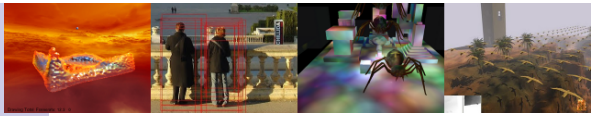


# GPU Architecture

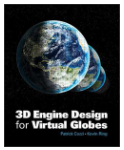
Patrick Cozzi
University of Pennsylvania
CIS 371 Guest Lecture
Spring 2012

## Who is this guy?

Analytical Graphics, Inc.

developer

lecturer

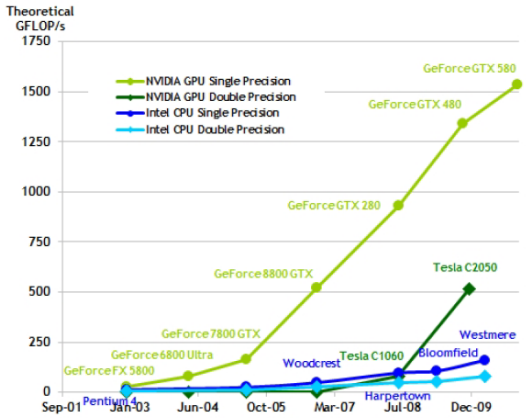



author

editor

See http://www.seas.upenn.edu/~pcozzi/

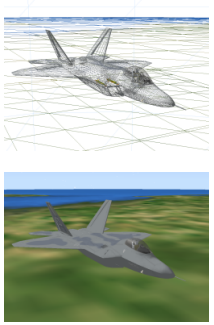
## How did this happen?




http://proteneer.com/blog/?p=263

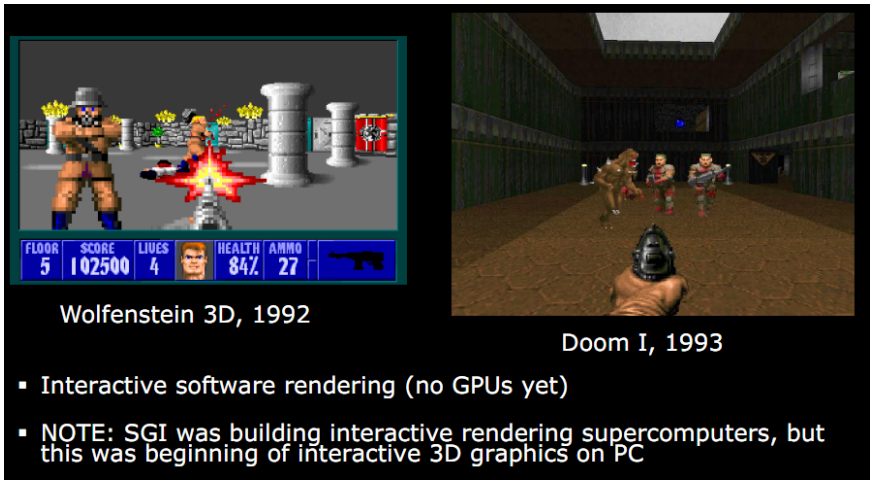
## Graphics Workloads

- Triangles/vertices and pixels/fragments



Right image from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch14.html

# Early 90s – Pre GPU




Slide from http://s09.idav.ucdavis.edu/talks/01-BPS-SIGGRAPH09-mhouston.pdf

# Why GPUs?

- Graphics workloads are embarrassingly parallel
  - Data-parallel
  - Pipeline-parallel
- CPU and GPU execute in parallel
- Hardware: texture filtering, rasterization, etc.

# Data Parallel

- *Beyond Graphics*
  - Cloth simulation
  - Particle system
  - Matrix multiply

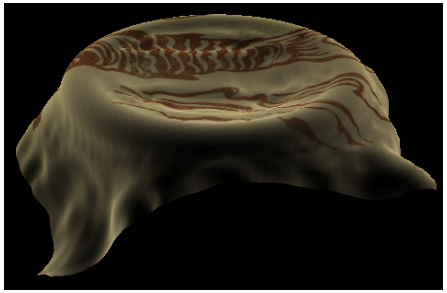


Image from: https://plus.google.com/u/0/photos/100838748547881402137/albums/5407605084626995217/5581900335460078306

