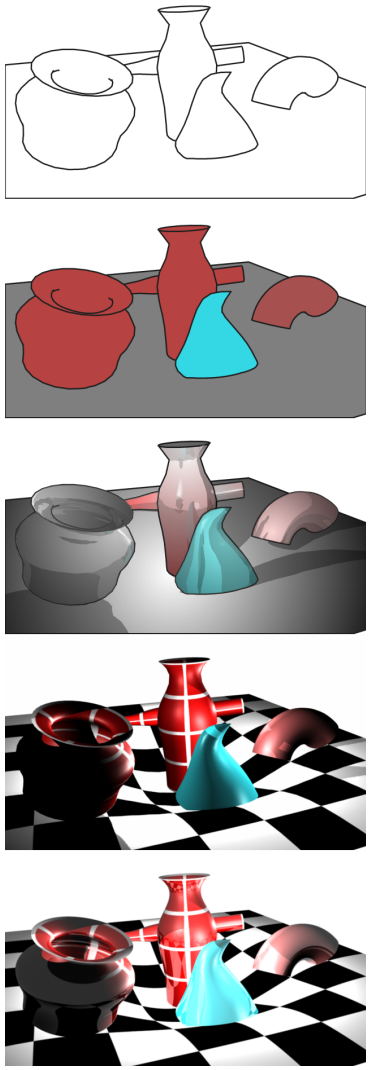
What is GPU Computing?

Stéphane Ethier

CPPG seminar

Friday October 19, 2012

First things first: What is a GPU?



- GPU stands for “Graphics Processing Unit”
- It is a specialized hardware optimized for the display/ rendering of complex 3D graphics (think “video games”) , the so-called “graphics pipeline”
  - Vertex operations
  - Rasterization
  - Fragment operations
  - Composition
- The technology evolved from “fixed-pipeline” to “programmable” (shader programs)
- To render a complex scene in real time from a 3D description in a file requires massive compute power

How much compute power are we talking about?

- NVIDIA GeForce GTX 690
- Based on latest Kepler technology
- 2X GK104 GPU chips
- **3072 CUDA cores**
- 384 GB/s memory bandwidth
- **4.58 Teraflops single precision performance**
- 0.19 Tflops double precision
- Frequency 1.019 GHz
- 4 GB GDDR5 memory

PPPL

Why not exploit this computing power for running scientific codes?

- That is exactly what several researchers tried to do starting in ~2000 by using graphics functions: the term “GPGPU” (General Purpose GPU computing) was coined.
- Only single precision though and required mapping the scientific problem in terms of operations done on triangles and polygons!
- In 2003, a team of researchers at Stanford U. developed a programming model for GPU that was based on extensions to the C language (http://graphics.stanford.edu/papers/brookgpu/brookgpu.pdf)
- Improving on that work, the NVIDIA company launched the CUDA language in 2006, along with CUDA-enabled GPU hardware

PPPL

GPGPU computing: marketing and reality

- Raw marketing numbers always quoted
  - > 4 Tflops peak floating point performance
  - Many papers claim > 100X speedup !!
- Looking more closely
  - Single or double precision? Same on both devices?
  - Sequential code vs. parallel code?
  - Standard operations or low-precision graphics constructs?
- The reality
  - GPUs are undoubtedly fast, but so are CPUs
  - CPU code not as carefully tuned as GPU version
  - Anything between 3 – 30X speedup is realistic

