

General-Purpose Computation on Graphics Hardware

Sebastian Schätz

Seminar at University of Applied Sciences Regensburg

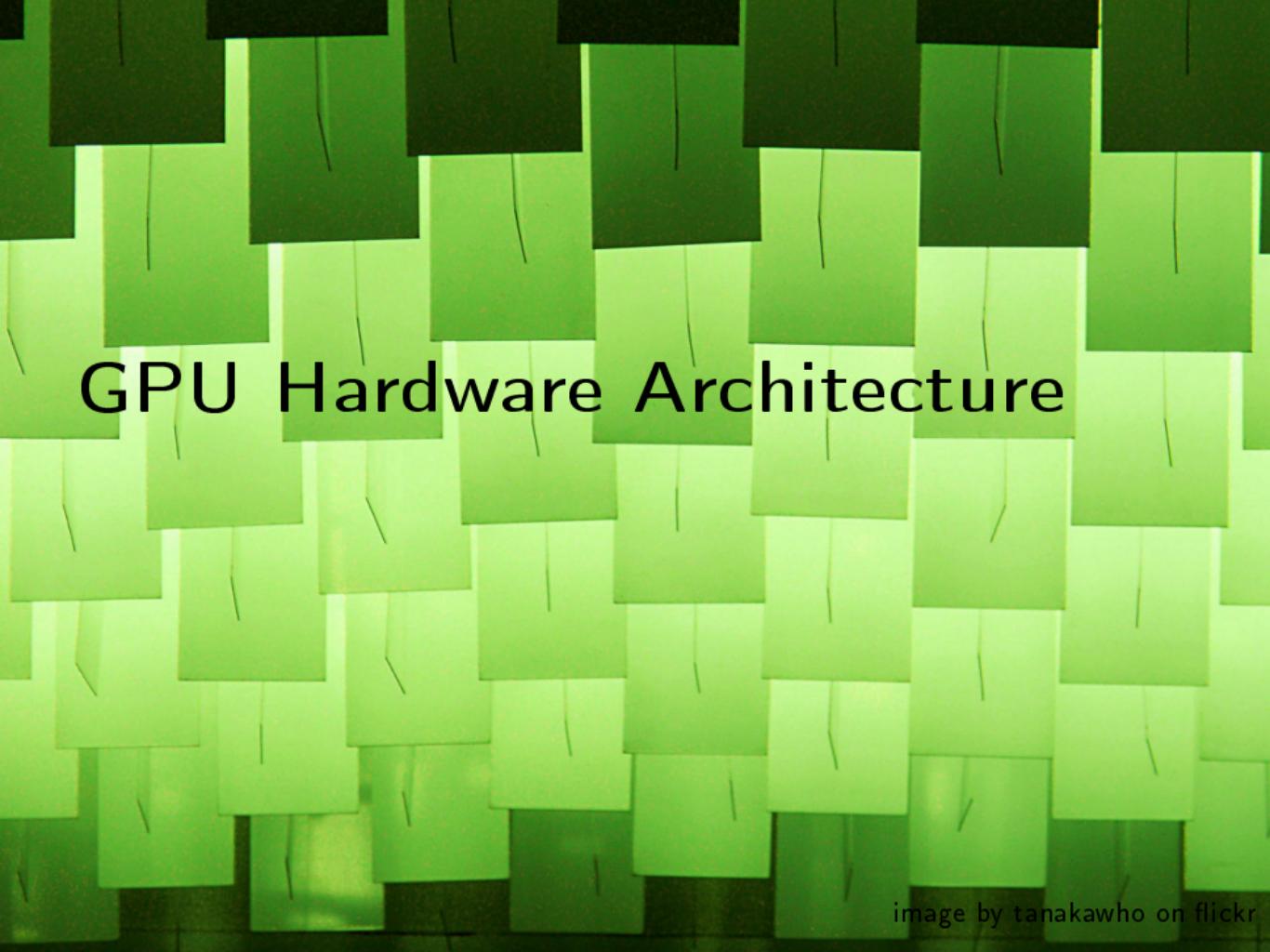
May 29th 2010

Schedule

- Theory
 - GPU Hardware Architecture
 - Stream Programming Model
 - Writing GPU code
- Lunch Break
- Lab Assignments
 - Numerical Integration
 - Reduction

Why GPGPU?

- Speed up parallel applications (signal/image processing, scientific/high performance computing)
- Focus on throughput rather than latency
- Small-sized data parallel applications (e.g. matrix multiplication) compared to standard applications (e.g. a word processor)

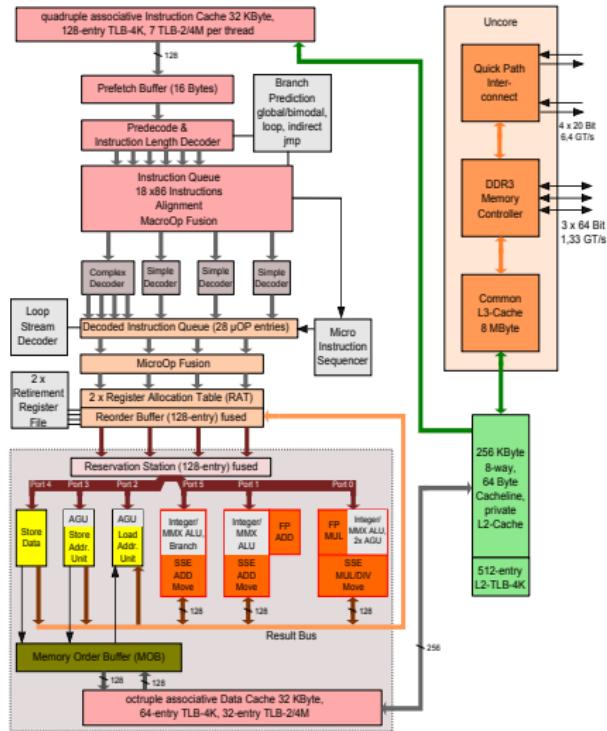
The background of the slide features a repeating pattern of green and black squares, creating a digital or pixelated effect.