# NVIDIA GeForce 6 (2004)

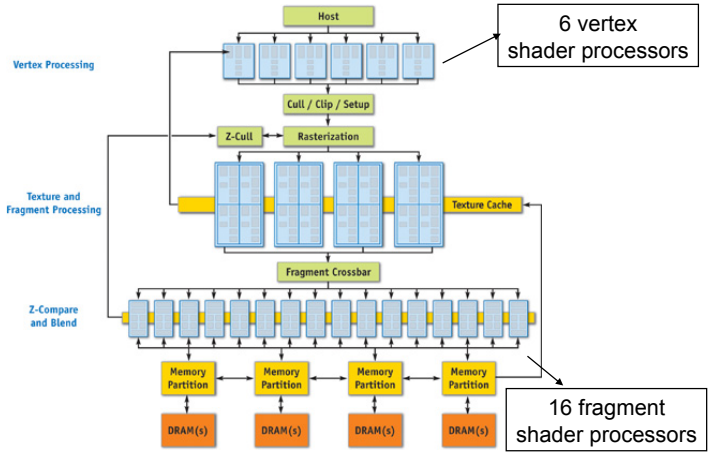
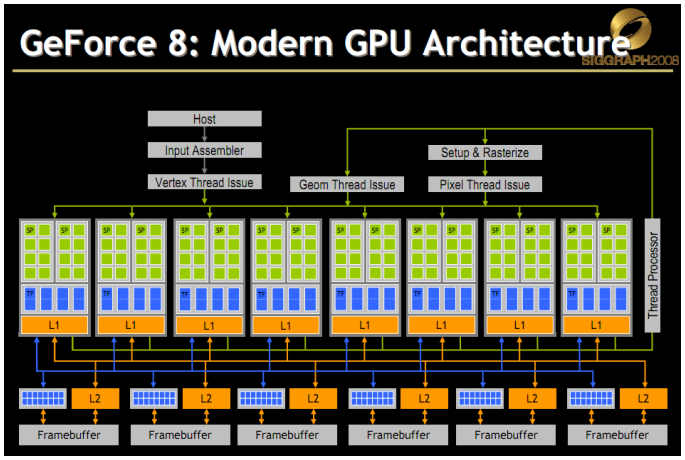


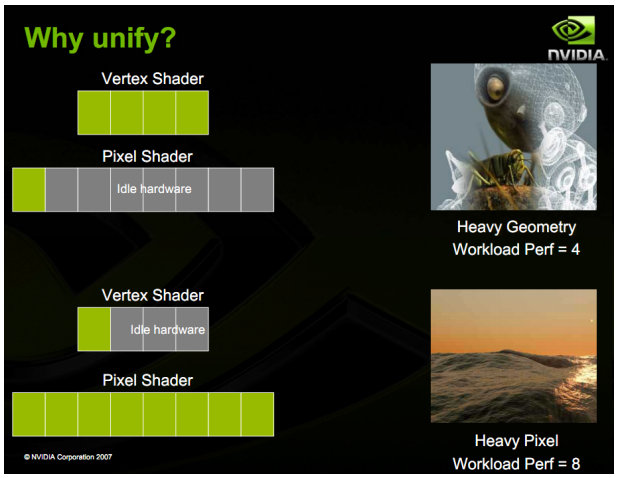

Image from http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter30.html

# NVIDIA G80 Architecture



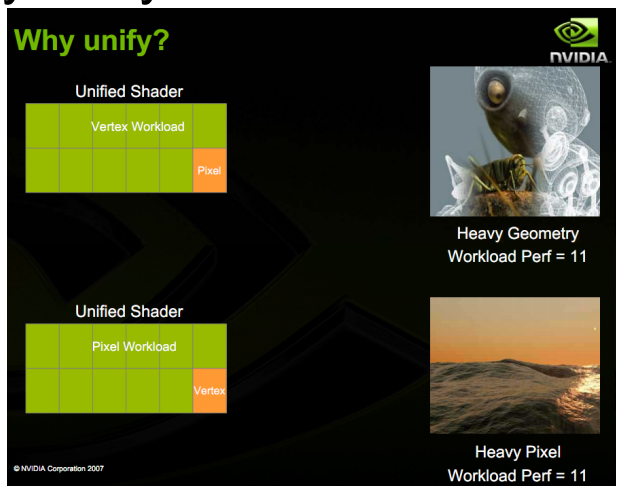

Slide from http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf

# Why Unify Shader Processors?




Slide from http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf

# Why Unify Shader Processors?

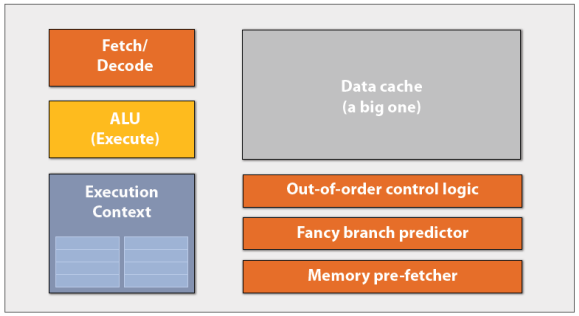



Slide from http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf

# GPU Architecture Big Ideas

- GPUs are specialized for
  - Compute-intensive, highly parallel computation
  - Graphics is just the beginning.
- Transistors are devoted to:
  - Processing
  - Not:
    - Data caching
    - Flow control

