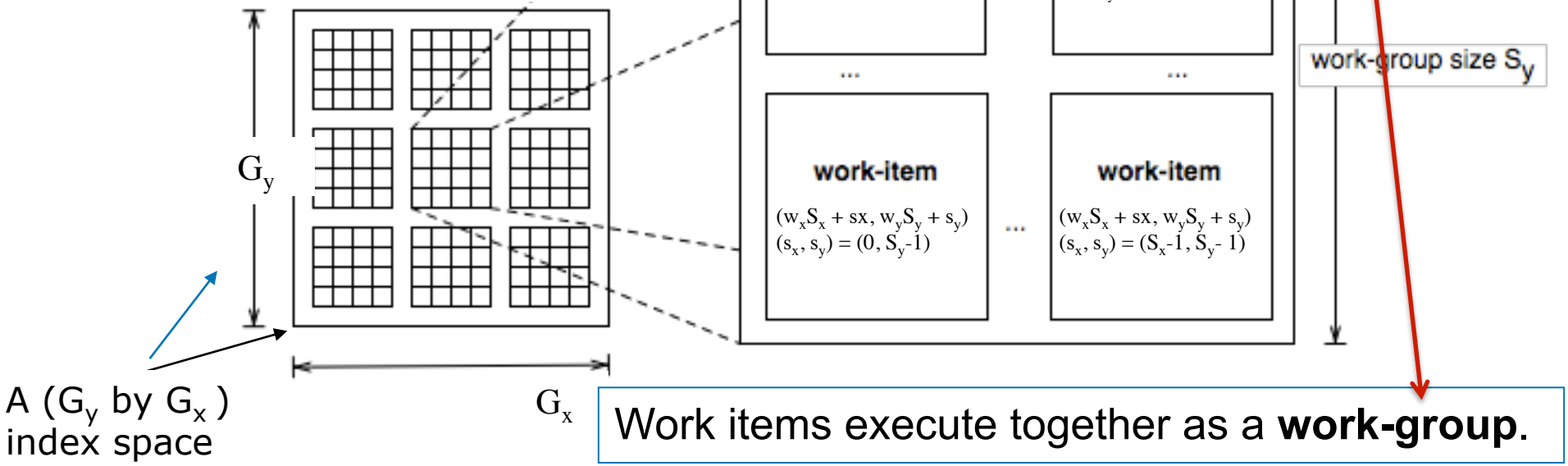
CUDA Stream

Thread Block

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

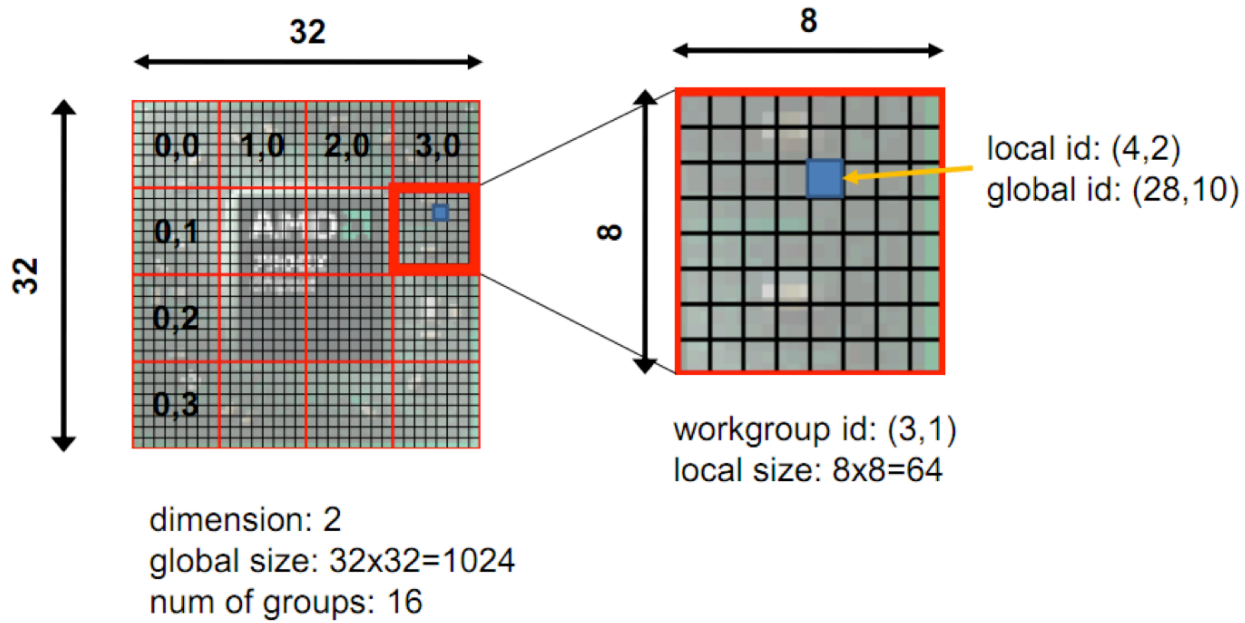
Threads

Grid



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

OpenCL Thread Hierarchy

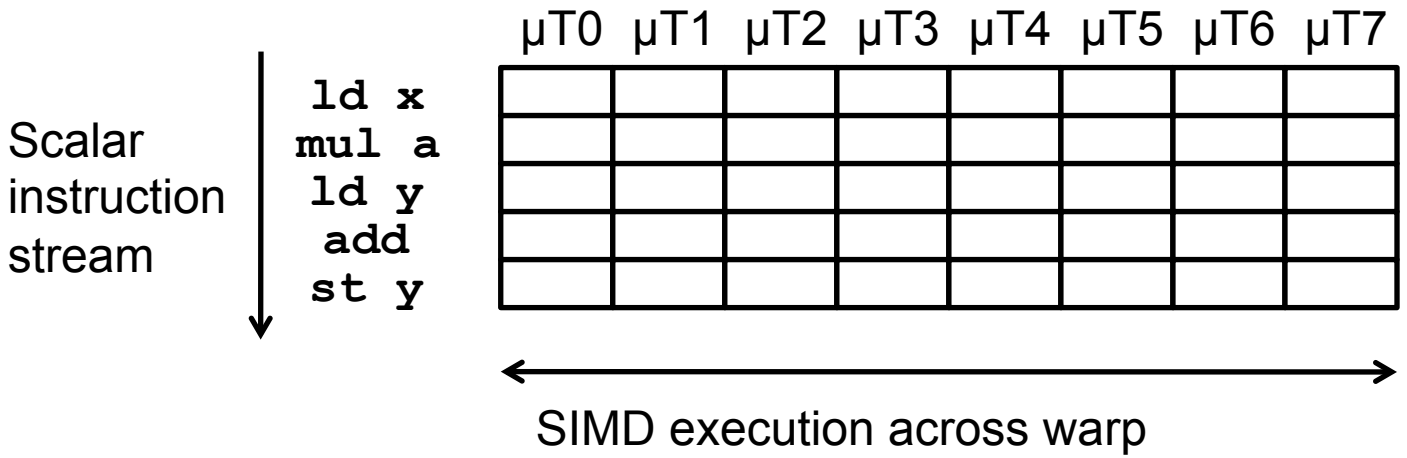


- Parallelism in OpenCL is expressed as a 3-level Hierarchy
- An ***Index Space*** is a collection of up to 65,535 x 65,535 ***Work Groups***.
- A ***Work Group*** is a collection of up to 512 [1024 on Fermi] ***Work Items***
  - All threads in a thread block execute concurrently, and can synchronize via a barrier intrinsic, communicate via shared memory
- Each ***Work Items*** executes independently, and communicate with the rest of the Block
- Groups of 32 ***Work Items*** execute in SIMD as ***Warps***

Image from http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

“Single Instruction, Multiple Thread”

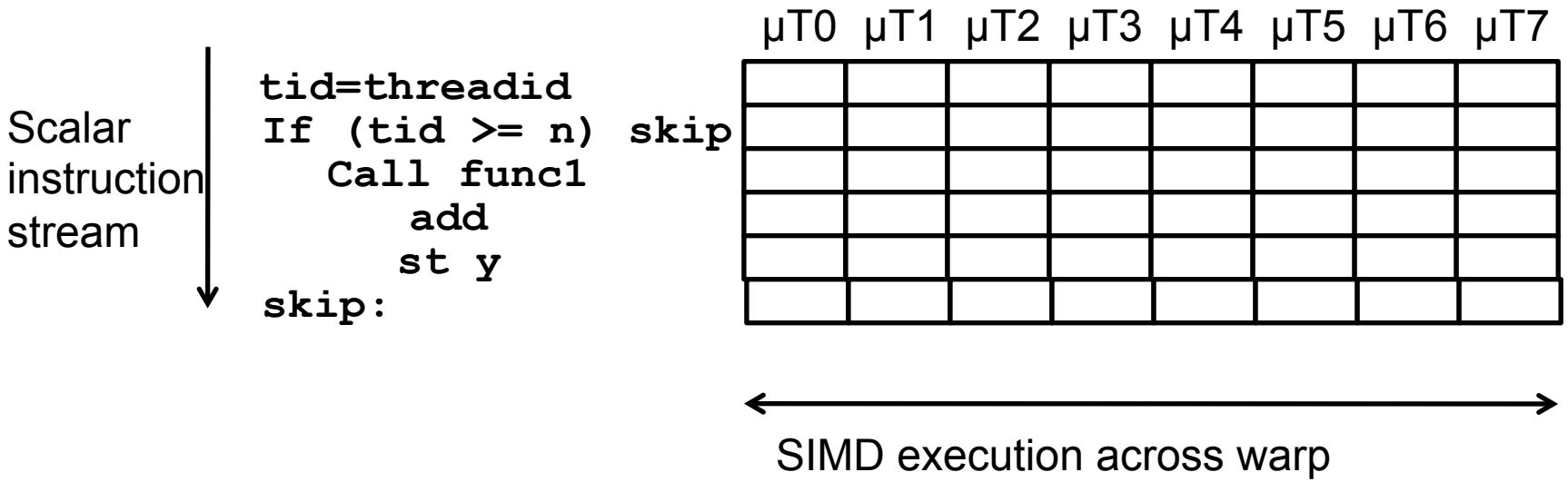
- GPUs use a SIMT model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (Nvidia groups 32 CUDA threads into a *warp*)