PPPL

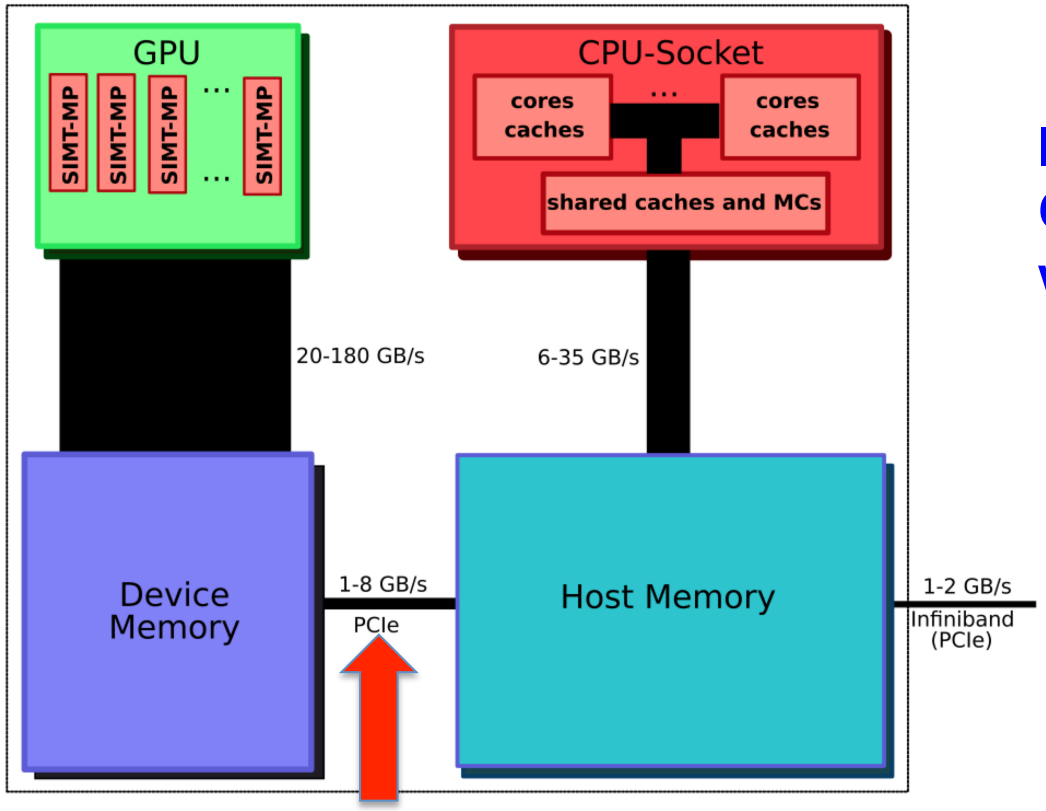
Side-by-side comparison of CPU vs. GPU

Processor	Intel Xeon E5-2690 "Sandy Bridge"	NVIDIA K20 Tesla GPU "Kepler" (Fermi)
No. of cores per "node"	16 cores (2x 8/chip)	**2,880** (512)
Frequency	2.9 GHz	~1 GHz
Memory bandwidth	51.2 GB/sec	**~320 GB/s (177)**
Peak Flops	371 GFlops	**~2000 Gflops (665)**
Memory (shared)	32 – 64 GB	**6 GB**

- The Kepler K20 will be the GPU in the Titan system at ORNL
- The Cray Cascade system "Edison" at NERSC will have the latest version of the Intel Sandy Bridge processor

PPPL

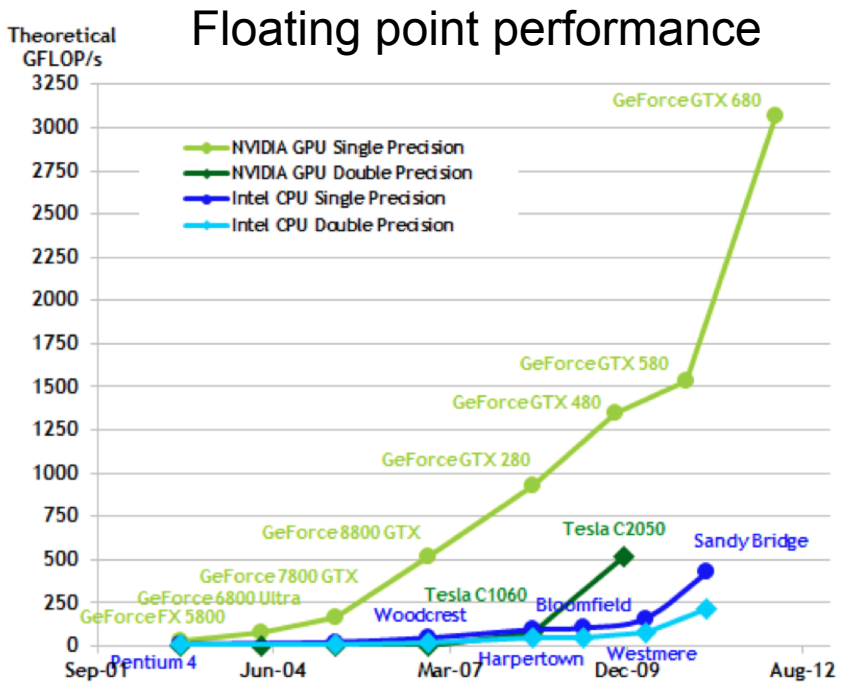
Current model: a CPU is required to drive the GPU