GPU Hardware Architecture

Acknowledgements

- Kayvon Fatahalian – “From shader code to a Teraflop: How shader cores work” <http://graphics.stanford.edu/~kayvonf/> [2]
- David Kirk, Wen-mei W. Hwu – “ECE 498 AL : Applied Parallel Programming” at University of Illinois
<http://courses.ece.illinois.edu/ece498/al/> [1]

Nehalem Microarchitecture



- focus on single-threaded application latency, single instruction stream runs fast
- less than about 5% of the area and the peak power are consumed by arithmetic
- fetch/decode unit, ALU, out-of-order control logic, branch prediction, memory pre-fetcher

from [3]

Programming Multi-Core Processors

To get high throughput on Multi-Core processors we can use

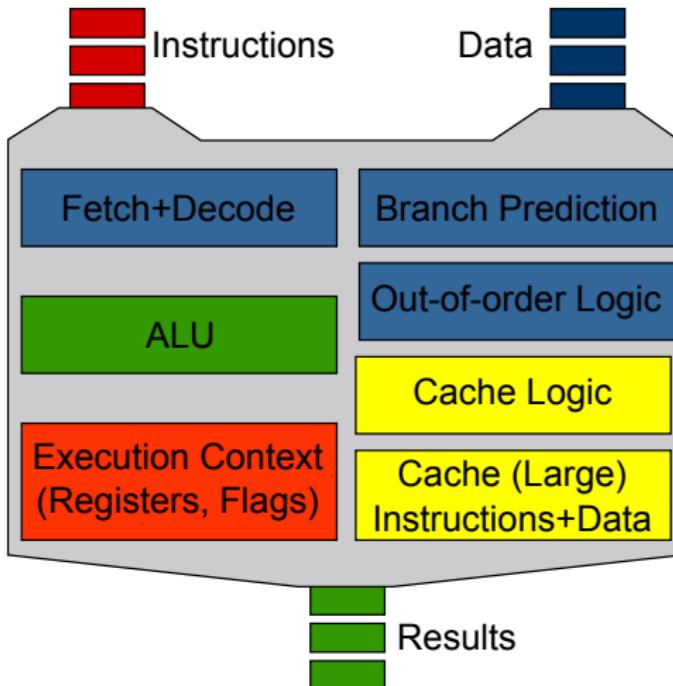
- Multithreading

```
#include <thread>
void computation(){ /* do computation */ }
int main() {
    std::thread t1(computation);
    std::thread t2(computation);
    ...
}
```

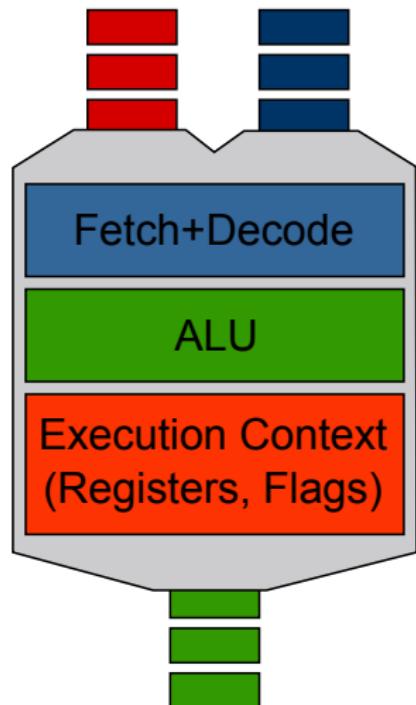
- SIMD

```
#include <emmintrin.h>
__m128 x = x_array[i]; __m128 y = y_array[i];
__m128 z = _mm_add_ps(x, y);
```

Streaming Model of CPU

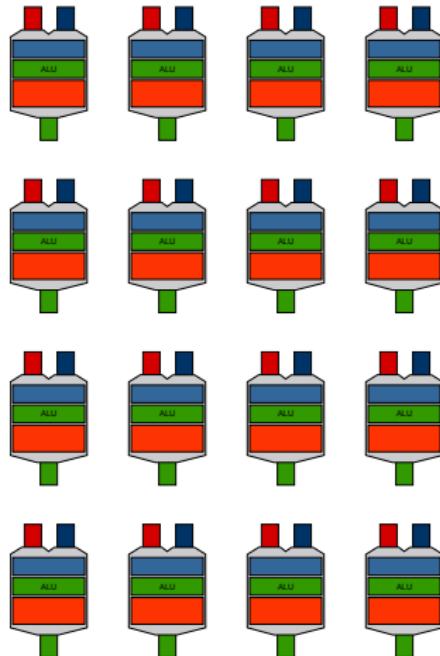


From CPU to GPU (Step 1)



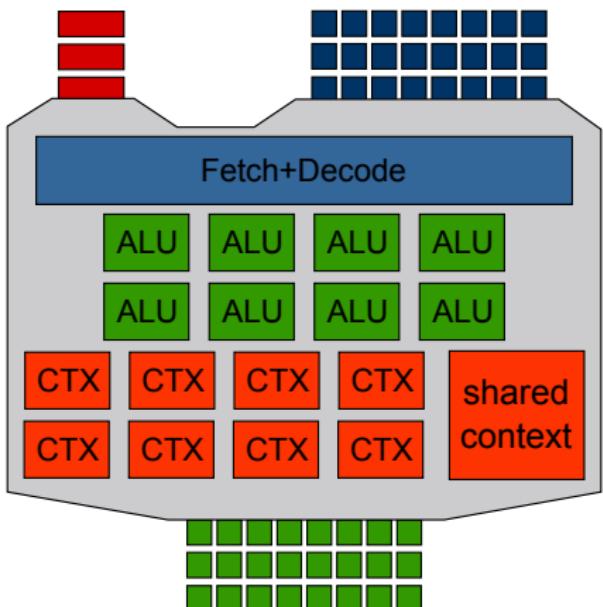
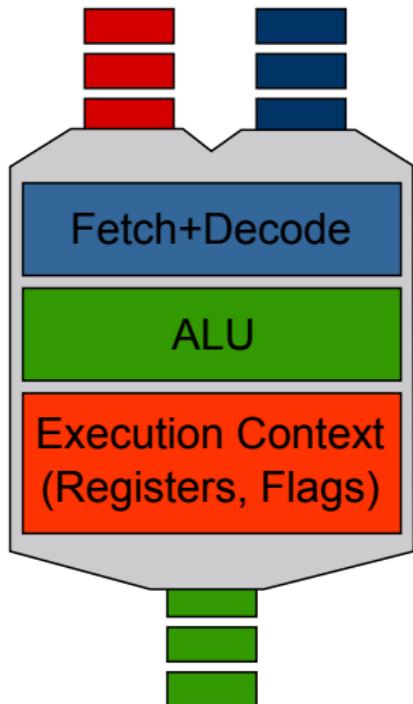
Remove unnecessary components that speed up single instruction stream execution