Slide thanks to Krste Asanovic

Conditionals in SIMT model

- Simple if-then-else are compiled into predicated execution, equivalent to vector masking
- More complex control flow compiled into branches
- How to execute a vector of branches?



Slide thanks to Krste Asanovic

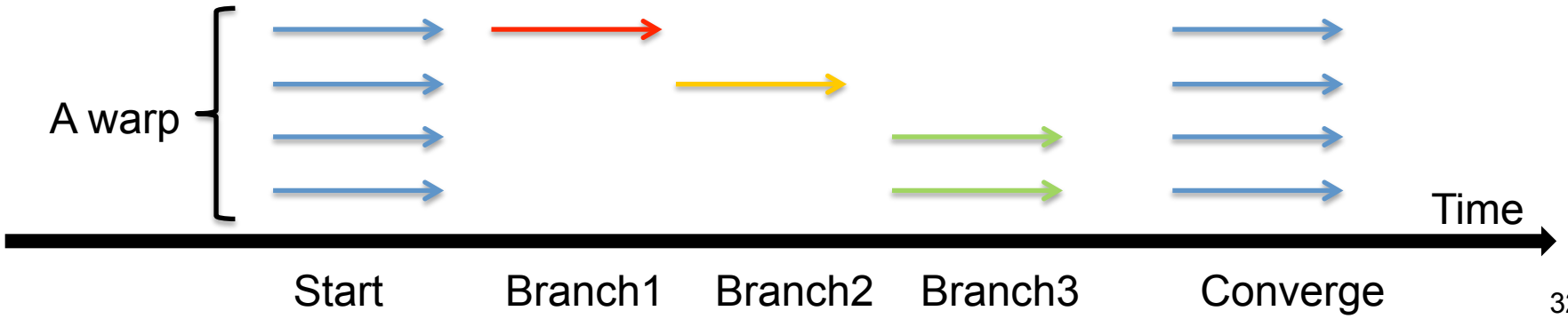
Branch divergence

- Hardware tracks which µthreads take or don't take branch
- If all go the same way, then keep going in SIMD fashion
- If not, create mask vector indicating taken/not-taken
- Keep executing not-taken path under mask, push taken branch PC+mask onto a hardware stack and execute later
- When can execution of µthreads in warp reconverge?

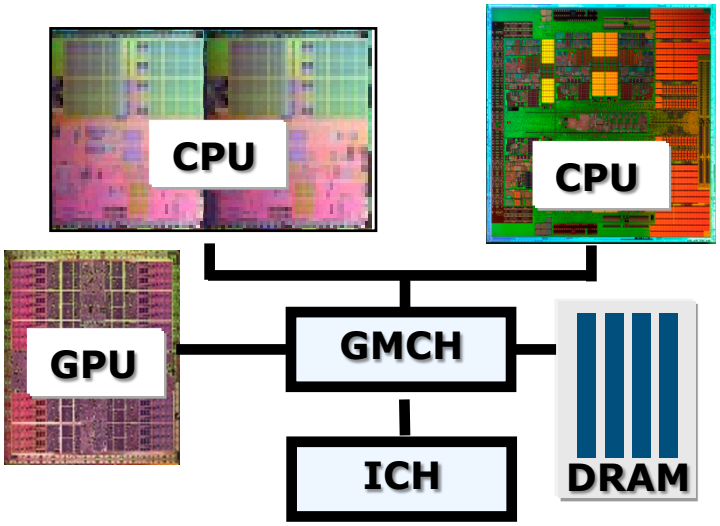
Slide thanks to Krste Asanovic

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

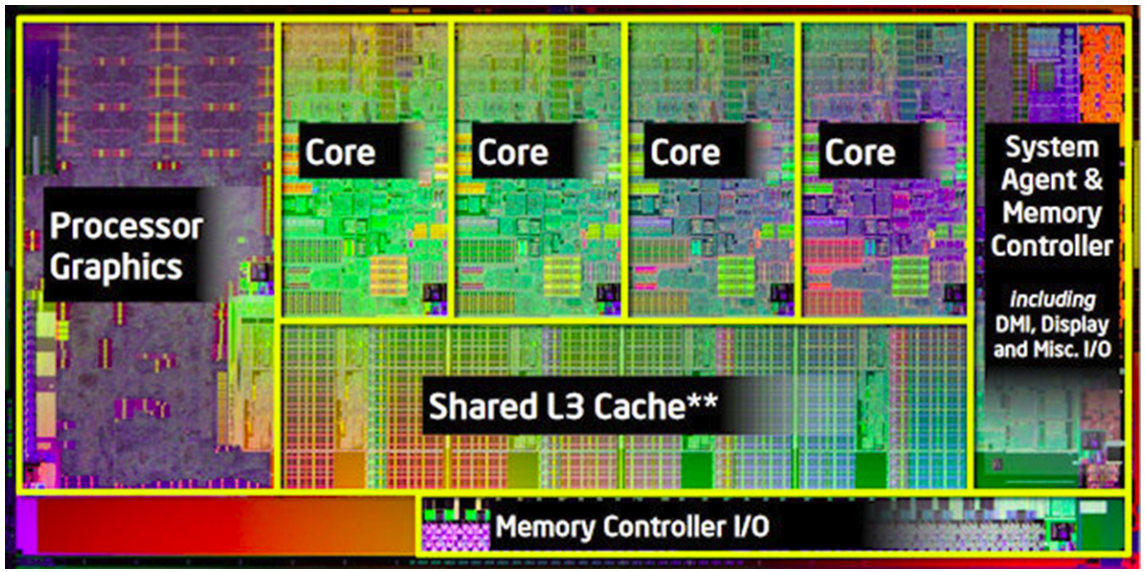


Our HW future is clear:



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

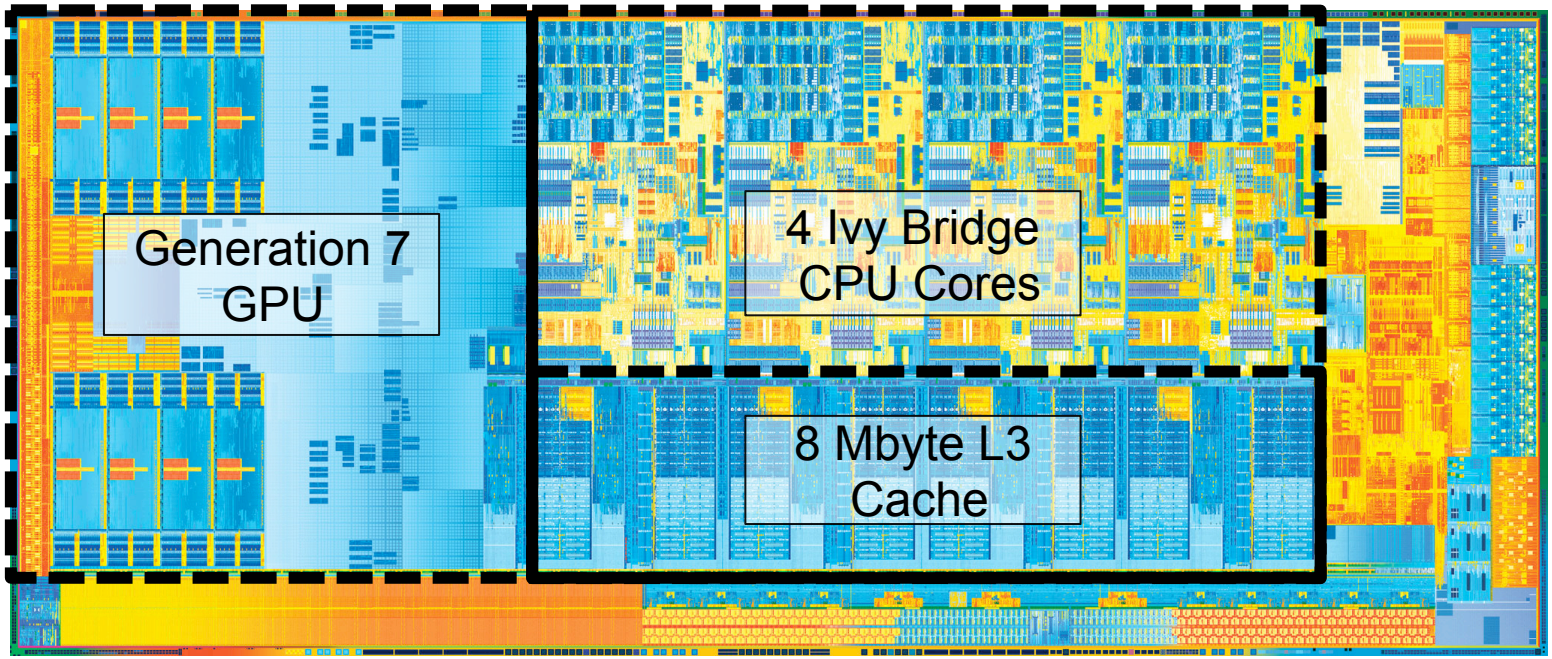
- Current designs put this functionality onto a single chip … mitigates the PCIe bottleneck in GPGPU computing!