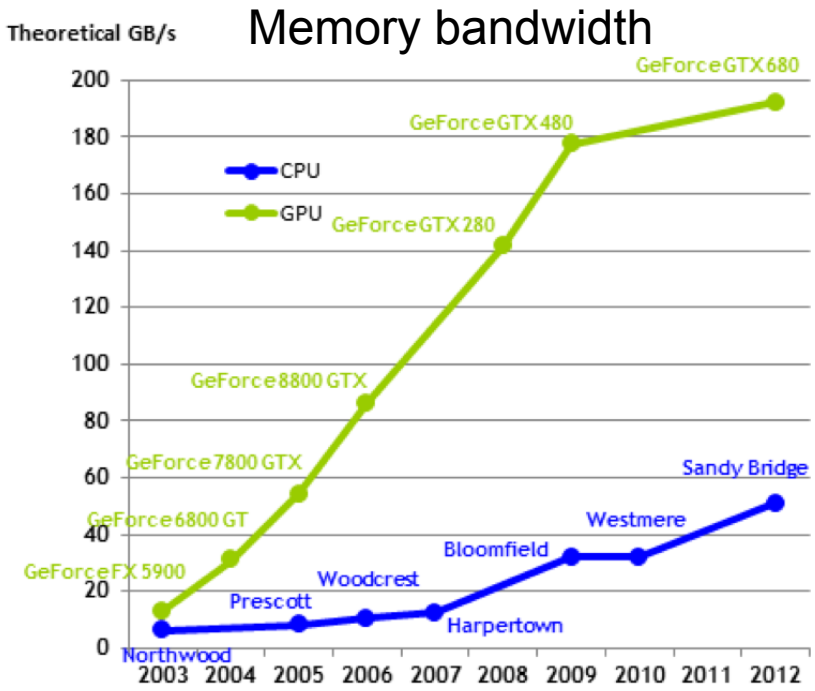


Ideally, both GPU and CPU should be working concurrently

Bottleneck in many cases

The standard performance plots comparing GPU and CPU





*Plots from CUDA C Programming Guide Version 4.2

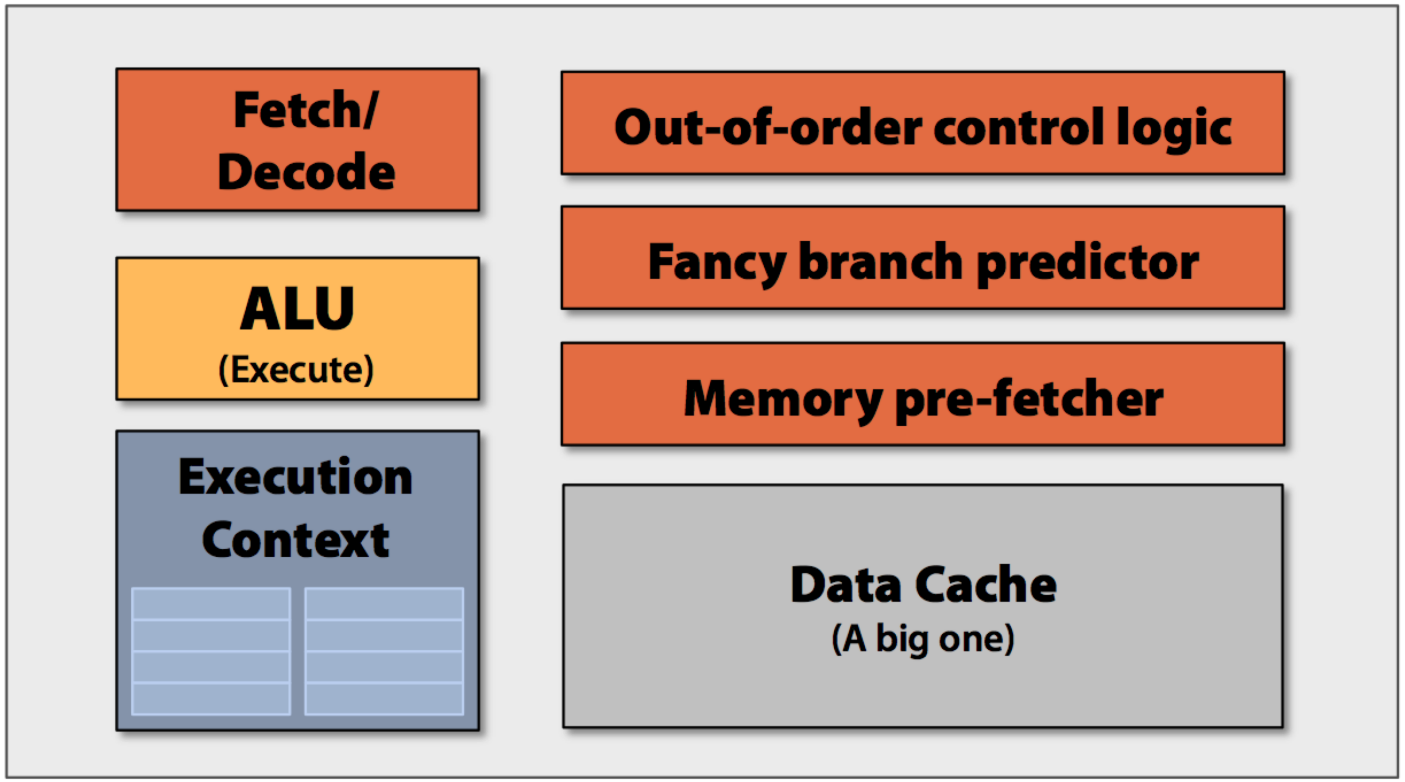
PPPL

What is the difference between a GPU and a CPU?

- How different is the GPU in terms of hardware architecture?
  - NVIDIA has been introducing more “scientific-code-friendly” hardware in its GPUs, such as more and faster double precision units, memory error correction (ECC), IEEE standard operations, etc.
  - The introduction of the “TESLA” model of GPU by NVIDIA has taken the hardware beyond the “gaming” business
- What makes it achieve such high performance compared to CPU?
- Why do we need CPUs at all then?!!?