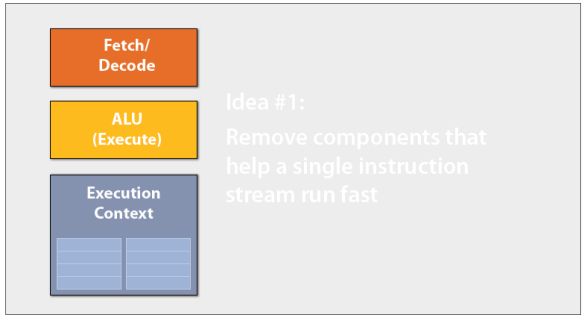
## "CPU-style" cores






Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

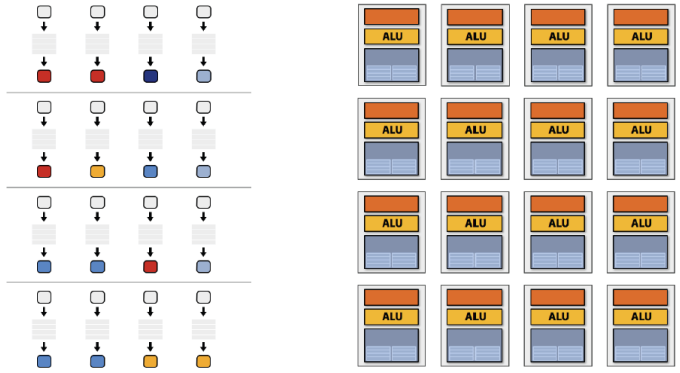
## Slimming down






Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

## Sixteen cores (sixteen fragments in parallel)

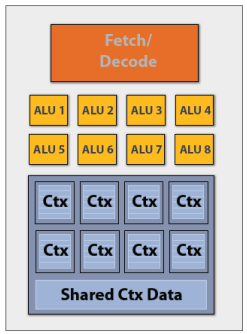



16 cores = 16 simultaneous instruction streams



Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

## Add ALUs



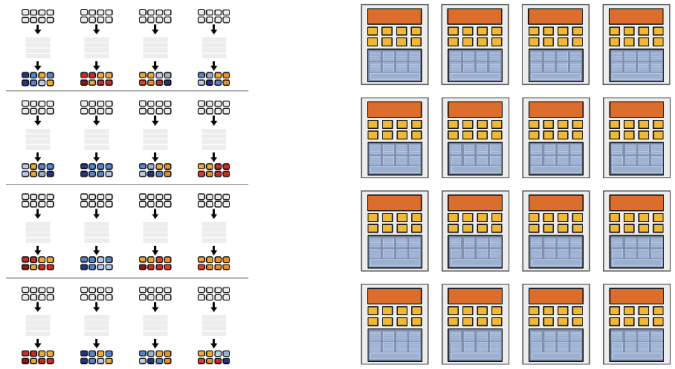

Idea #2:
Amortize cost/complexity of managing an instruction stream across many ALUs

**SIMD processing**



Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

# 128 fragments in parallel

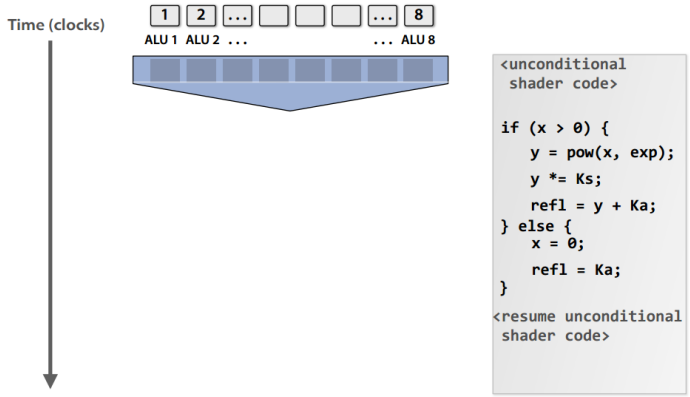


16 cores = 128 ALUs, 16 simultaneous instruction streams



Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

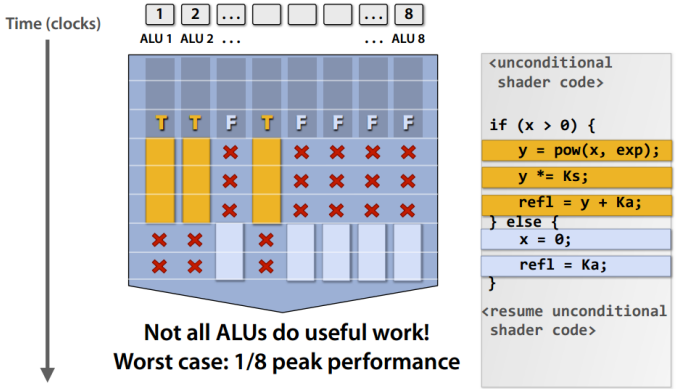
# But what about branches?






Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

# But what about branches?

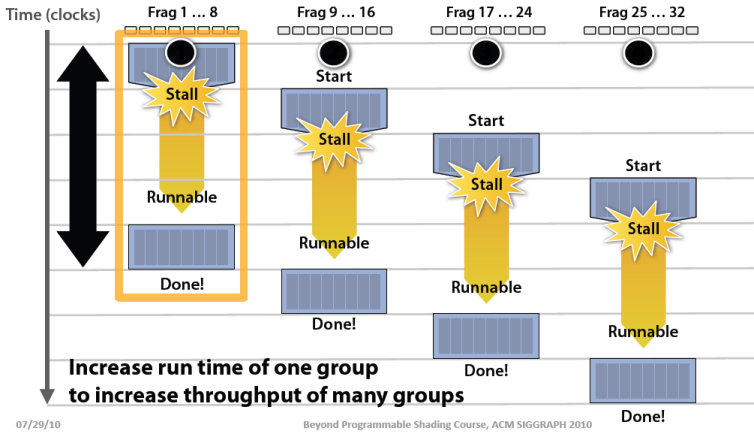



Not all ALUs do useful work!
Worst case: 1/8 peak performance



Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

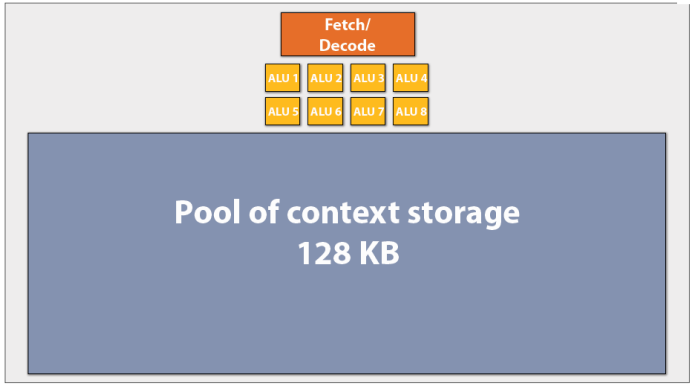
# Throughput!






Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

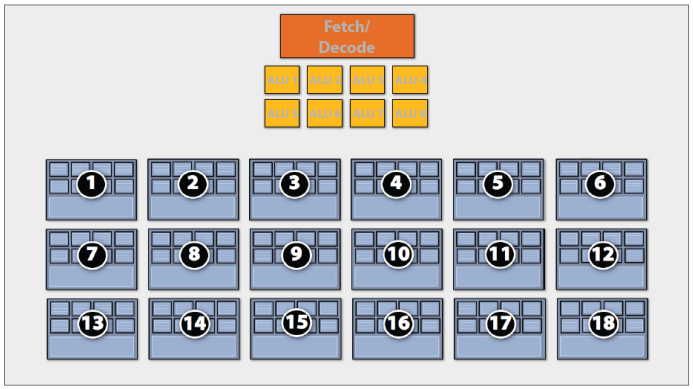
## Storing contexts






Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

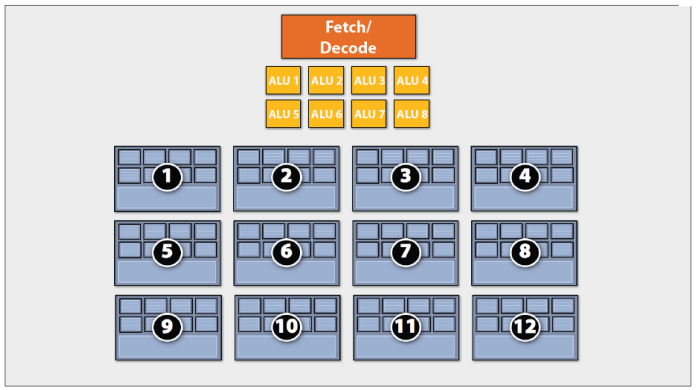
## Eighteen small contexts (maximal latency hiding)






Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

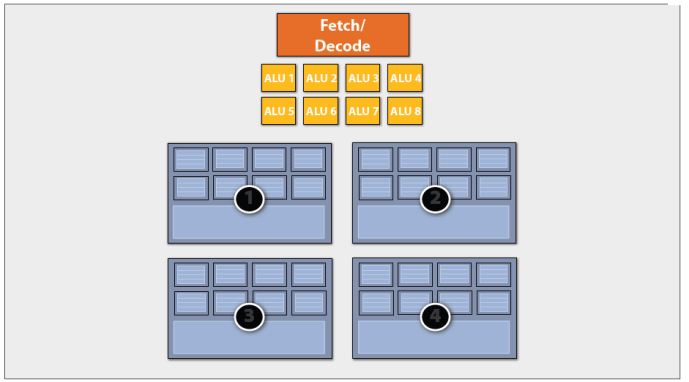
## Twelve medium contexts






Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

## Four large contexts (low latency hiding ability)






Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

## My chip!

16 cores

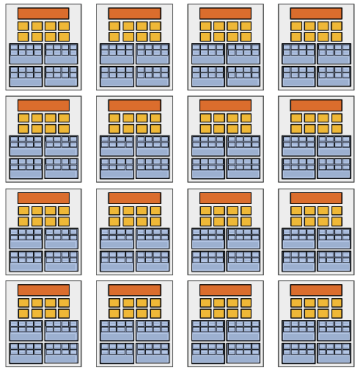
8 mul-add ALUs per core
(128 total)

16 simultaneous
instruction streams

64 concurrent (but interleaved)
instruction streams

512 concurrent fragments

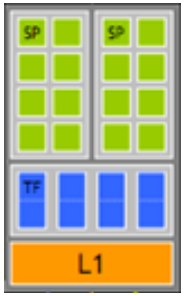
= 256 GFLOPs (@ 1GHz)