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

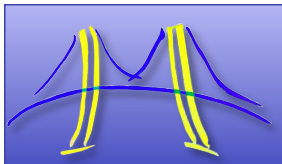
GMCH = graphics memory control hub,
ICH = Input/output control hub

Intel's Ivy Bridge GPU

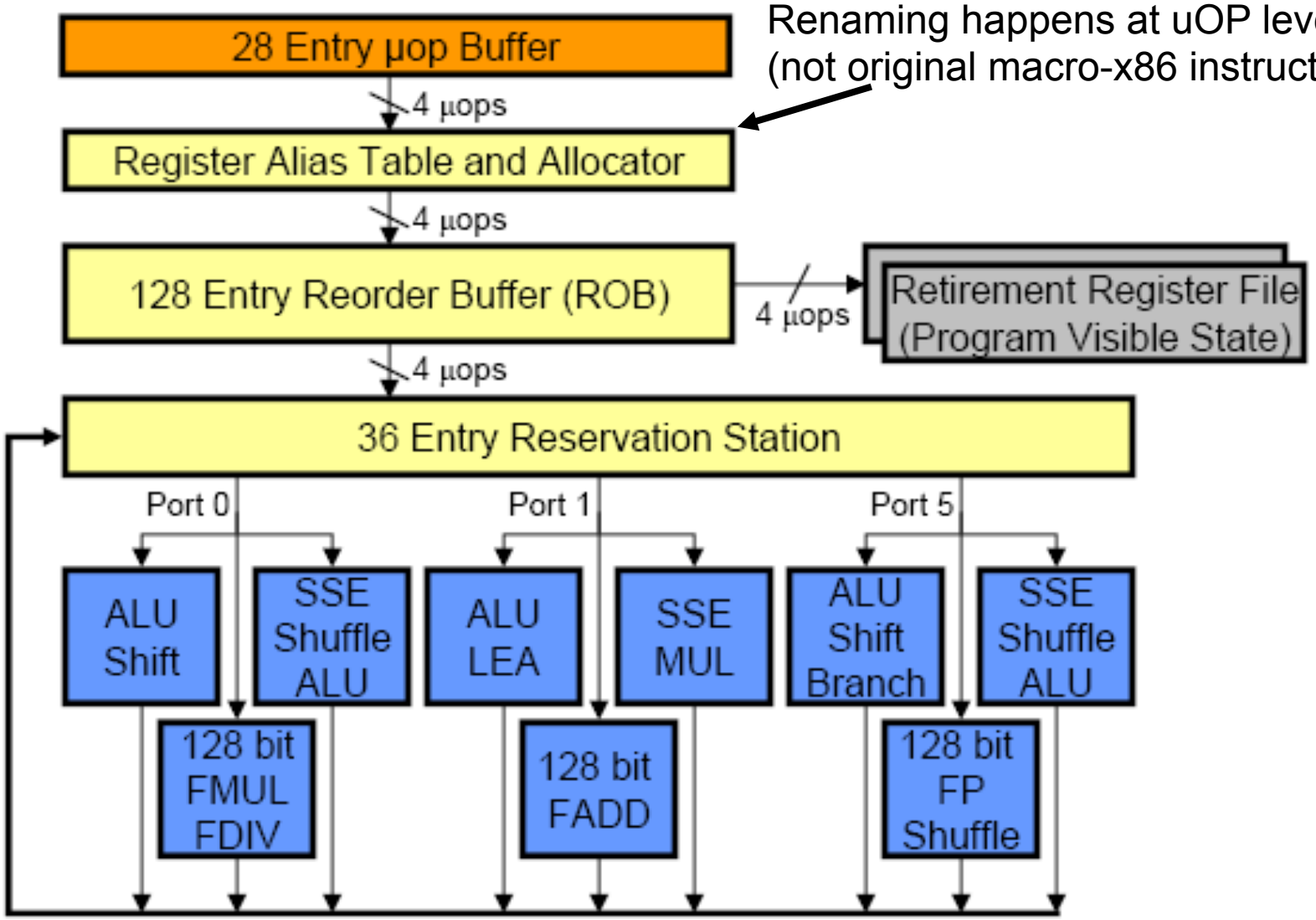


- 4 CPU cores + GPU
  - All integrated on the same die
  - GPU and aggregate CPUs have about the same peak performance
    - 256 single-precision Gflops/sec
- GPU is fully programmable with OpenCL
  - DirectX 11 too

Thanks to David Kanter of Real World Tech and Intel

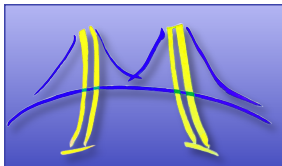


Out-of-Order Execution Engine

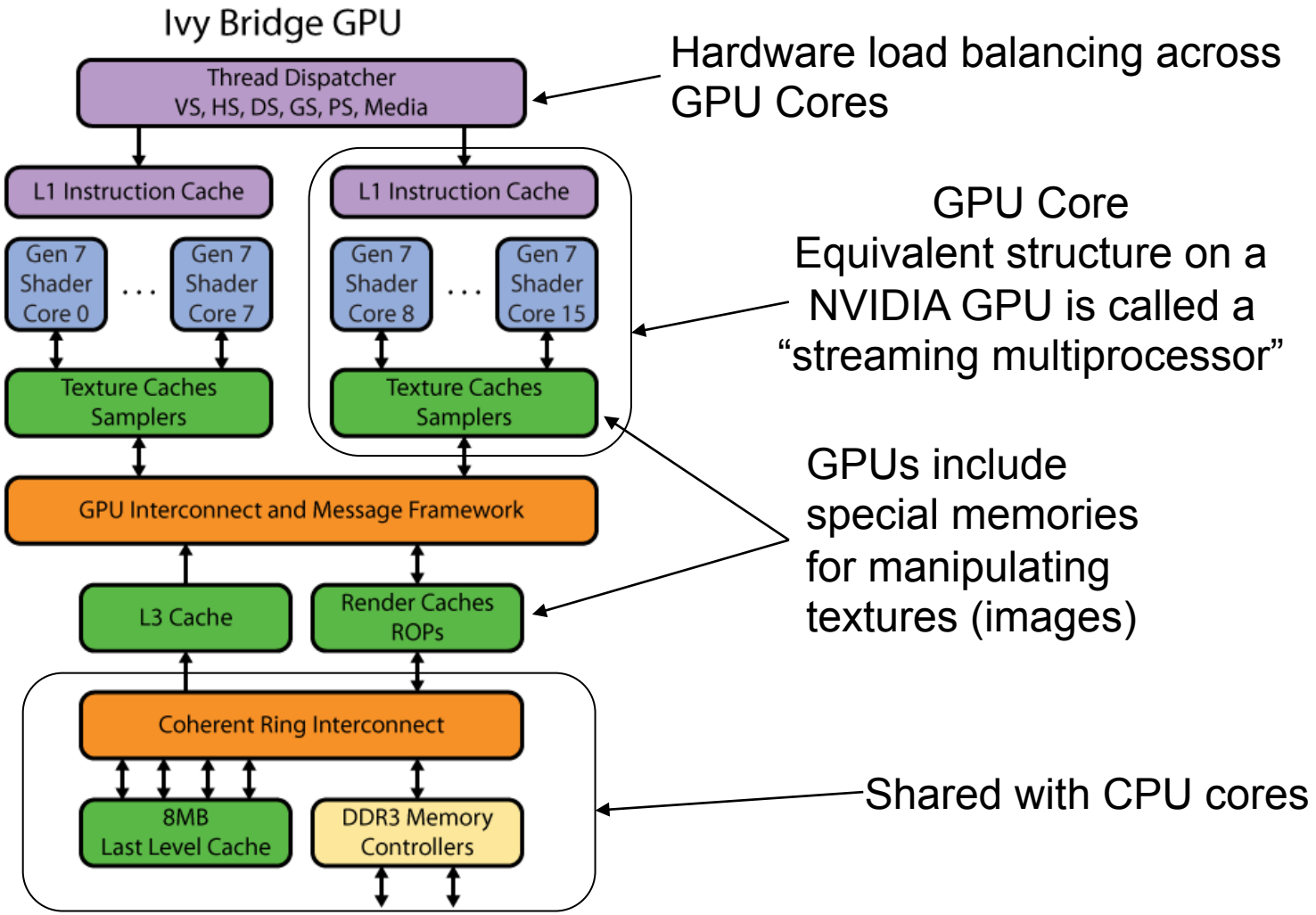


Renaming happens at uOP level
(not original macro-x86 instructions)