PPPL

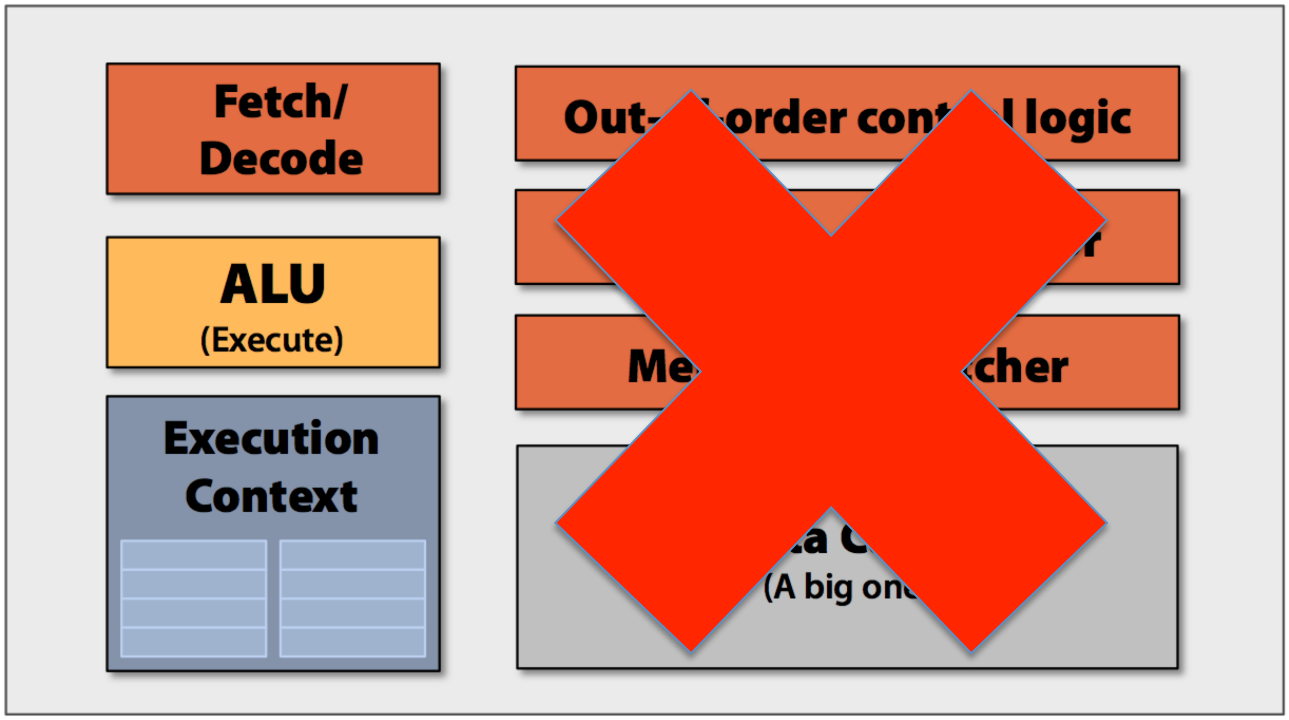
Start with a “CPU-style” core…



http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf

PPPL

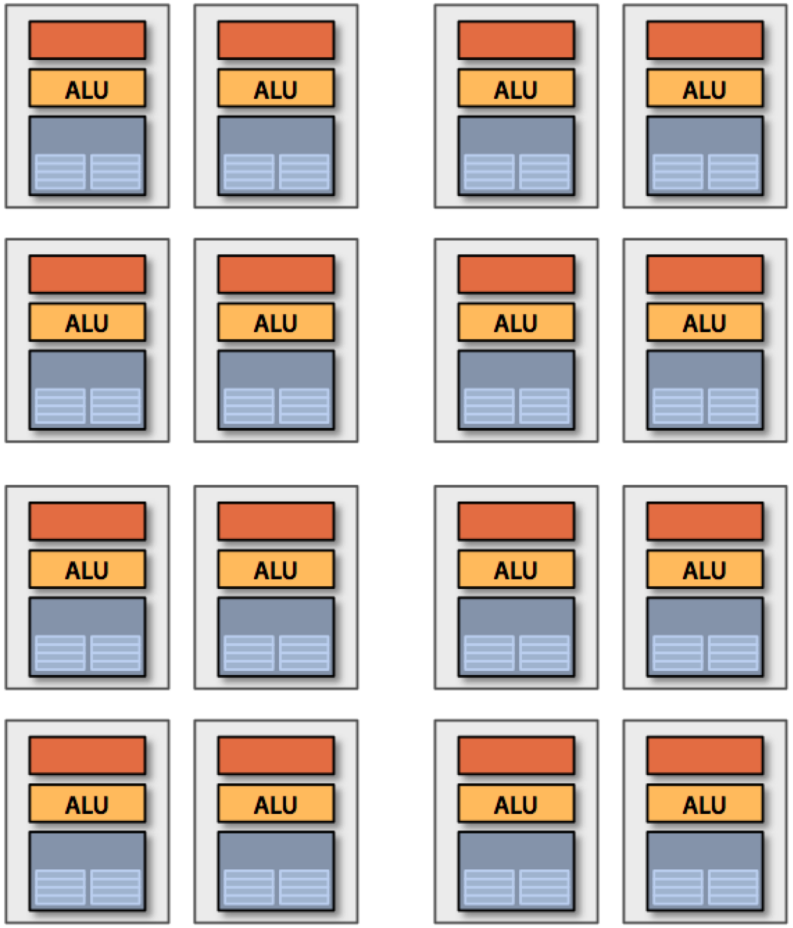
Remove everything that makes a single instruction stream go fast



http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf

PPPL

Put many of these simple cores together



- Let's not forget the original purpose of the GPU though:
  - Same operations applied to a large number of vertices, pixels, polygons, …