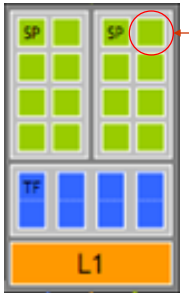


Slide from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

# NVIDIA G80

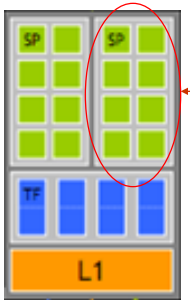



# NVIDIA G80




Streaming Processing (SP)

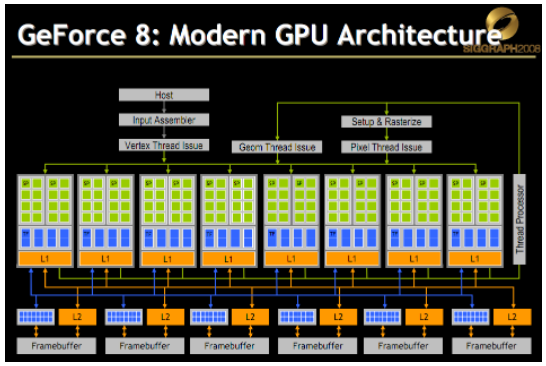
# NVIDIA G80




Streaming Multi-Processor (SM)

# NVIDIA G80

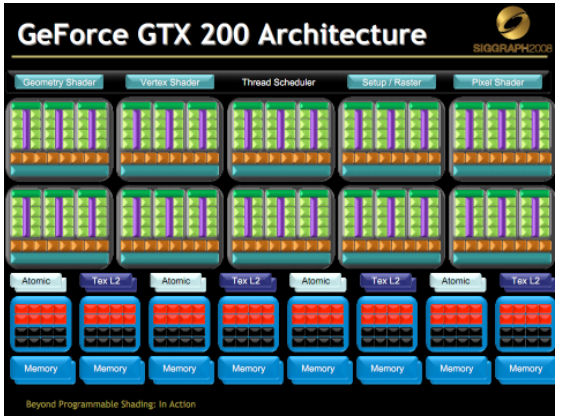
- 16 SMs
- Each with 8 SPs
  - 128 total SPs
- Each SM hosts up to 768 threads
- Up to 12,288 threads in flight




Slide from David Luebke: http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf

# NVIDIA GT200

- 30 SMs
- Each with 8 SPs
  - 240 total SPs
- Each SM hosts up to
  - 1024 threads
- In flight, up to
  - 30,720 threads




Slide from David Luebke: http://s08.idav.ucdavis.edu/luebke-nvidia-gpu-architecture.pdf

# Let's program this thing!

# GPU Computing History

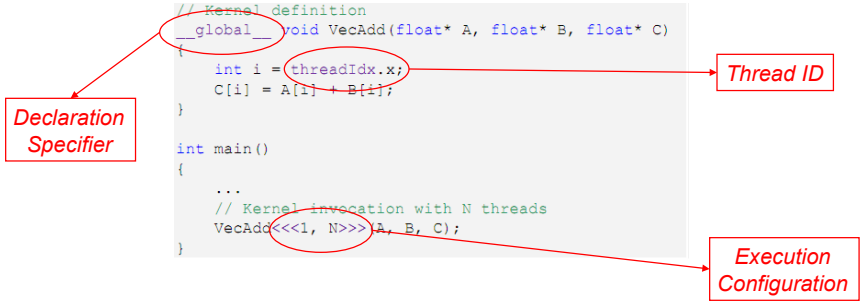
- 2001/2002 – researchers see GPU as data-parallel coprocessor
  - The *GPGPU* field is born
- 2007 – NVIDIA releases CUDA
  - *CUDA* – Compute Uniform Device Architecture
  - GPGPU shifts to *GPU Computing*
- 2008 – Khronos releases *OpenCL* specification

# CUDA Abstractions

- A hierarchy of thread groups
- Shared memories
- Barrier synchronization

# CUDA Kernels

- Executed N times in parallel by N different *CUDA threads*




# CUDA Program Execution

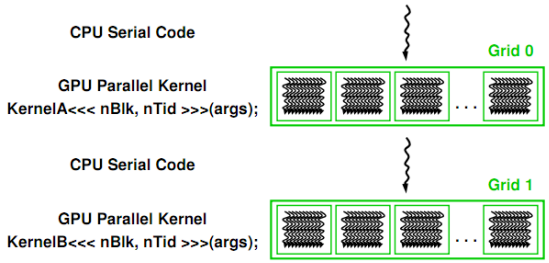



Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

# Thread Hierarchies

- *Grid* – one or more thread blocks
  - 1D or 2D
- *Block* – array of threads
  - 1D, 2D, or 3D
  - Each block in a grid has the same number of threads
  - Each thread in a block can
    - Synchronize
    - Access shared memory