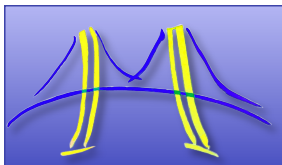
Thanks to David Kanter of Real World Tech, Intel, and Krste Asanovic of UCB



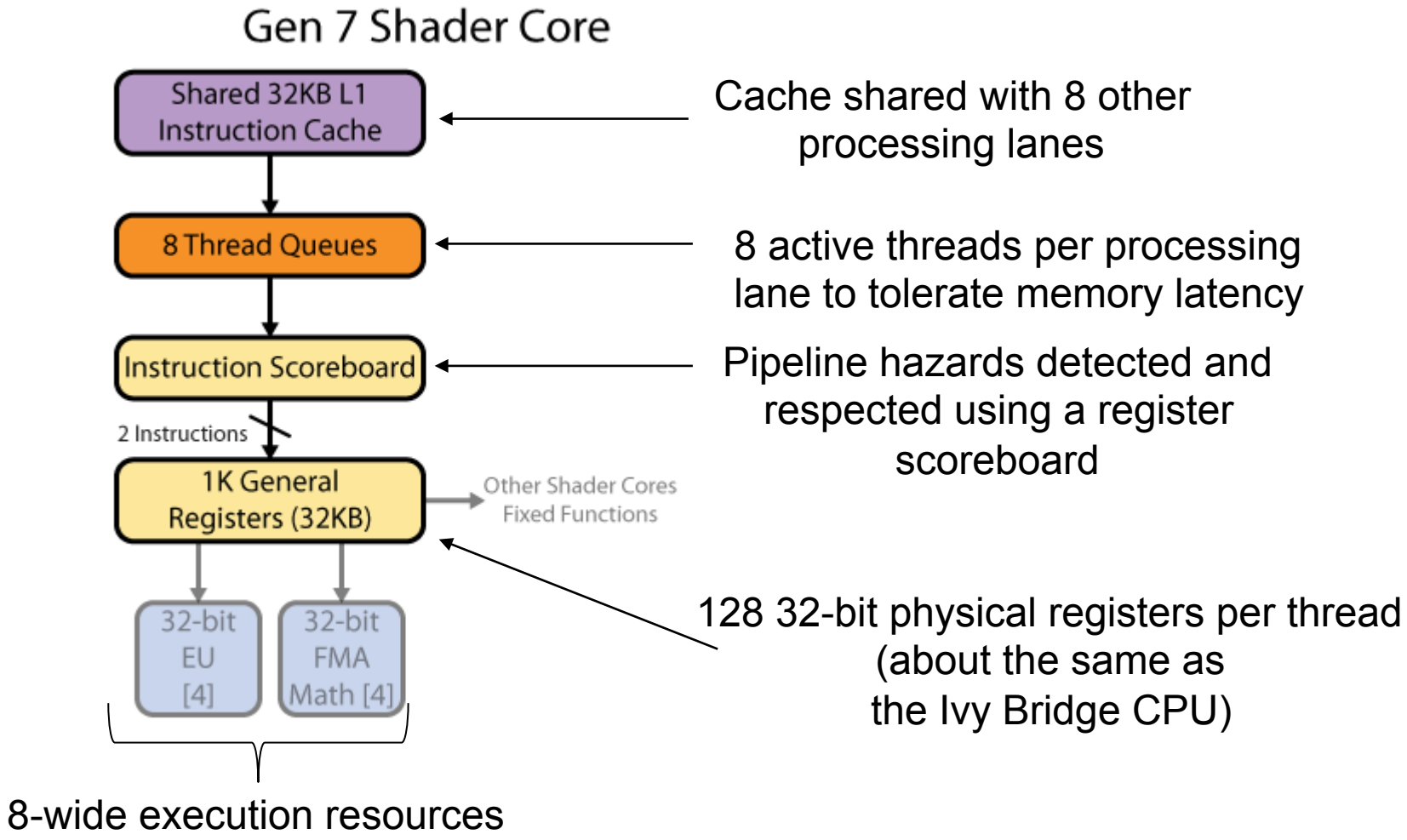
Ivy Bridge System Architecture



Thanks to David Kanter of Real World Tech and Intel



Ivy Bridge GPU Core Microarchitecture

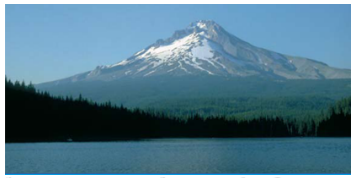


Thanks to David Kanter of Real World Tech and Intel

OpenACC

Directive driven programming of Heterogeneous systems

- Portland group (PGI) introduced proprietary directives for programming GPUs



- OpenMP (with help from PGI) launched a working group to define “accelerator directives” In OpenMP.

OpenMP

- A subset of the participants grew tired of the cautious, slow and methodical approach in the OpenMP group … and split off to form their own group (OpenACC)
  - NVIDIA, Cray, PGI, and CAPS

OpenACC
DIRECTIVES FOR ACCELERATORS

- They launched the OpenACC directive set in November of 2011 at SC11.
- At SC12:
  - The OpenACC group released a review draft of OpenACC 2.0
  - The OpenACC and OpenMP groups stated their intent to rejoin the 2 efforts.
- Summer 2013 … OpenMP released OpenMP 4.0 which includes accelerator directives for functionality analogous to OpenACC.
- Fall’2013 … gcc annouces work on OpenACC support. Should be ready soon.

OpenACC
DIRECTIVES FOR ACCELERATORS

Source: John Levesque of Cray

- A common directive programming model for today's GPUs
  - Announced at SC11 conference
  - Offers portability between compilers
    - Drawn up by: NVIDIA, Cray, PGI, CAPS
    - Multiple compilers offer portability, debugging, permanence
  - Works for Fortran, C, C++
    - Standard available at www.OpenACC-standard.org
    - Initially implementations targeted at NVIDIA GPUs
- Current version: 2.0 (November 2012)
- Compiler support:
  - Cray CCE: complete support in 2012
  - PGI Accelerator: released product in 2012
  - CAPS: released product in Q1 2012

CAPS CRAY THE SUPERCOMPUTER COMPANY NVIDIA The Portland Group

OpenACC: Core concepts

- Pragmas direct the compiler to generate code to run on the host and the GPU. Basic form of an OpenACC directive:

  #pragma acc construct [clause(s)]

- Two core constructs define work that runs on the accelerator (coarse grained, gang parallelism):
- Parallel construct:
  - Tells compiler to create a single kernel to accelerate the code.
  - This is explicit, analogous to the parallel region in OpenMP.
  - Clauses can be used with the parallel construct

    num_gang, num_worker, vector_length
  - Kernel construct:
    - Tells the compiler to accelerate the code with one or more kernels.
    - This is NOT explicit. If the compiler deems the code unsafe for execution on the GPU, it will not execute on the accelerator.

```
#pragma acc parallel
for(i=0;i<N;i++)
   A[i] = B[i]+C[i];
```

Code in block defines a kernel which runs in **gang-redundant mode** … one worker and one vector lane per gang redundantly executes the code.

OpenACC loop construct

- Typically programmers want to split loops between gangs (gang partitioned mode). The loop construct does this:

```
#pragma acc parallel loop
for(i=0;i<N;i++)
    A[i] = B[i]+C[i];
```

Loop iterations spread out across the units of execution on the accelerator.

- Can have multiple loop constructs in a single parallel region

```
#pragma acc parallel
{
#pragma acc loop
for(i=0;i<N;i++)
    A[i] = 2*A[i];

#pragma acc loop
For(i=0;i<N;i++)
    A[i] = B[i]+C[i];
}
```

Warning: there is NO IMPLIED barrier between loop constructs (i.e. acts like “nowait” in OpenMP)

Loop clauses

- Clauses can be used to direct loop scheduling … the parallel region defines gangs, workers and vectors and these clauses to the loop construct split up iterations at the indicated level

  Gang, worker, vector

- Connections between OpenACC scheduling clauses and CUDA

OpenACC	CUDA
gang	A thread block
worker	A warp (32 threads)
vector	Threads within a warp

- Reduction clause … same as OpenMP reduction

  reduction(op:vars)

OpenACC in a real program: the "vadd" program

- 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.