We can have many of those simple cores

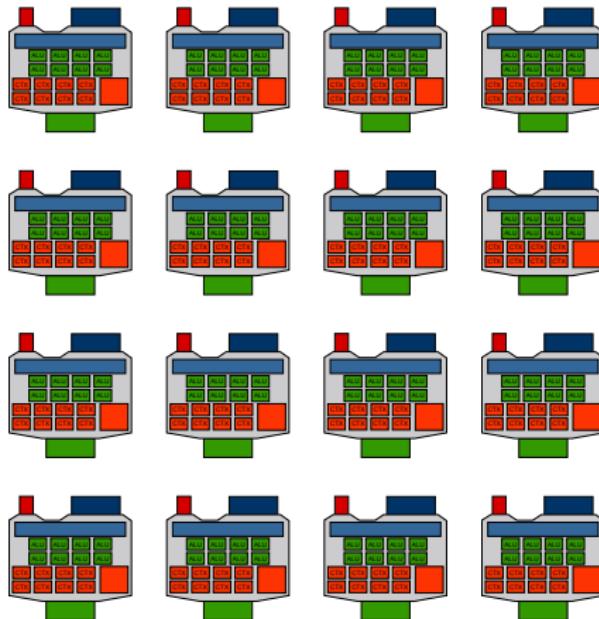


16 cores
16 simultaneous streams

And we can add SIMD processing (Step2)

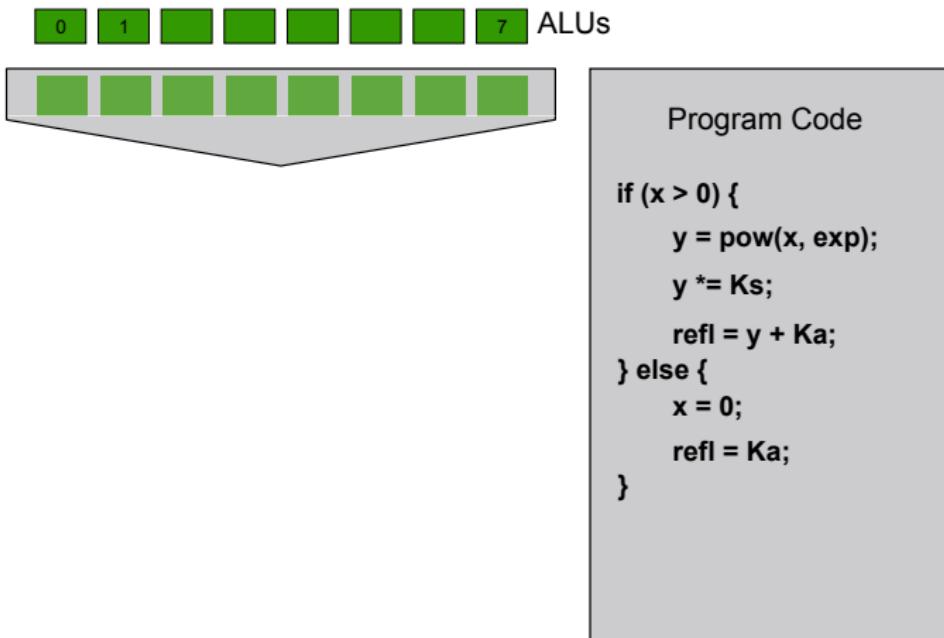


Of course we can have many of those

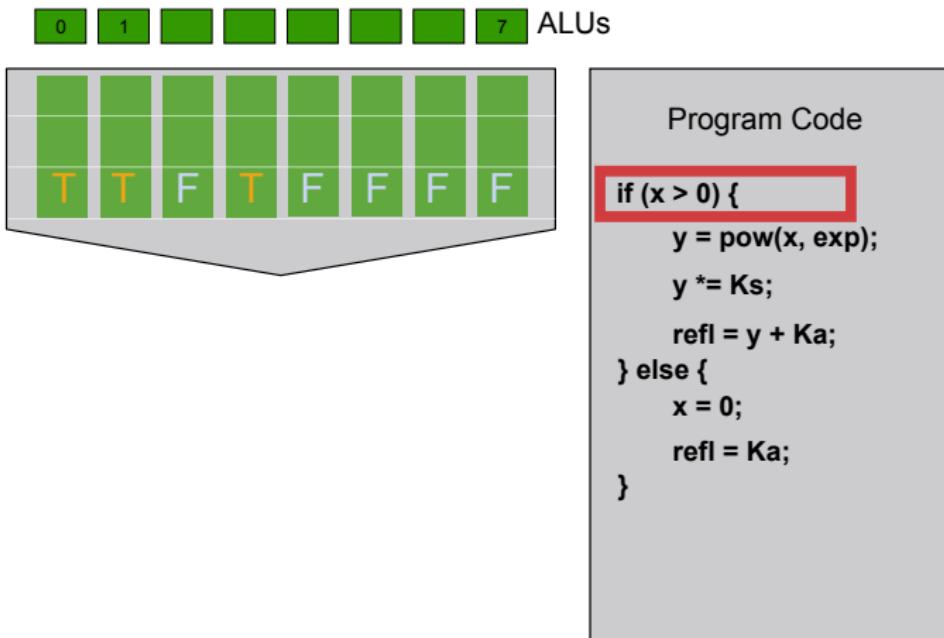


- in this example: 16 cores
 $* 8 \text{ ALUs per core} = 128 \text{ ALUs}$
- in practice (for example GeForce GTX 480): 15 stream processors * 32 ALUs per processor = 480 ALUs