# Thread Hierarchies

- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate

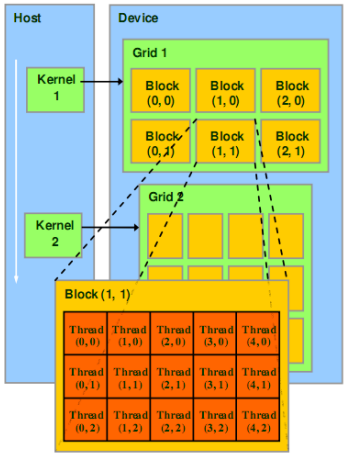



Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

# Thread Hierarchies

- Thread Block
  - Group of threads
    - G80 and GT200: Up to 512 threads
    - Fermi: Up to 1024 threads
  - Reside on same processor core
  - Share memory of that core

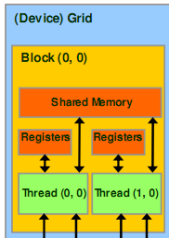



Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

# Thread Hierarchies

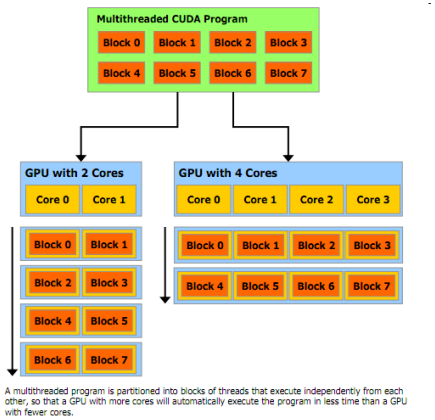



Figure 1-4. Automatic Scalability

Image from: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

# Thread Hierarchies

- Threads in a block
  - Share (limited) low-latency memory
  - Synchronize execution
    - To coordinate memory accesses
    - `__syncThreads()`
      - Barrier – threads in block wait until all threads reach this
      - Lightweight

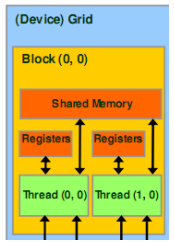



Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

# Scheduling Threads

- *Warp* – threads from a block
  - G80 / GT200 – *32* threads
  - Run on the same SM
  - Unit of thread scheduling
  - Consecutive `threadIdx` values
  - An implementation detail – in theory
    - `warpSize`

# Scheduling Threads

- Warps for three blocks scheduled on the same SM.

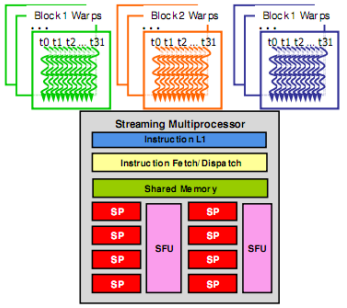



Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter3-CudaThreadingModel.pdf

# Scheduling Threads

Remember this:

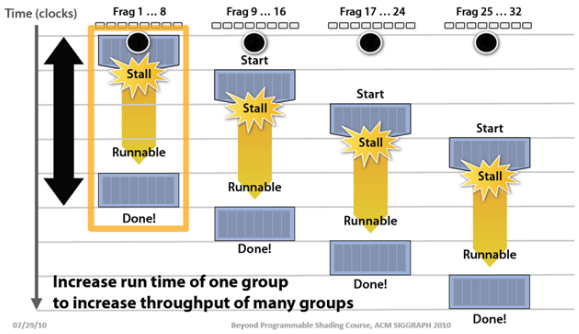
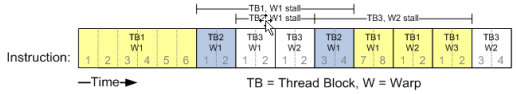



Image from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

# Scheduling Threads

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected






Slide from: http://courses.engr.illinois.edu/ece498/al/Syllabus.html

# Scheduling Threads

- What happens if branches in a warp diverge?

# Scheduling Threads

Remember this:

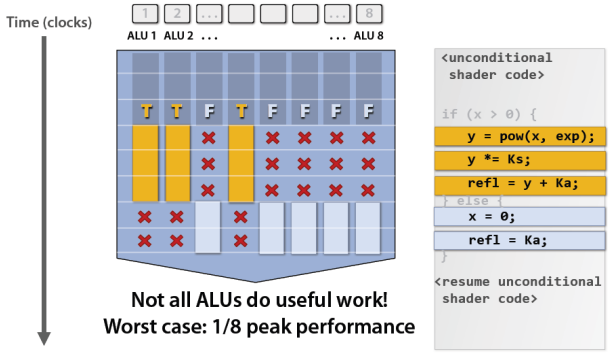
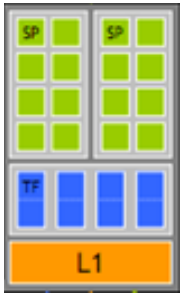



Image from: http://bps10.idav.ucdavis.edu/talks/03-fatahalian_gpuArchTeraflop_BPS_SIGGRAPH2010.pdf

# Scheduling Threads

- 32 threads per warp but 8 SPs per SM. What gives?