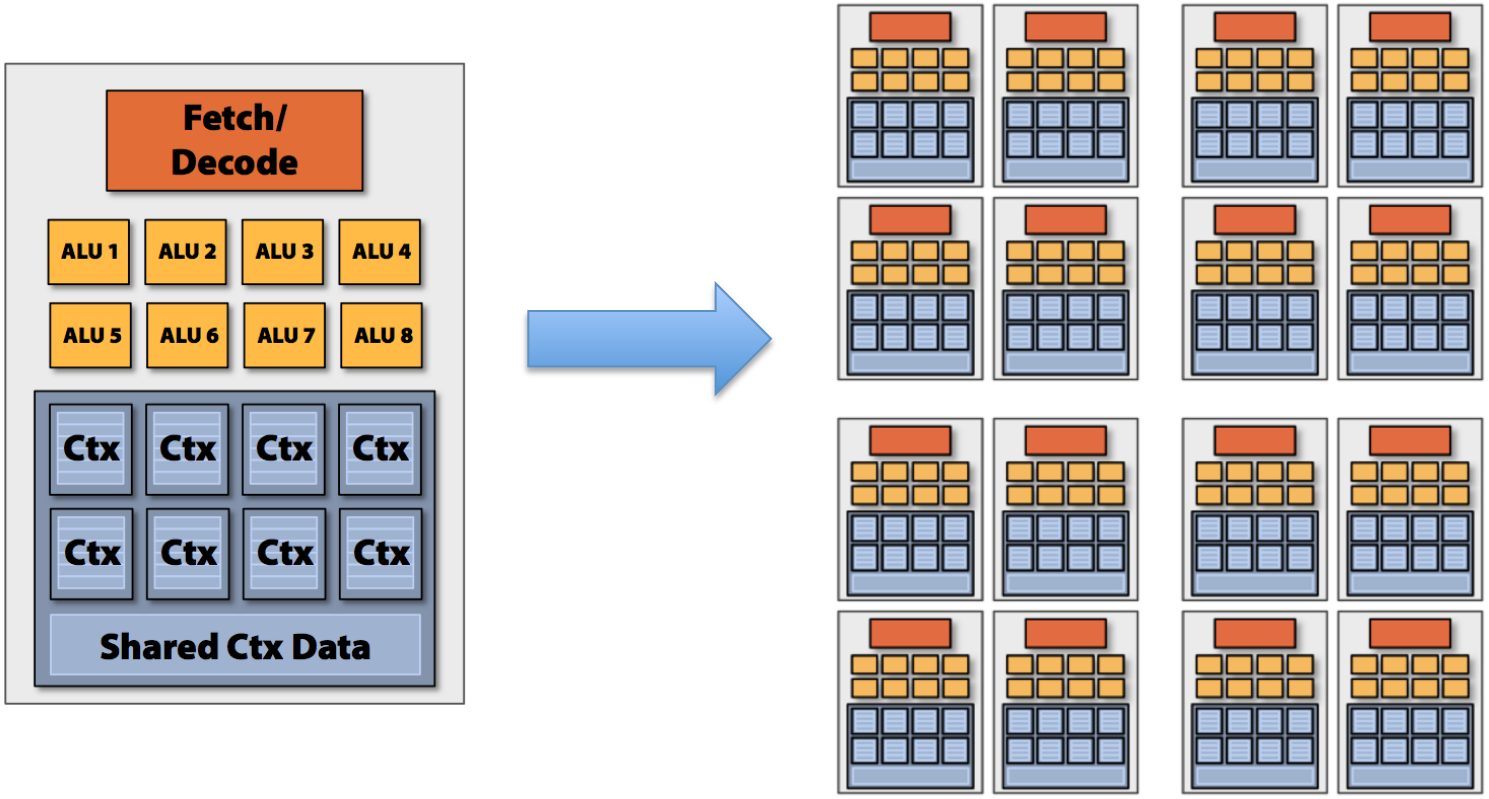
= DATA PARALLEL or SIMD (Single Instruction Multiple Data)