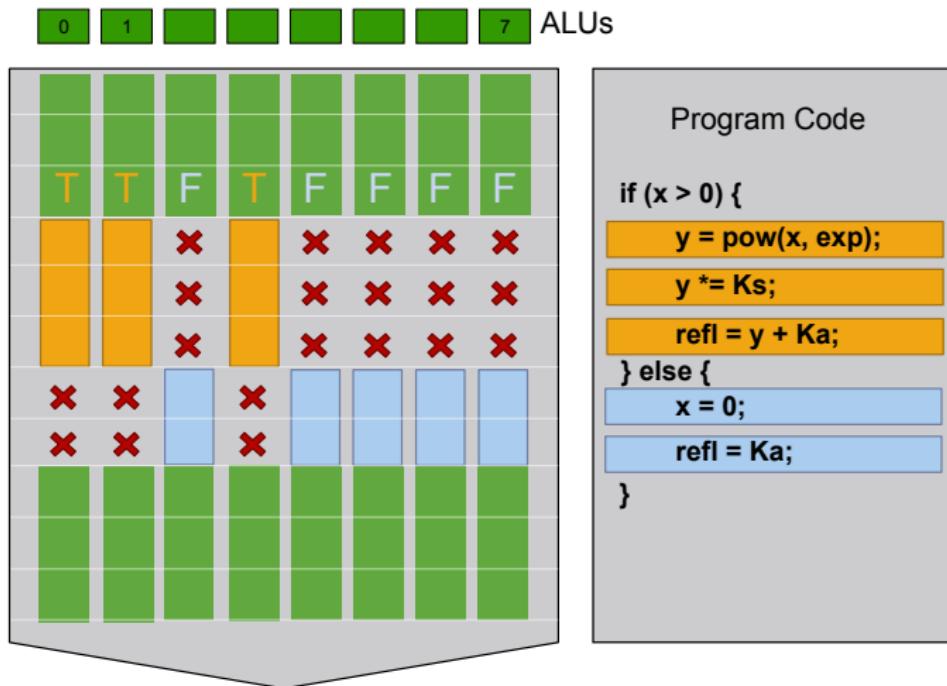
Problem: SIMD and Branching (1)



Problem: SIMD and Branching (2)



Problem: SIMD and Branching (3)



Problem: SIMD and Branching (4)

Remember:

- Branching/divergence can impact performance significantly
- No real solution for this problem: try to avoid branching in between SIMD boundaries

Clarification: there are 2 different ways to do SIMD:

- Explicit vector instructions (a little bit like writing assembler without good tools) or:
- Scalar instructions, implicit vectorization by hardware

Problem: Memory Latency (1)

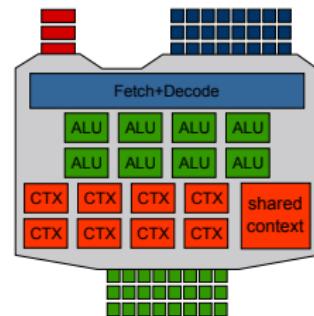
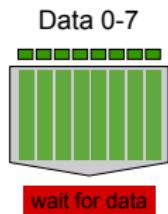
“Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck.” – John Backus

- Memory access latency many times higher than register access (about 400-600 times)
- We removed all the good features that tackle the problem of memory latency

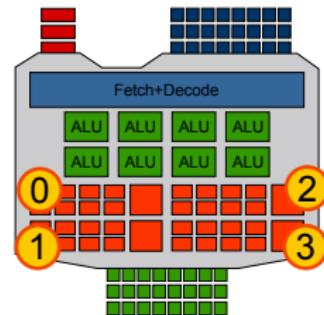
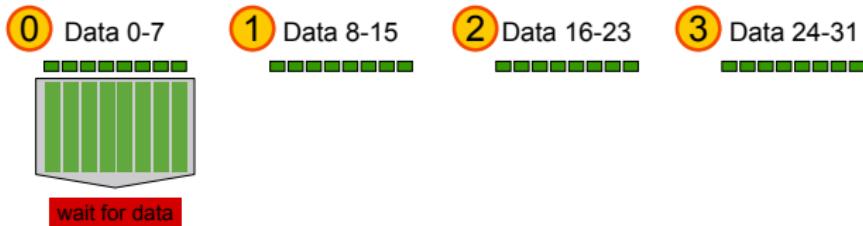
Idea: while waiting we can do something else!

This is possible because of data parallelism.

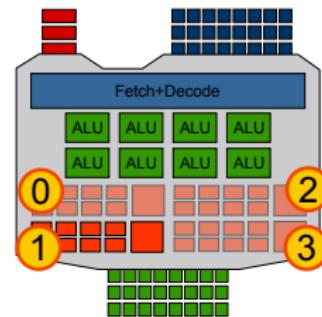
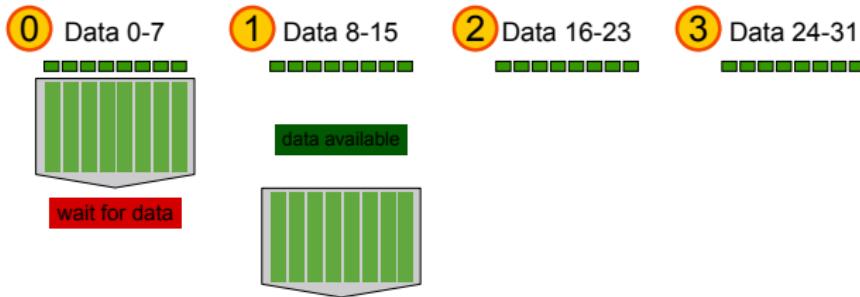
Problem: Memory Latency (2)



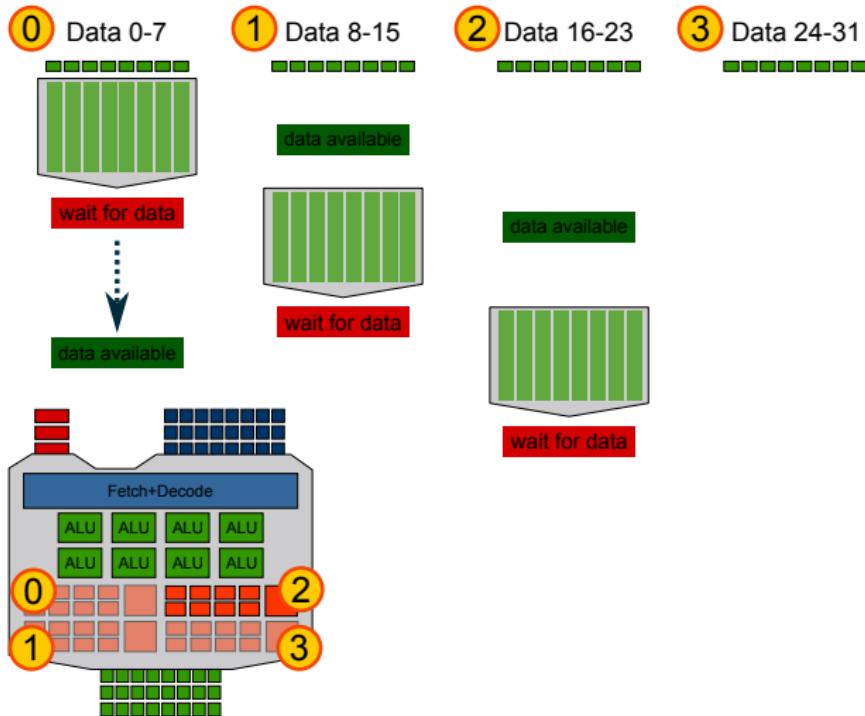
Multiple Execution Contexts (1)