Exercise 1

- Goal
  - Verify that you can build and run an OpenACC program.
- Problem
  - Write your own simple vector add program
  - Insert the OpenACC construct and run on the GPU
  - Time the program … do you see any speedup relative to running the code on the CPU?
- Extra work
  - Experiment with the different scheduling clauses

```
#include <openacc.h>
#pragma acc parallel loop
#pragma acc parallel loop vector_length(64)
#pragma acc parallel loop reduction (+:var)
*restrict
```

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)

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)

```
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 ++;
}
```

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

Performance was poor due to excess memory movement overhead

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

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

Exercise 2

- Goal
  - Parallelize and then optimize the provided jacobi solver.
- Problem
  - Start simple … put “parallel loop” constructs in the right places.
  - Test and time the program.
  - Modify the data environment to improve performance
- Extra work
  - Experiment with the different scheduling clauses

```
#include <openacc.h>
#pragma acc data copy(A), create(Anew) firstprivate(tmp)
#pragma acc parallel loop reduction (+:var)
*restrict
```

OpenACC vs. OpenMP

- OpenACC suffers from a form of the CUDA™ problem … it is focused on GPUs only … and mostly those from NVIDIA.
- With the purchase of PGI by NVIDIA, the chances of long term cross platform support is reduced.
- OpenACC is an open standard (which is great) but its only a small subset of the industry … not the broad coverage of OpenMP or OpenCL.
- The OpenMP 4.0 accelerator directives:
  - Mesh with the OpenMP directives so you can use both in a single program.
  - They are designed to support many core CPUs (such as MIC), multicore CPUs, and GPUs.
  - Support a wider range of algorithms (though OpenACC 2.0 closes this gap).
- So … OpenMP directive set will hopefully displace the OpenACC directives as they are finalized and deployed into the market.

A more complicated example:

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 ++;
}
```

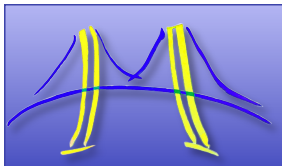
Create a data region on the GPU. Map A and Anew onto the target device

Uses existing OpenMP constructs such as parallel and for

Copy A back out to host … but only once

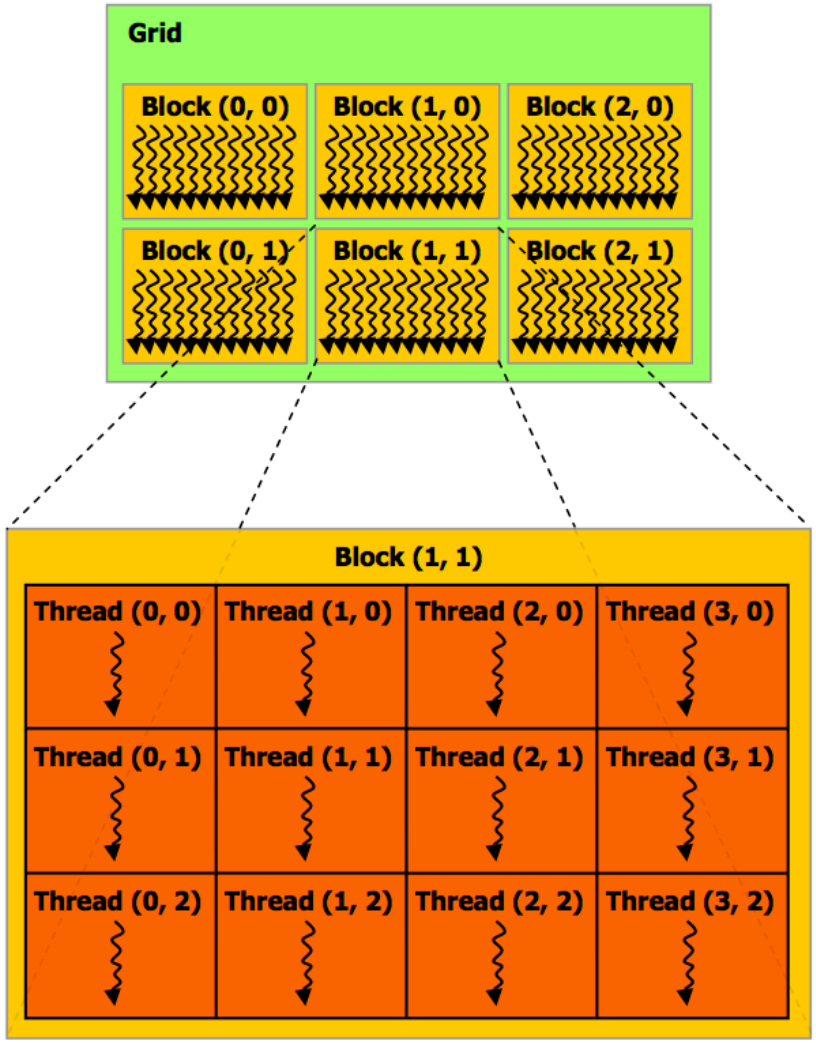
Introduction to CUDA

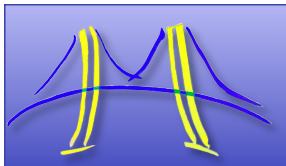
Acknowledgements: CUDA content from comes from David Sheffield, Michael Anderson, Kurt Keutzer, Mark Murphy, Bryan Catanzaro of UC Berkeley



What is a CUDA thread?

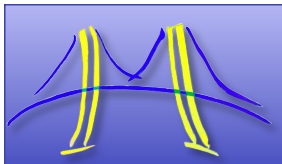
- Logically, each CUDA thread is its own very lightweight **independent execution context**
  - Has its own control flow and PC, register file, call stack, ...
  - Can access any GPU memory address at any time
  - Identifiable uniquely within a grid by the six integers: `threadIdx.{x,y,z}, blockIdx.{x,y,z}`
- **Very fine granularity**: do not expect any single thread to do a substantial fraction of an expensive computation
  - At full occupancy, each Thread has 21 32-bit registers
  - ... 1,536 Threads share a 48 KB L1 Cache / `__shared__` mem





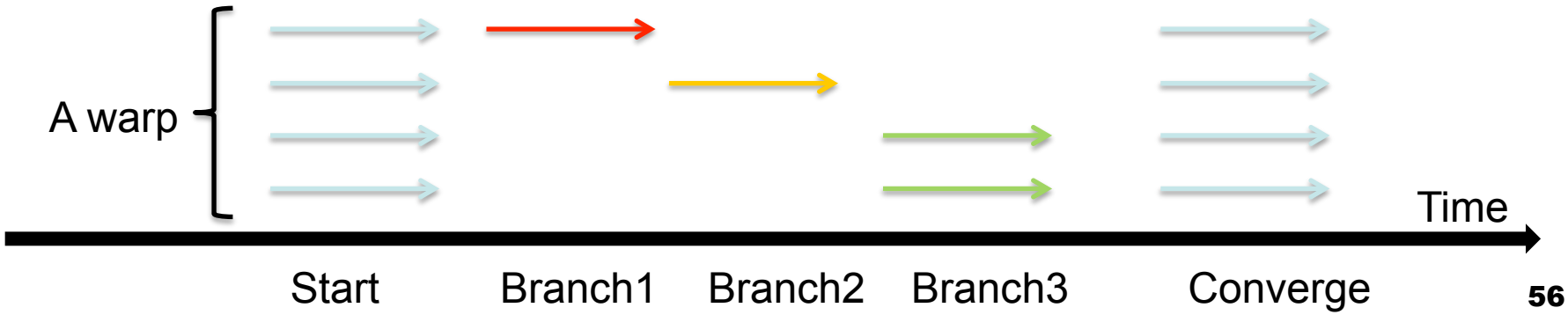
What is a CUDA warp?

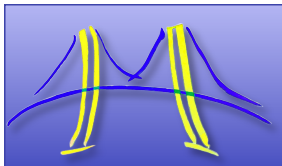
- A group of 32 CUDA threads that execute simultaneously
  - Execution hardware is most efficiently utilized when all threads in a warp execute instructions from the same PC.
  - Identifiable uniquely by dividing the Thread Index by 32
  - If threads in a warp ***diverge*** (execute different PCs), then some execution pipelines go unused
  - If threads in a warp access aligned, contiguous blocks of DRAM, the accesses are ***coalesced*** into a single high-bandwidth access
- The minimum granularity of efficient SIMD execution, and the maximum hardware SIMD width in a CUDA processor



Single Instruction Multiple Data

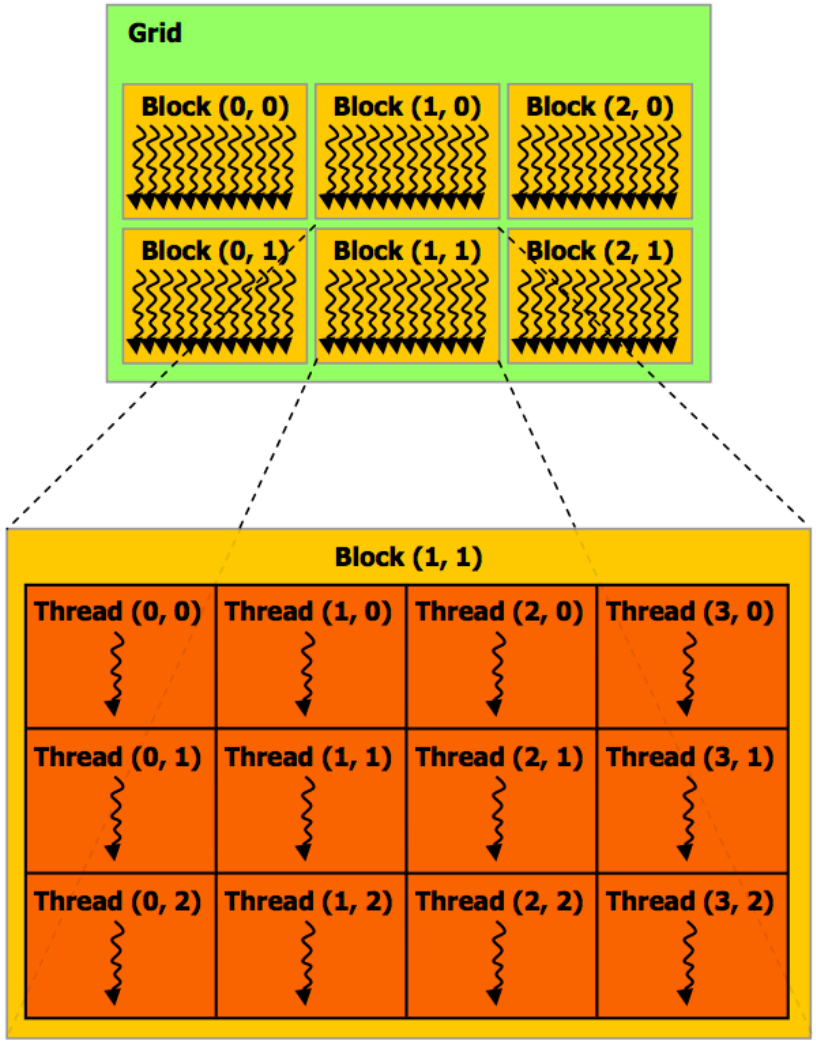
- Individual threads of a warp start together at the same program address
- Each thread has its own instruction address counter and register state
  - Each thread is free to branch and execute independently
  - Provide the MIMD abstraction
- Branch behavior
  - Each branch will be executed serially
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

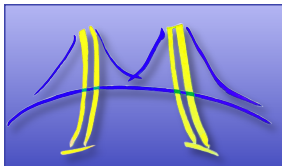




What is a CUDA thread block?

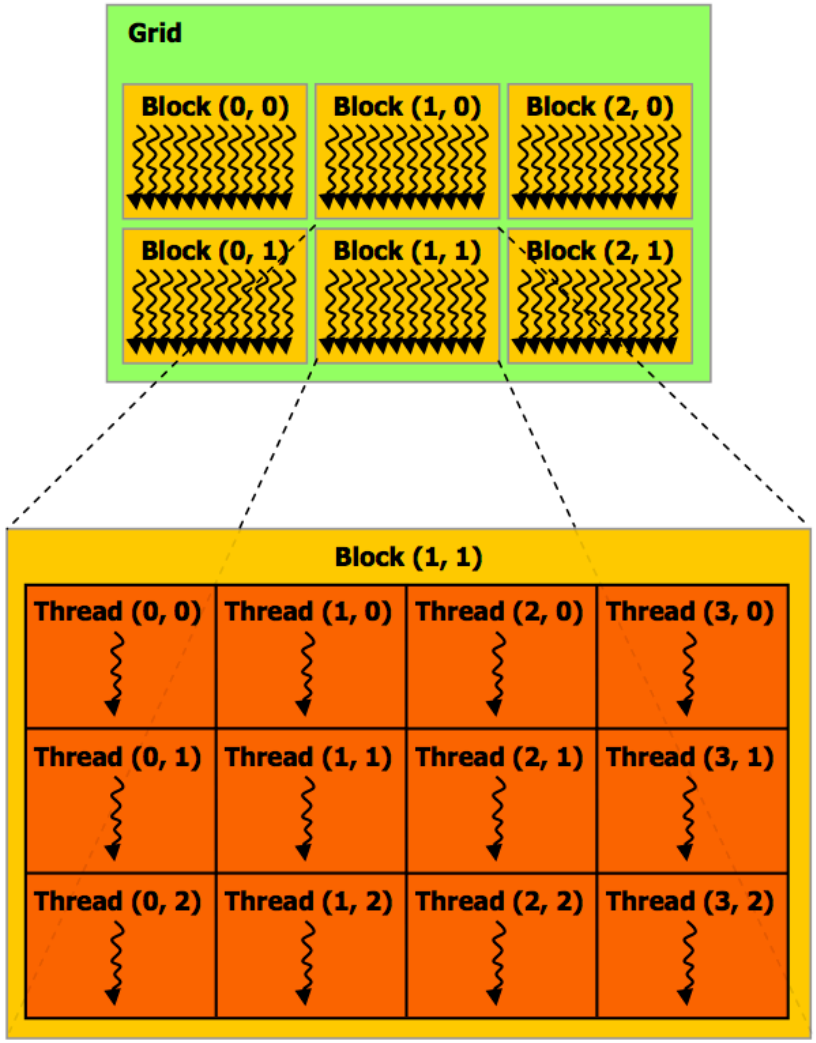
- A thread block is a **virtualized multi-threaded core**
  - Configured at kernel-launch to have a number of scalar processors, registers, `__shared__` memory
  - Consists of a number (32-1024) of CUDA threads, who all share the integer identifier `blockIdx.{x,y,z}`
- ... executing a **data parallel task** of moderate granularity
  - The cacheable working-set should fit into the register file and the L1 cache
  - All threads in a block share a (small) instruction cache and synchronize via the barrier intrinsic `__syncthreads()`

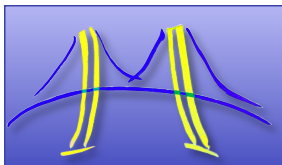




What is a CUDA grid?

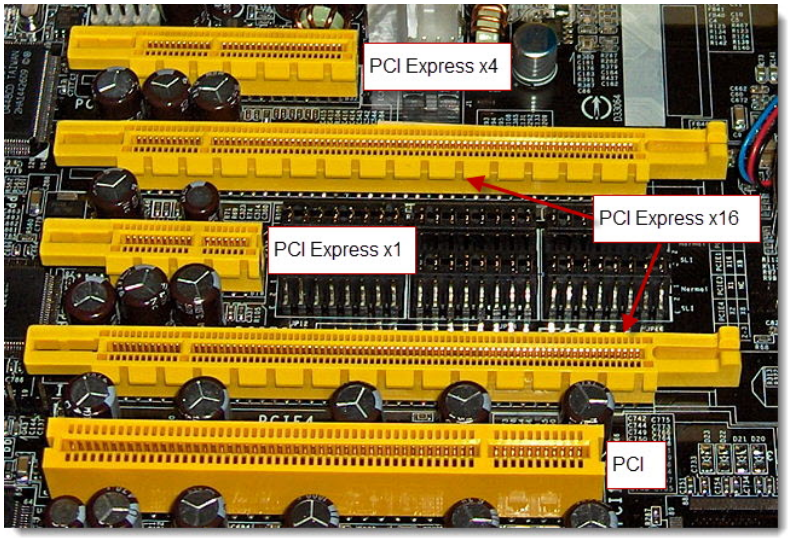
- A set of Thread Blocks performing related computations
  - All threads in a single kernel call have the same entry point and function arguments, initially differing only in `blockIdx.{x,y,z}`
  - Thread blocks in a grid may execute any code they want, e.g. `switch (blockIdx.x) { ... }` incurs no penalty
- There is an implicit global barrier between kernel calls
- Thread blocks of a kernel call must be **parallel sub-tasks**
  - Program must be valid for ***any interleaving*** of block executions
  - The flexibility of the memory system technically allows Thread Blocks to communicate and synchronize in arbitrary ways ...
  - But there is **no guarantee** that all Thread Blocks execute concurrently, and inter-block communication is risky!

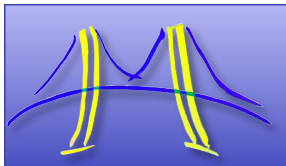




CUDA Host Runtime Support

- CUDA is a heterogeneous programming model
  - Sequential code runs in the “Host Thread” on a CPU core, and the “Device” code runs on the many cores of the GPU
  - The Host and the Device communicate via a PCI-Express link
  - The PCI-E link is slow (high latency, low bandwidth)
    - Desirable to minimize the amount of data transferred and the number of transfers





CUDA Host Runtime Support

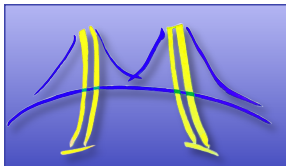
- Allocation/Deallocation of memory on the GPU:
  - **cudaMalloc(void**, int), cudaFree(void*)**
- Memory transfers to/from the GPU:
  - **cudaMemcpy(void*,void*,int, dir)**
  - **dir can be "cudaMemcpyHostToDevice" or "cudaMemcpyDeviceToHost"**