Better known in our community as **VECTOR PROCESSING**

http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf

PPPL

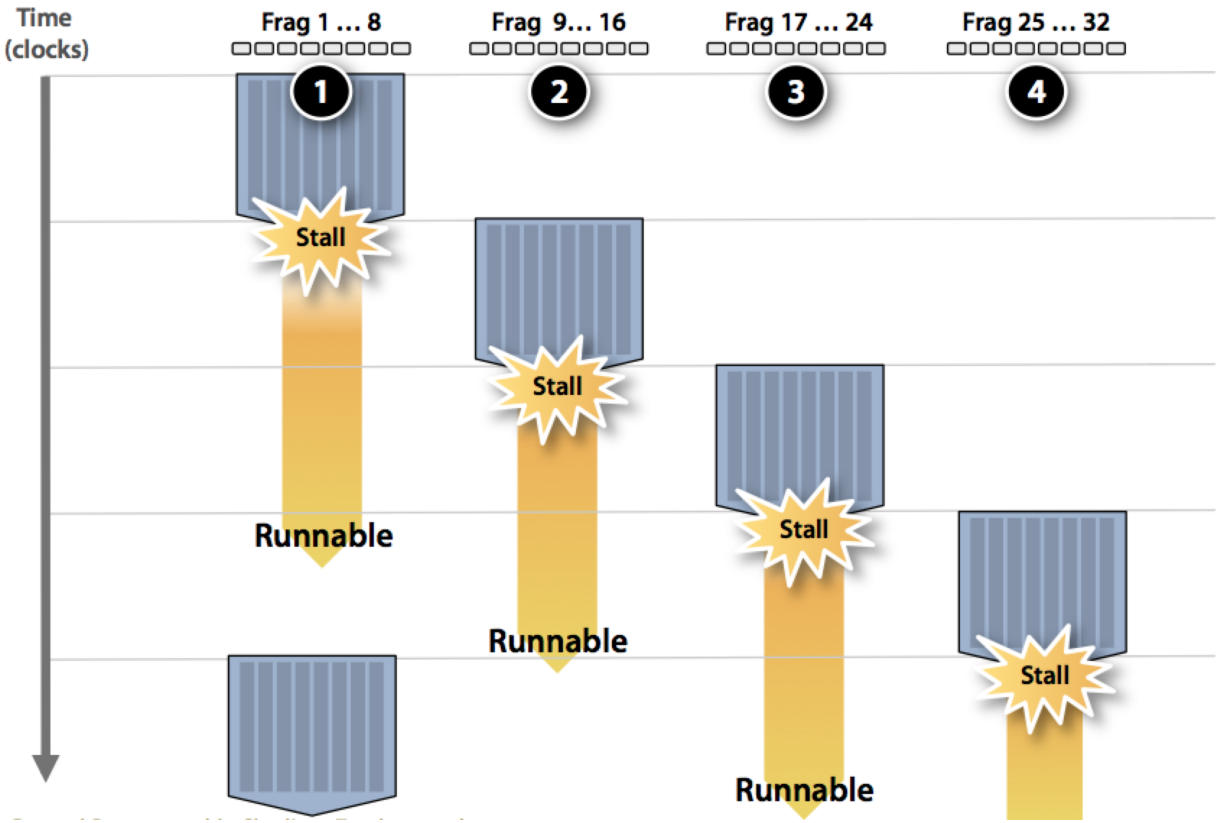
So let's add some arithmetic units and have them share a stream of instructions