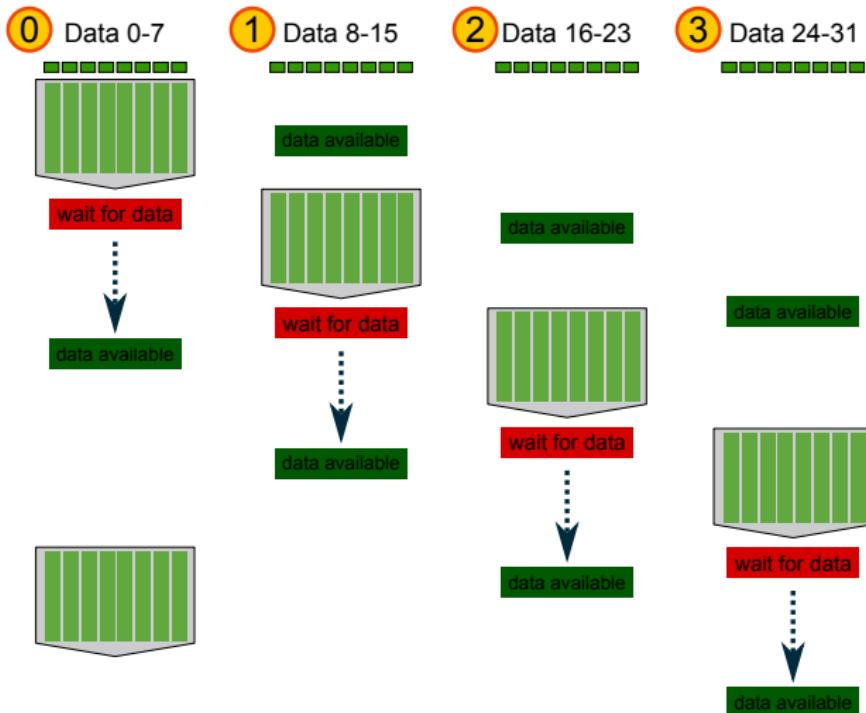
Multiple Execution Contexts (2)



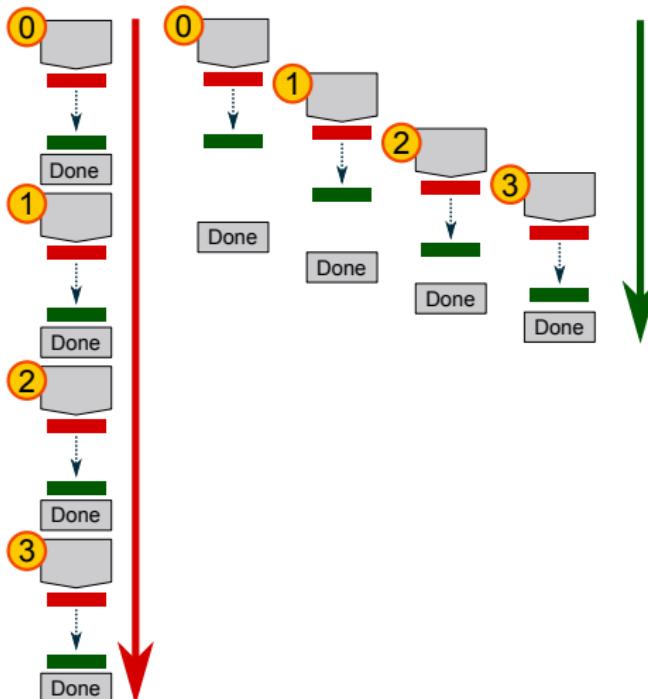
Multiple Execution Contexts (3)



Multiple Execution Contexts (4)



Multiple Execution Contexts Benefit



Multiple Execution Contexts Conclusion

- Great way to hide memory latency/automatically overlap memory access and computation
- Stream processors can dynamically allocate contexts out of a large bank of registers
- If code uses only few registers (a small execution context) many contexts can run simultaneously
- If code uses many registers only a few big contexts can be created: hiding memory access might not be as efficient in this case

Model GPU



Summary

- Use many simplified cores to run in parallel
- Pack cores full of ALUs (SIMD)
- Allow many execution contexts to interleave memory access and computation

Modern GPUs



Nvidia G100 Architecture: Fermi

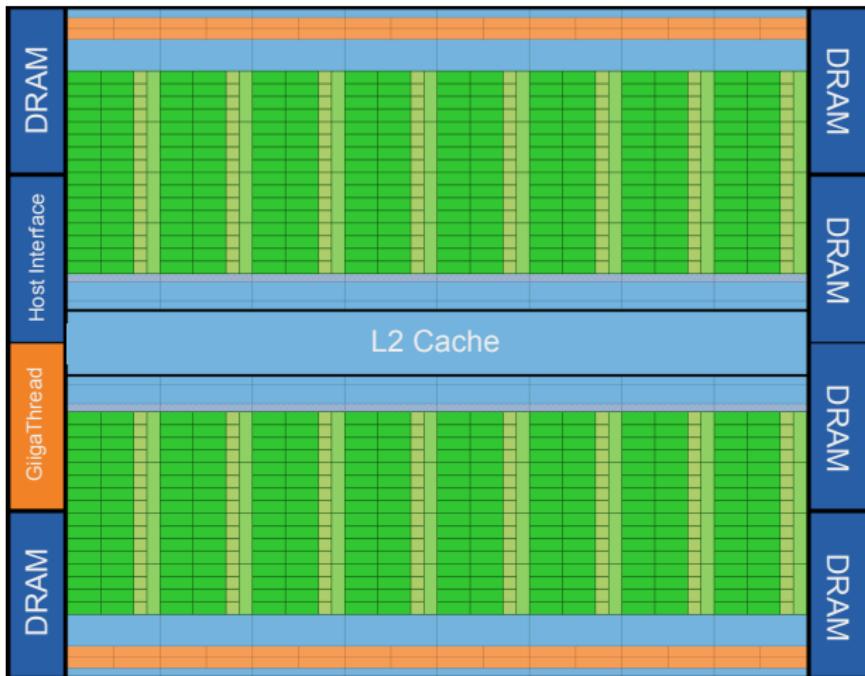
NVIDIA Therminology

- Streaming Multiprocessor
- Stream Processors (scalar ALU)