# Scheduling Threads

- 32 threads per warp but 8 SPs per SM. What gives?
- When an SM schedules a warp:
  - Its instruction is ready
  - 8 threads enter the SPs on the 1st cycle
  - 8 more on the 2nd, 3rd, and 4th cycles
  - Therefore, 4 cycles are required to dispatch a warp

# Scheduling Threads

- Question
  - A kernel has
    - `1` global memory read (`200` cycles)
    - `4` non-dependent multiples/adds
  - How many warps are required to hide the memory latency?

# Scheduling Threads

- Solution
  - Each warp has `4` multiples/adds
    - `16 cycles`
  - We need to cover `200` cycles
    - `200 / 16 = 12.5`
    - `ceil(12.5) = 13`
  - 13 warps are required

# Memory Model

Recall:

- Device code can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
- Host code can
  - R/W per grid global and constant memories

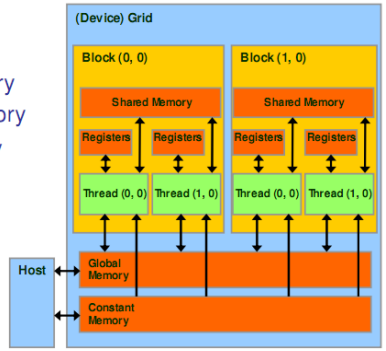


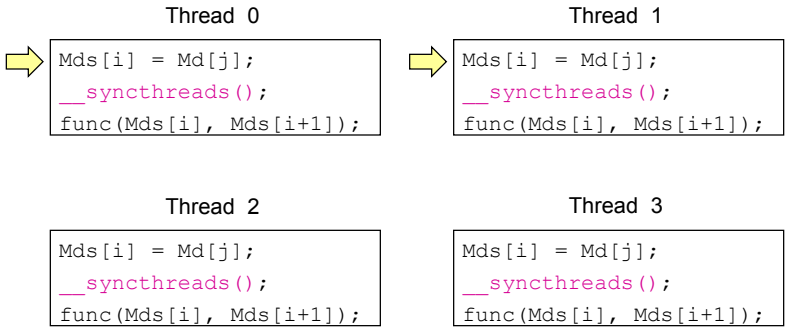

Image from: http://courses.engr.illinois.edu/ece498/al/textbook/Chapter2-CudaProgrammingModel.pdf

# Thread Synchronization

- Threads in a block can synchronize
  - call `__syncthreads` to create a barrier
  - A thread waits at this call until all threads in the block reach it, then all threads continue

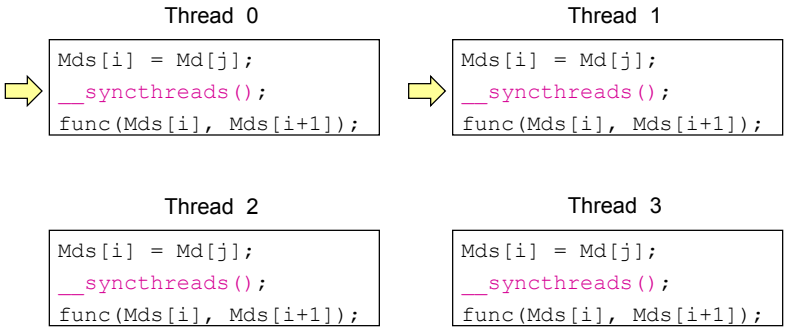
```
Mds[i] = Md[j];
__syncthreads();
func(Mds[i], Mds[i + 1]);
```