http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf

PPPL

The secret to GPU high throughput: massive multi-threading + interleaving



NOT SIMD
But rather

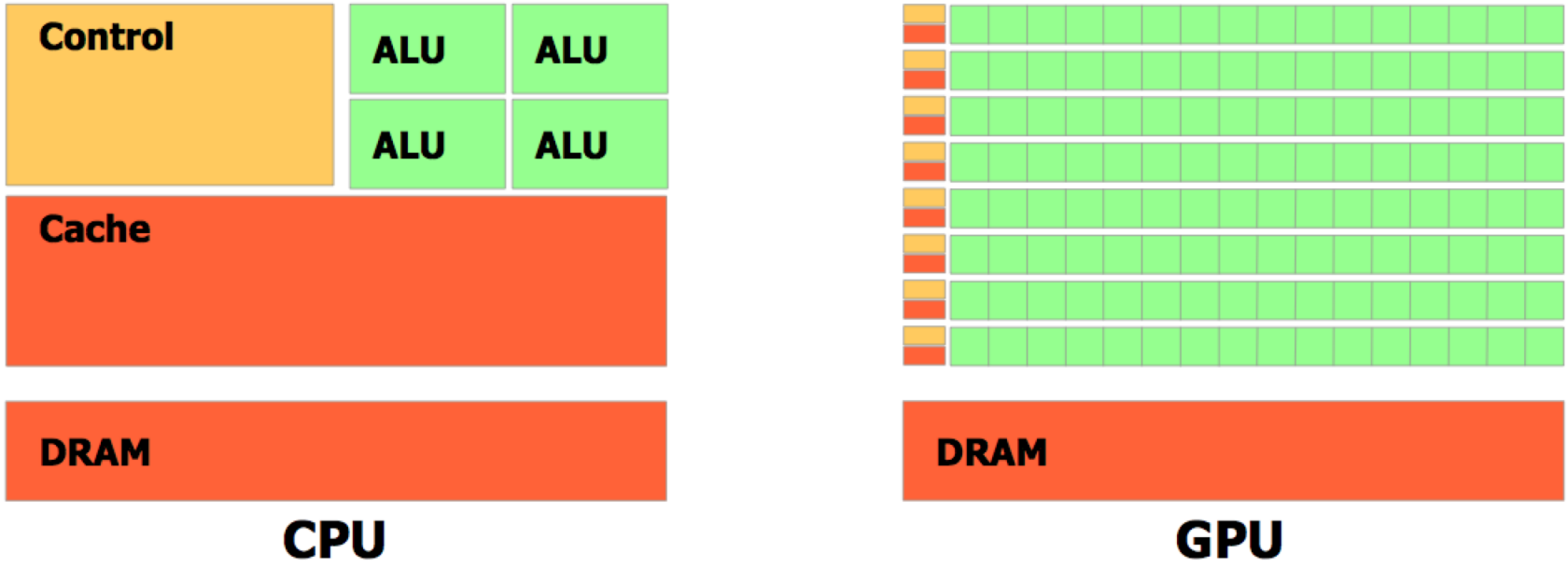
SIMT!
Single
Instruction
Multiple
Threads