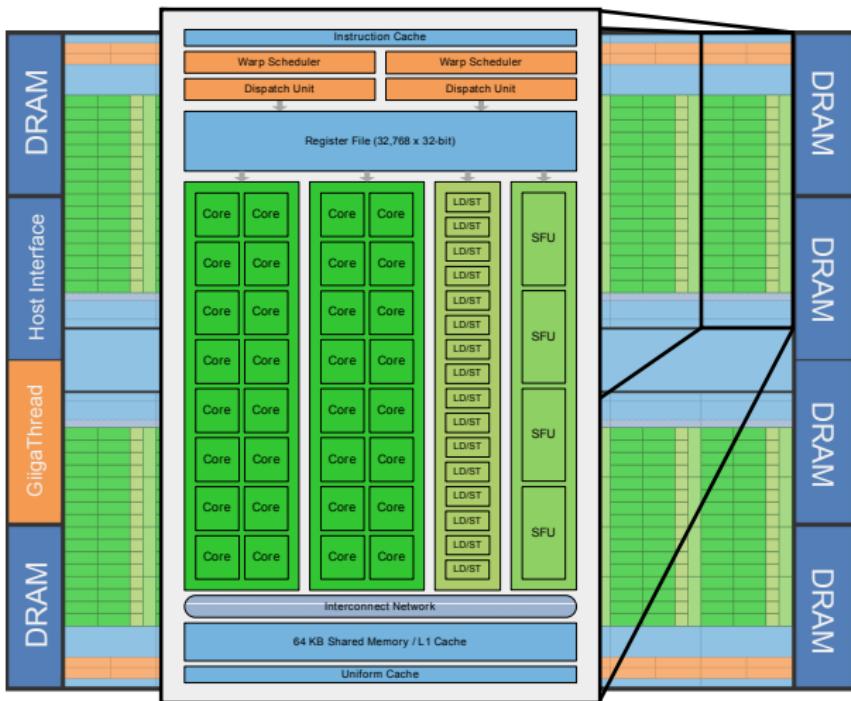
GeForce GTX 480

- Streaming Multiprocessors: 15
- Stream Processors per Streaming Multiprocessor: $32 = 480$ cores
- Processor Clock: 1401 MHz
- Memory Bandwidth: 177.4 GB/s
- 1.3TFLOPS (comparison: Nehalem 100 - 200 GFLOP/s)
- Power requirement: 250W (recommended system power: 600W)
- Maximum GPU Temperature: 105C

Fermi Overview



Fermi Details



GeForce GT 240

- Streaming Multiprocessors: 12
- Stream Processors per Streaming Multiprocessor: 8 = 92 cores
- Processor Clock: 1340 MHz
- Memory Bandwidth: 54.4 GB/s
- 385GFLOPS
- TDP: 69W

Memory

Compute capability	1.0	1.1	1.2	1.3	2.0
32 bit registers	8192		16384		32768
local memory		16kB			512kB
shared memory		16kB in 16 banks			16kB to 48kB in 32 banks
constant memory		64kB (8kB Cache per SM)			
global memory	no cache, strict rules for optimal throughput		no cache, relaxed rules for optimal throughput		16kB to 48kB L1 cache per SM, global L2 cache
texture memory		all of memory (Cache between 6kB and 8kB)			
write only memory			infinite		

Memory

Compute capability	1.0	1.1	1.2	1.3	2.0
32 bit registers	8192		16384		32768
local memory		16kB			512kB
shared memory		16kB in 16 banks		16kB to 48kB in 32 banks	
constant memory		64kB (8kB Cache per SM)			
global memory	no cache, strict rules for optimal throughput	no cache, relaxed rules for optimal throughput		16kB to 48kB L1 cache per SM, global L2 cache	
texture memory		all of memory (Cache between 6kB and 8kB)			

Compute Capability

compute capability	GPUs
1	G80
1.1	G86, G84, G98, G96, G96b, G94, G94b, G92, G92b
1.2	GT218, GT216, GT215
1.3	GT200, GT200b
2	GF100

An aerial photograph of a river delta, likely the Fraser River in British Columbia, Canada. The image shows a complex network of waterways, numerous small islands, and extensive agricultural fields. In the background, a range of mountains is visible under a clear sky.

Stream Programming Model

image by [ecstaticist](#) on flickr

Introduction

- A way of programming massively parallel architectures
- Is closer to sequential paradigm than SIMD paradigm and thus arguably easier to develop

Idea: **Apply a kernel function written like a sequential program to many data elements.**

Simple Example

Element-wise multiplication of 2 2D matrices:

```
a [ y*dimX+x ] = b [ y*dimX+x ] * c [ y*dimX+x ] ;
```

Sequential implementation:

```
for (int y=0; y<dimY; y++)
    for (int x=0; x<dimX; x++)
        a [ y*dimX+x ] = b [ y*dimX+x ] * c [ y*dimX+x ] ;
```

CUDA implementation:

```
int x = threadIdx.x;
int y = blockIdx.x;
a [ y*dimX+x ] = b [ y*dimX+x ] * c [ y*dimX+x ] ;
```

Simple Example continued

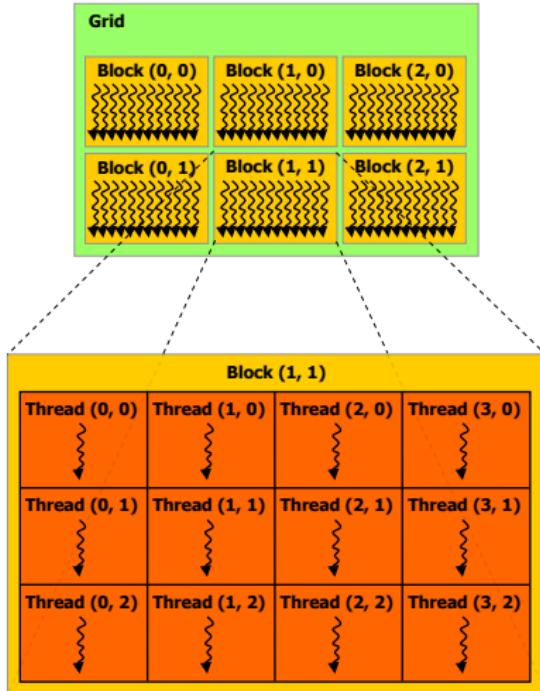
Full sequential code:

```
void elmatrix_mul(T * a, T * b, T * c,
                  int dimX, int dimY) {
    for(int y=0; y<dimY; y++)
        for(int x=0; x<dimX; x++)
            a[y*dimX+x] = b[y*dimX+x] * c[y*dimX+x];
}
elmatrix_mul(a, b, c, 512, 512);
```