# Thread Synchronization



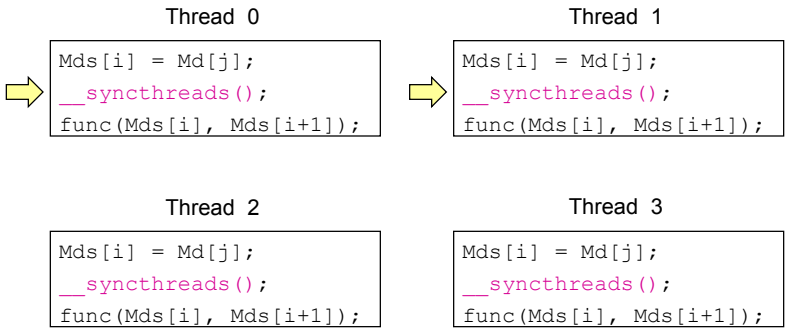

Time: 0

# Thread Synchronization




Time: 1

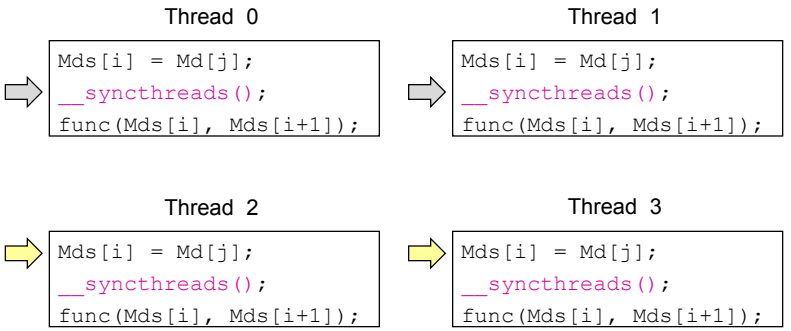
# Thread Synchronization




Threads 0 and 1 are blocked at barrier

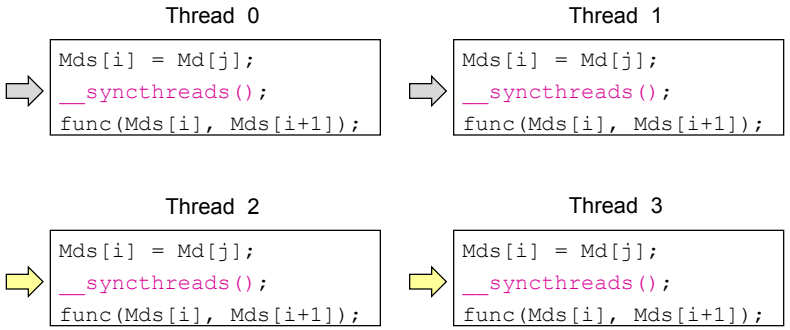
Time: 1

# Thread Synchronization



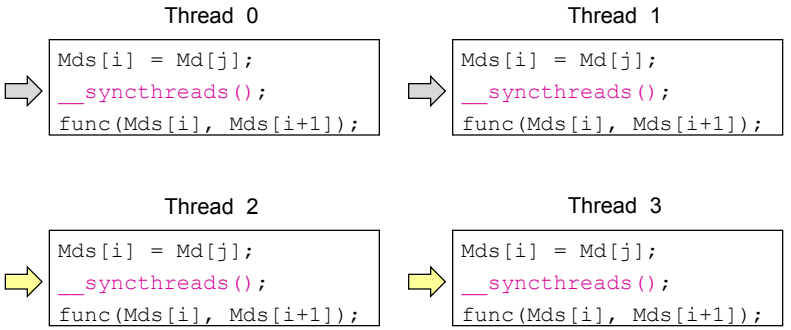

Time: 2

# Thread Synchronization




Time: 3

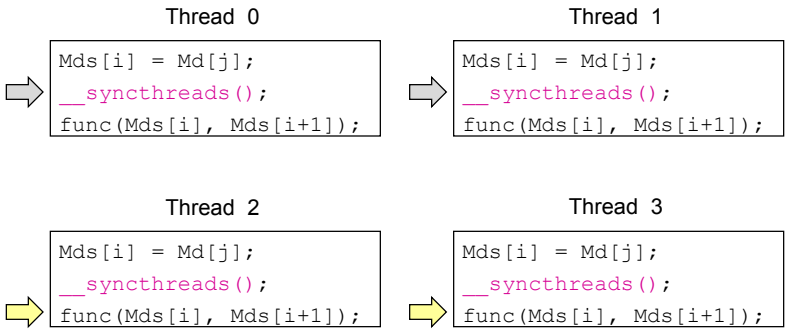
# Thread Synchronization




All threads in block have reached barrier, any thread can continue

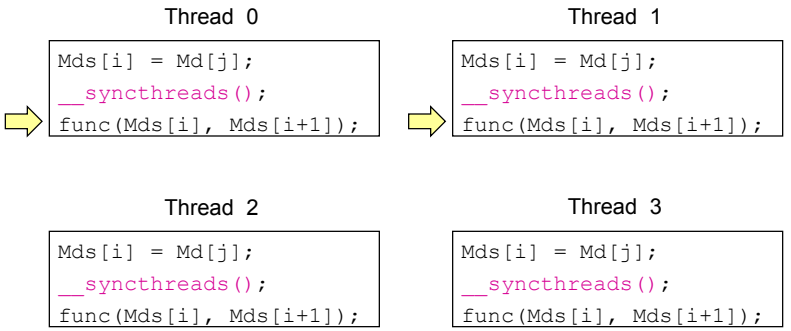
Time: 3

# Thread Synchronization




Time: 4

# Thread Synchronization




Time: 5

# Thread Synchronization

- Why is it important that execution time be similar among threads?
- Why does it only synchronize within a block?

# Thread Synchronization

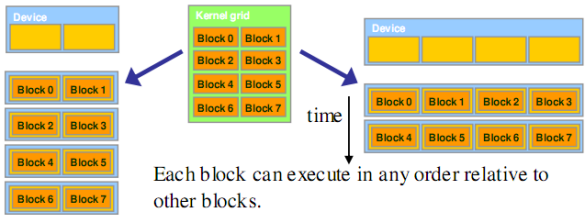



Figure 3.5 Lack of synchronization across blocks enables transparent scalability of CUDA programs

Image from http://courses.engr.illinois.edu/ece498/al/textbook/Chapter3-CudaThreadingModel.pdf

# Thread Synchronization

- Can `__syncthreads()` cause a thread to hang?

# Thread Synchronization

```
if (someFunc())
{
   __syncthreads();
}
// ...
```

# Thread Synchronization

```
if (someFunc())
{
   __syncthreads();
}
else
{
  __syncthreads();
}
```