255 registers per thread!!

http://s08.idav.ucdavis.edu/fatahalian-gpu-architecture.pdf

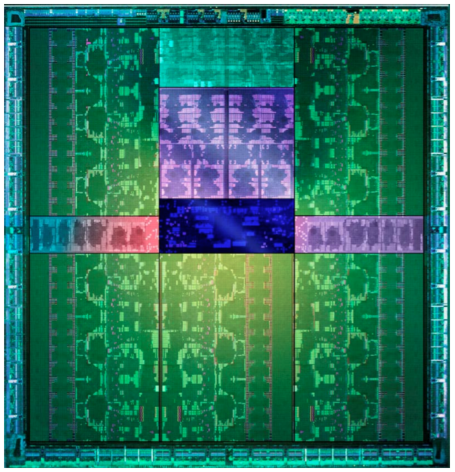
PPPL

Classic picture of CPU vs. GPU



Latest NVIDIA Kepler GPU

GK110 chip (15 SMX, 2048 threads/SMX, 7.1 billion transistors)



Using and programming GPUs

- Many possibilities
  - OpenGL: computer graphics functions used by game developers. NOT a good idea for scientific codes!!
  - CUDA: NVIDIA-specific programming language built as an extension of standard C language. Best approach to get the most out of your GPU. CUDA kernel not portable. Also available for FORTRAN but only for the PGI compiler.
  - OpenACC compiler directives similar to OpenMP. Portable code. Easy to get started. Available for a few compilers.
  - Libraries, commercial software, domain-specific environments, . . .
  - OpenCL: open standard, platform- and vendor independent
    - Works on both GPU AND CPU!!
    - Even harder than CUDA though…

PPPL

What to do first…

- MOST IMPORTANT:
  - Find and expose as much parallelism as you can in your code.
  - Try to remove as many dependencies as you can between successive iterations in a loop.
  - The ideal case is when each iteration is completely independent from the others ➔ VECTORIZATION

Try OpenACC directives first

- http://www.openacc-standard.org
- http://www.pgroup.com/doc/openACC_gs.pdf
- Least changes to your code
- It is portable across different platforms and compilers
- Not all compilers support Open ACC though
  - CRAY, PGI, and CAPS are the only ones at this point
- When running the same code on a multi-core CPU you can use OpenMP directives instead of OpenACC and run in parallel there too!
- Hides a lot of the complexity
- Works for Fortran, C, C++

PPPL

Example of OpenACC directive

It can be as simple as the following:

```
subroutine smooth( a, b, w0, w1, w2, n, m, niters )
 real, dimension(:,:) :: a,b
 real :: w0, w1, w2
 integer :: n, m, niters
 integer :: i, j, iter
  do iter = 1,niters
   !$acc kernels loop
   do i = 2,n-1
    do j = 2,m-1
     a(i,j)= w0 * b(i,j) + &
        w1 * (b(i-1,j) + b(i,j-1) + b(i+1,j) + b(i,j+1)) + &
        w2 * (b(i-1,j-1) + b(i-1,j+1) + b(i+1,j-1) + b(i+1,j+1))
    enddo
   enddo
```

OpenACC not giving good performance? move to CUDA

- CUDA C
  - http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf
- CUDA FORTRAN
  - http://www.pgroup.com/doc/pgicudaforug.pdf

PPPL

Some GPGPU references

- http://www.gputechconf.com/gtcnew/on-demand-gtc.php
- http://www.nvidia.com
- http://gpgpu.org
  - In particular: http://gpgpu.org/ppam2011
- http://www.olcf.ornl.gov/event/cray-technical-workshop-on-xk6-programming/
- http://www.pgroup.com/resources/index.htm
- http://www.caps-entreprise.com/products/openacc-compiler/

Conclusion

- No matter where you will run your code tomorrow, you need to exploit all the parallelism and think in terms of shared memory multi-threading
- This is valid for both CPU and GPU
- In the future, CPU and GPU will merge to become a highly power efficient, highly multi-threaded compute hardware

PPPL