Full CUDA code:

```
__global__ void elmatrix_mul(T * a, T * b, T * c)
{
    int x = threadIdx.x; int y = blockIdx.x;
    int dimX = blockDim.x;
    a[y*dimX+x] = b[y*dimX+x] * c[y*dimX+x];
}
elmatrix_mul<<<512, 512>>>(A, B, C);
```

Thread Hierarchy



- grids consists of blocks (1D or 2D)
- maximum number of blocks:
 65535×65535
- blocks consist of threads (1D, 2D or 3D)
- maximum number of threads per block: 512 (1024 on CC2.0)
- $\text{gridsize} \times \text{blocksize} = \text{number of threads that are executed}$

Thread Blocks

- inside thread block: synchronization possible
- `__syncthreads()` acts as a barrier at which all threads in the block must wait before any is allowed to proceed
- Thread blocks are required to execute independently
- Inter-block synchronization only possible through kernel exit/relaunch
- Kernel variables:
 - `block`: `blockIdx.x` `blockIdx.y` `gridDim.x` `gridDim.y`
 - `thread`: `threadIdx.x` `threadIdx.y` `threadIdx.z` `blockDim.x` `blockDim.y` `blockDim.z`
 - to start multidimensional kernel:

```
dim3 numBlocks(64,64); dim3 threadsPerBlock(16,16);
kernel<<<numBlocks, threadsPerBlock>>>();
```

Second Example

Element-wise multiplication of 2 **3D matrices**:

given dimX, dimY, dimZ, x, y, z, what is the kernel function?

a[????] =

b[????] * c[????];

Second Example

Element-wise multiplication of 2 **3D matrices**:

given dimX, dimY, dimZ, x, y, z, what is the kernel function?

$a[z \cdot \text{dimX} \cdot \text{dimY} + y \cdot \text{dimX} + x] =$

$b[z \cdot \text{dimX} \cdot \text{dimY} + y \cdot \text{dimX} + x] * c[z \cdot \text{dimX} \cdot \text{dimY} + y \cdot \text{dimX} + x];$

Second Example

Element-wise multiplication of 2 **3D matrices**:

given dimX, dimY, dimZ, x, y, z, what is the kernel function?

a[z*dimX*dimY+y*dimX+x] =

b[z*dimX*dimY+y*dimX+x] * c[z*dimX*dimY+y*dimX+x];

Sequential code:

```
void elmatrix_mul_cpu_3d(T * a, T * b, T * c,
                           int dimX, int dimY, int dimZ) {
    for(int z=0; z<dimZ; z++)
        for(int y=0; y<dimY; y++)
            for(int x=0; x<dimX; x++)
                a[z*dimX*dimY+y*dimX+x] =
                    b[z*dimX*dimY+y*dimX+x] * c[z*dimX*dimY+y*dimX+x];
}
```

CUDA Solution (1)

```
__global__ void elmatrix_mul_gpu_kernel_3d(T * a, T * b, T * c, int dimX)
{
    a[z*dimX*dimY+y*dimX+x] = b[z*dimX*dimY+y*dimX+x] * c[z*dimX*dimY+y*dimX+x];
}

elmatrix_mul_gpu_kernel_3d<<<dimZ, dimY>>>(A, B, C, dimX);
```

CUDA Solution (1)

Third dimension in for-loop

```
__global__ void elmatrix_mul_gpu_kernel_3d(T * a, T * b, T * c, int dimX)
{
    int y = threadIdx.x;
    int z = blockIdx.x;
    int dimY = blockDim.x;
    for(int x=0; x<dimX; x++)
        a[z*dimX*dimY+y*dimX+x] = b[z*dimX*dimY+y*dimX+x] * c[z*dimX*dimY+y*dimX+x];
}

elmatrix_mul_gpu_kernel_3d<<<dimZ, dimY>>>(A, B, C, dimX);
```

CUDA Solution (2)

```
__global__ void elmatrix_mul_gpu_kernel_3d(T * a, T * b, T * c)
{
    int x = ?;
    int y = ?;
    int z = ?;
    int dimX = ?;
    int dimY = ?;
    a[z*dimX*dimY+y*dimX+x] = b[z*dimX*dimY+y*dimX+x] * c[z*dimX*dimY+y*dimX+x];
}

dim3 gridsize(?);
dim3 blocksize(?);
elmatrix_mul_gpu_kernel_3d<<<gridsize, blocksize>>>(A, B, C);
```

CUDA Solution (2)

2D grid

```
__global__ void elmatrix_mul_gpu_kernel_3d(T * a, T * b, T * c)
{
    int x = threadIdx.x;
    int y = blockIdx.x;
    int z = blockIdx.y;
    int dimX = blockDim.x;
    int dimY = gridDim.x;
    a[z*dimX*dimY+y*dimX+x] = b[z*dimX*dimY+y*dimX+x] * c[z*dimX*dimY+y*dimX+x];
}

dim3 gridsize(dimY, dimZ);
dim3 blocksize(dimX);
elmatrix_mul_gpu_kernel_3d<<<gridsize, blocksize>>>(A, B, C);
```

Writing GPU code

Warps

- A multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps
- A multiprocessor partitions blocks into warps that get scheduled by a warp scheduler for execution
- They execute together but may diverge (branches are serialized)
- Branch divergence occurs only within a warp; different warps execute independently

Efficient Memory Access (1)

- Memory access often limits the performance of kernels: we call them memory-bound
- Global memory access can take up to 400-600 times as long as register access
- For efficient memory access a half-warp must access:
 - words of size 4, 8, or 16 bytes
 - words that are placed right next to each other
 - words one after another (thread0 word0, thread1 word1..)
 - naturally aligned addresses
- If requirements are met 64-byte, 128-byte or 2x128-byte memory transactions are issued
- If not: 16 separate 32-byte memory are issued
- This is the requirement for CC 1.0 and CC 1.1

Efficient Memory Access (2)

For CC 1.2 and 1.3 the following protocol applies:

- Find the memory segment that contains the address requested by the first thread
- Find all other active threads whose requested address lies in the same segment
- Reduce the transaction size, if possible (if only the first part of the segment is used, reduce the size)
- Carry out the memory transaction
- Repeat until all threads in the half-warp are serviced