```
int main () {
    int N = (1024*1024);
 // pointers to array on the CPU
    float *h_a = new float[N];
    for(int i=0; i < N; i++) h_a[i] = i;
 // pointers to array on the GPU
    float *g_a;
    cudaMalloc(&g_a, sizeof(float)*N);
    cudaMemcpy(g_a, h_a, sizeof(float)*N,
        cudaMemcpyHostToDevice);
```

Create an array on the host CPU and fill with data

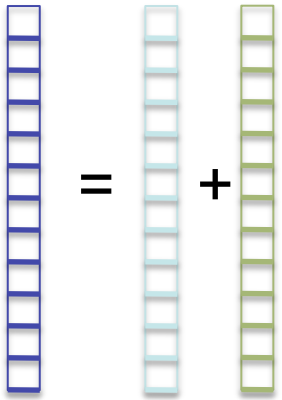
Allocate array on the GPU

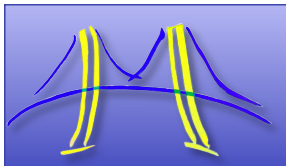
Copy data from host CPU upto the GPU



Hello World: Vector Addition (C++)

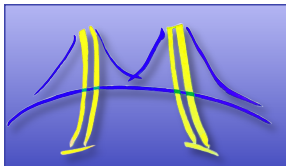
```
// Compute sum of length-N vectors: C = A + B
void
vecAdd (float* a, float* b, float* c, int N) {
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *d_a, *d_b, *d_c;
    d_a = new float[N];
  // ... allocate other arrays, fill with data


    vecAdd (d_a, d_b, d_c, N);
}
```





Hello World: Vector Addition (CUDA)

```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *d_a, *d_b, *d_c;
    cudaMalloc (&d_a,  sizeof(float) * N);
  // ... allocate other arrays, fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (d_a, d_b, d_c, N);
}
```



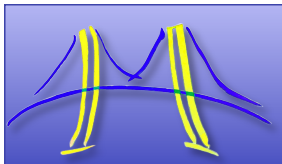
Vector addition: side by side

```
// Compute sum of length-N vectors: C = A + B
void
vecAdd (float* a, float* b, float* c, int N) {
    for (int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *d_a, *d_b, *d_c;
    d_a = new float[N];
  // ... allocate other arrays, fill with data


    vecAdd (d_a, d_b, d_c, N);
}
```

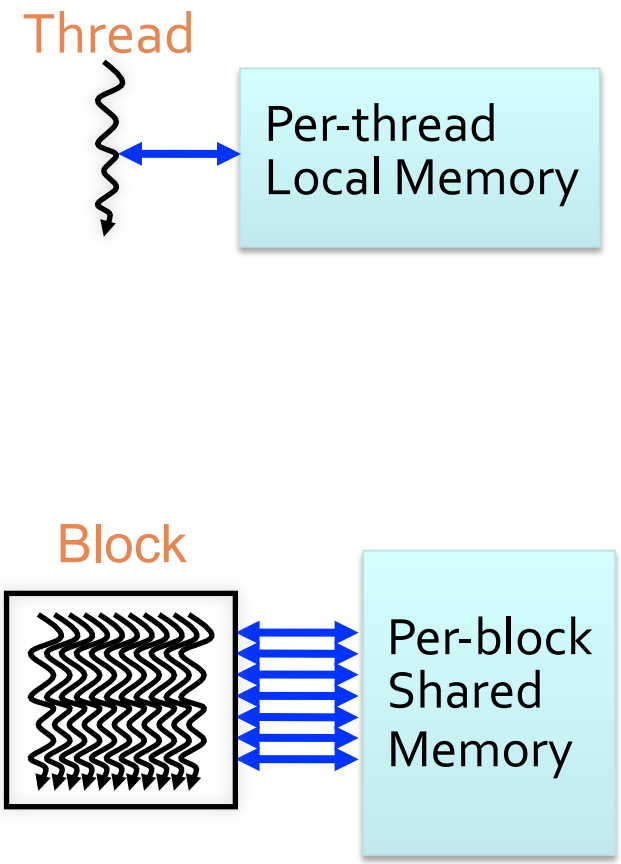
```
// Compute sum of length-N vectors: C = A + B
void __global__
vecAdd (float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x +
    threadIdx.x;
    if (i < N) c[i] = a[i] + b[i];
}

int main () {
    int N = ... ;
    float *d_a, *d_b, *d_c;
    cudaMalloc (&d_a,  sizeof(float) * N);
  // ... allocate other arrays, fill with data

  // Use thread blocks with 256 threads each
    vecAdd <<< (N+255)/256, 256 >>> (d_a, d_b,
    d_c, N);
}
```

CUDA Exercise 1

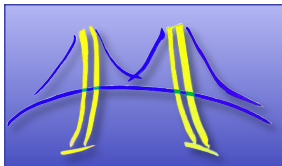
- Goal
  - Verify that you really understand the constructs by playing with the vector add program
- Problem
  - Start with the vector addition program we provide, create a CUDA version of the program.
- Extra work
  - Experiment with the different lengths of the vectors. How does performance depend on the length of the vectors?



CUDA memory hierarchy

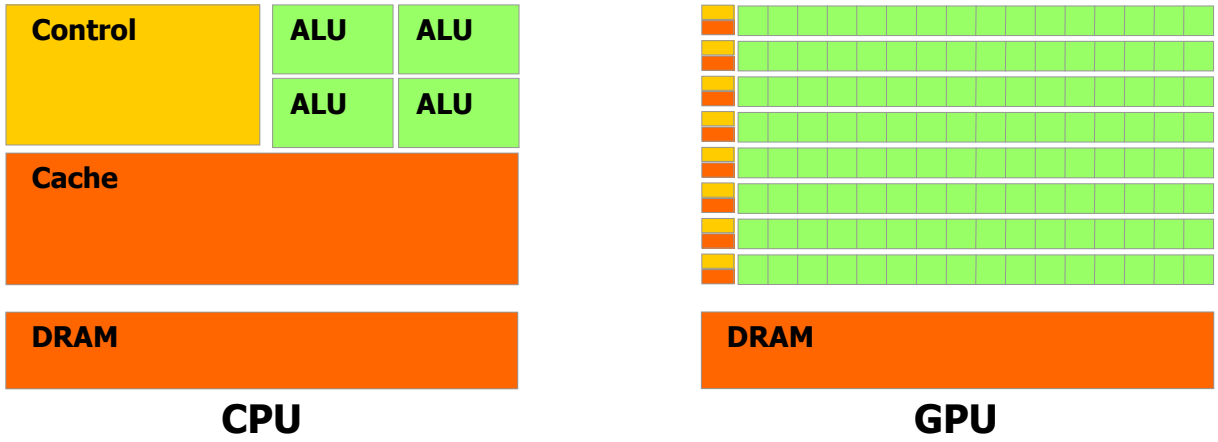


- Each CUDA thread has private access to a configurable number of registers
  - The 64 KB SM register file is partitioned among all resident threads
  - The CUDA program can trade degree of thread block concurrency for amount of per-thread state
  - Registers, stack spill into “local” DRAM if necessary
- Each thread block has private access to a configurable amount of scratchpad memory
  - Pre-Fermi SM’s have 16 KB scratchpad only
  - The available scratchpad space is partitioned among resident thread blocks, providing another concurrency-state tradeoff

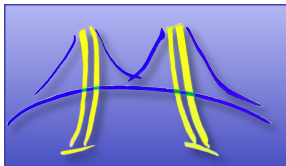


Memory, Memory, Memory

- A many core processor ≡ A device for turning a compute bound problem into a memory bound problem



- Lots of processors, only one socket
- Memory concerns dominate performance tuning

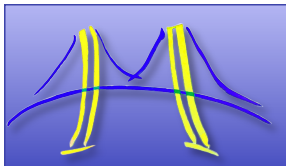


Thread-Block Synchronization

- Intra-block barrier instruction **`__syncthreads()`** for synchronizing accesses to **`__shared__`** memory
  - To guarantee correctness threads must **`__syncthreads()`** before reading values written by other threads
  - All threads in a block must execute the same **`__syncthreads()`** or the GPU will hang

```
extern __shared__ float T[];
__device__ void
transpose (float* a, int lda){
    int i = threadIdx.x, j = threadIdx.y;
    T[i + lda*j] = a[i + lda*j];
    __syncthreads();
    a[i + lda*j] = T[j + lda*i];
}
```

"extern __shared__" allows the shared memory block to dynamically sized at run-time



Using per-block shared memory

- The per-block shared memory / L1 cache is a crucial resource: without it, the performance of most CUDA programs would be hopelessly DRAM-bound
- Block-shared variables can be declared statically:

```
__shared__ int begin, end;
```

- Software-managed scratchpad memory is allocated statically:

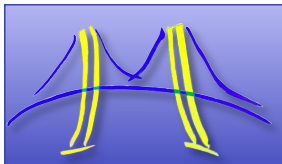
```
__shared__ int scratch[128];
scratch[threadIdx.x] = ... ;
```

- ... or dynamically:

```
extern __shared__ int scratch[];

kernel_call <<< grid_dim, block_dim, scratch_size >>> ( ... );
```

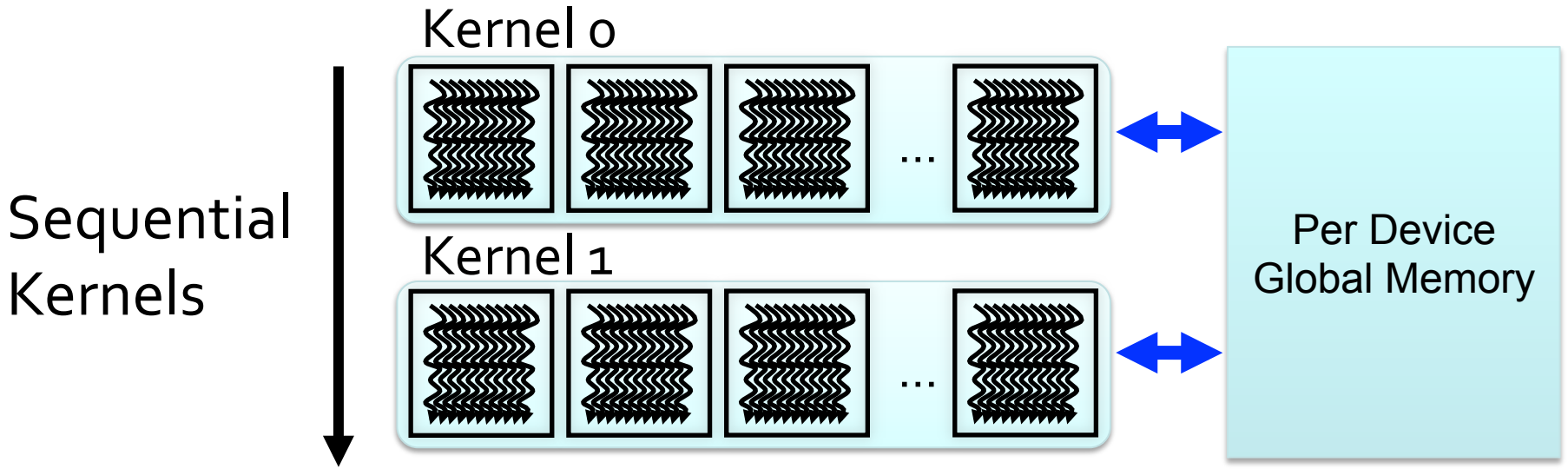
- Most intra-block communication is via shared scratchpad:

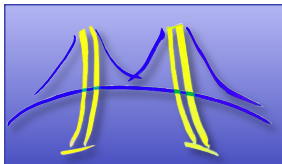
```
scratch[threadIdx.x] = ...;
__syncthreads();
int left = scratch[threadIdx.x - 1];
```



CUDA Memory Hierarchy

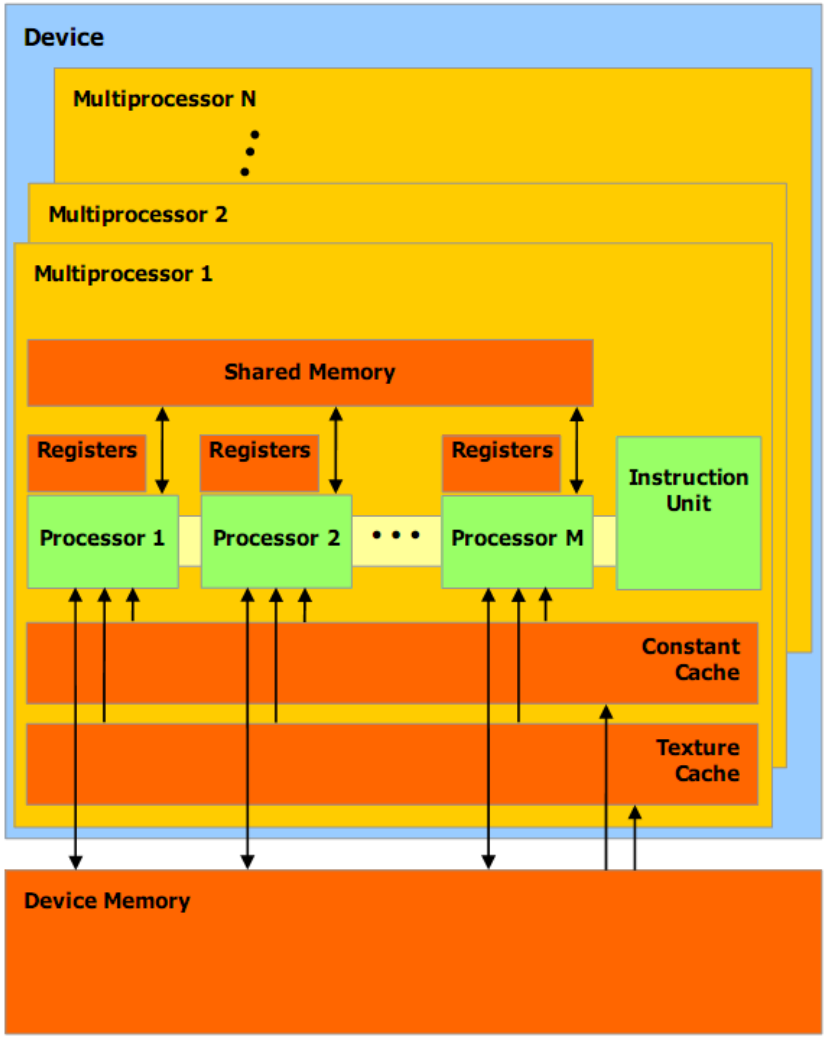
- Thread blocks in all Grids share access to a large pool of "Global" memory, separate from the Host CPU's memory.
  - Global memory holds the application's persistent state, while the thread-local and block-local memories are ephemeral
  - Global memory is much more expensive than local memories: O(100)x latency, O(1/50)x (aggregate) bandwidth
  - Registers and Cache multiply bandwidth, massive multithreading hides latency

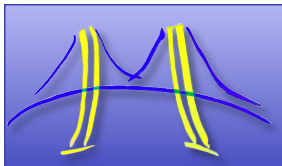




CUDA memory hierarchy

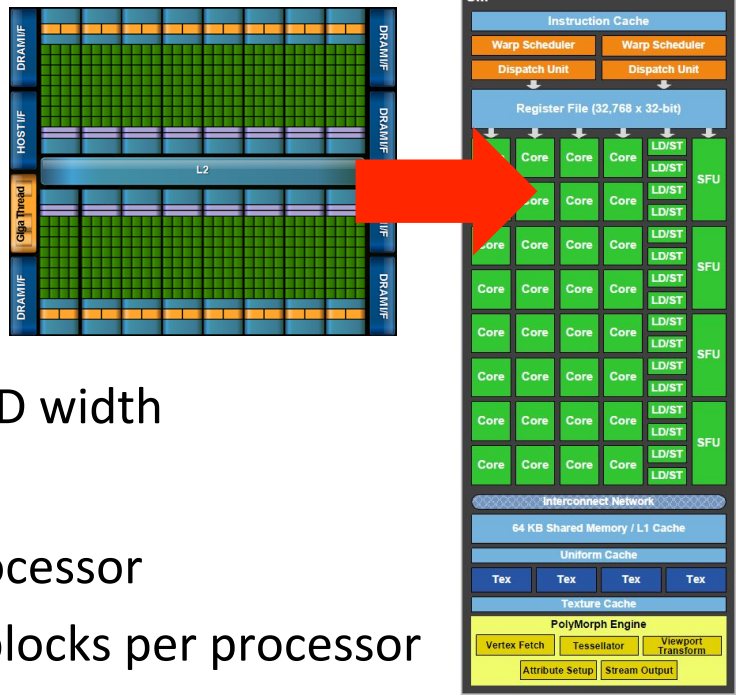
- There are other read-only components of the Memory Hierarchy that exist due to the graphics heritage of CUDA
- The 64 KB CUDA **Constant Memory** resides in the same address space DRAM as global memory, but is accessed via special read-only 8 KB per-SM caches
- The CUDA **Texture Memory** also resides in DRAM's address space and is accessed via small per-SM read-only caches, but also includes interpolation hardware
  - This hardware is crucial for graphics performance, but only occasionally is useful for general-purpose workloads
- The behaviors of these caches are highly optimized for their roles in graphics workloads.





Mapping CUDA to Nvidia GPUs

- CUDA is designed to be functionally forgiving
  - First priority: make things work. Second: get performance.
- However, to get good performance, one must understand how CUDA is mapped to Nvidia GPUs
- Threads:
  - each thread is a SIMD vector lane
- Warps:
  - A SIMD instruction acts on a “warp”
  - Warp width is 32 elements: ***LOGICAL*** SIMD width
- Thread blocks:
  - Each thread block is scheduled onto a processor
  - Peak efficiency requires multiple thread blocks per processor



CUDA Exercise 2

- Goal
  - Work with the CUDA memory hierarchy to optimize a matrix multiplication program.
- Problem
  - Start with the matrix multiplication program we provide to compute C= A*B in parallel
  - Parallelize with CUDA using the dot product for each element of C (I,j) as a CUDA-thread
  - Optimize performance by (1) putting rows of the A matrix in thread-local memory and (2) putting rows of the B matrix in thread-block shared memory.