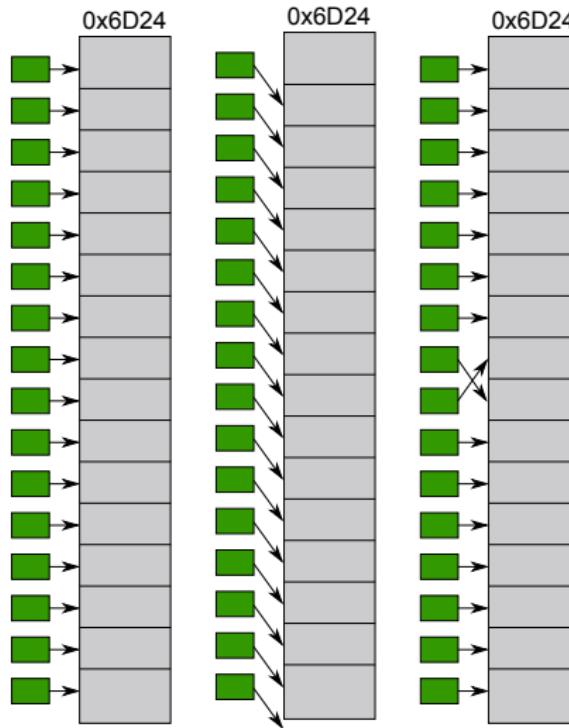
For CC 2.0 a L2 cache is available

Efficient Memory Access (3)

Memory access optimization ideas:

- Align memory naturally
- Pad multi-dimensional memory to a multiple of the element size and to a multiple of the half-warp size
- Optimize access pattern for locality (no strides within half-warp)
- Utilize shared memory as cache: replace global memory access with shared memory access

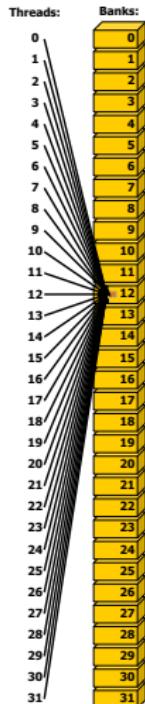
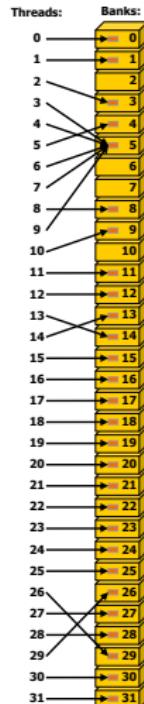
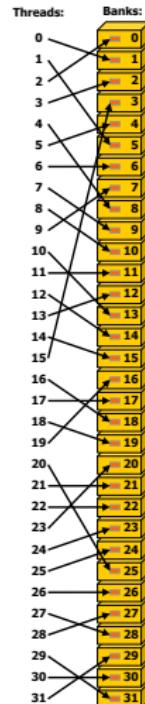
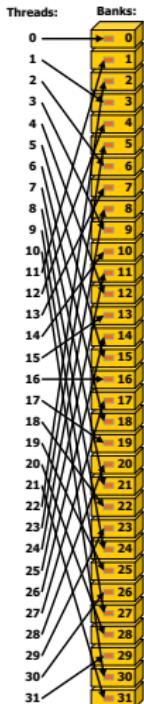
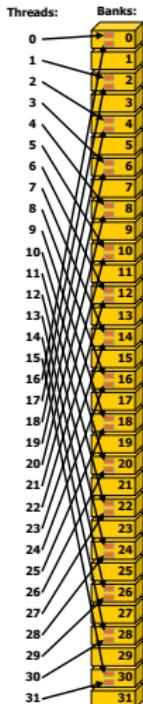
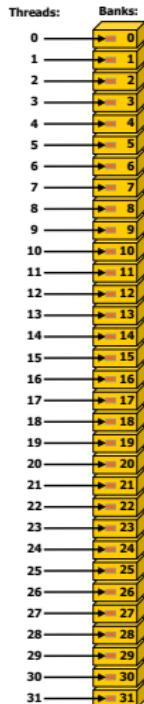
Efficient Memory Access Example



Shared Memory (1)

- Dedicated memory per multiprocessor
- Accessible per block (all threads in a block can access the same data)
- Divided into equally-sized memory modules, called banks, for fast access: memory read or write request made of n addresses that fall in n distinct memory banks can be serviced simultaneously; if not: bank conflict, serialization of access
- CC 2.0 32 banks, CC < 2.0 16 banks

Shared Memory (2)



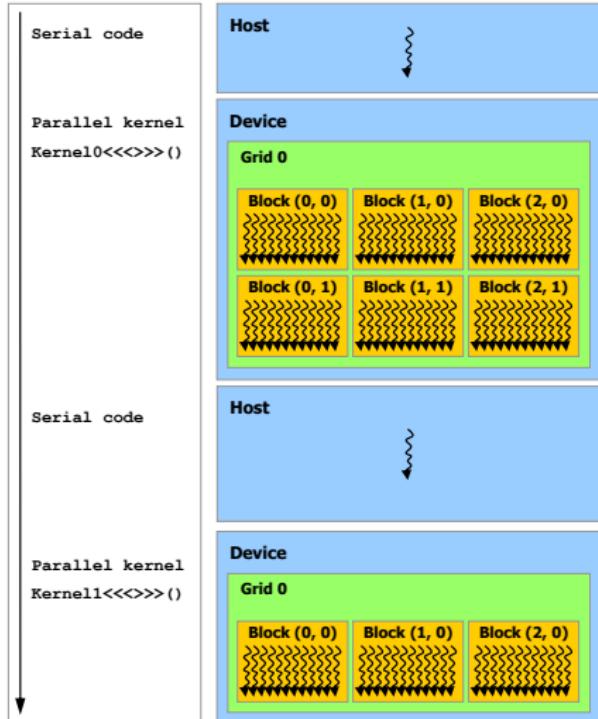
Other Memories

- Local memory: thread private memory in global memory (slow) that is used if no enough registers are used
- Constant memory: cached, read only, good to store constant kernel parameters, physical constants, etc.
- Texture memory: very powerfull addressing modes including interpolation, cached

Memory Overview

Memory	Location	Cache	Access	Scope	Latency
Register	On Chip	N/A	RW	1 Thread	1 Cycle
Shared	On Chip	N/A	RW	Block	2 Cycles
Local	Off Chip	CC 2.0	RW	1 Thread	400-600 Cycles (on cache miss)
Global	Off Chip	CC 2.0	RW	All Threads	400-600 Cycles (on cache miss)
Constant	Off Chip	Yes	R	All Threads	DRAM, cached
Texture	Off Chip	Yes	R	All Threads	DRAM, cached

Heterogeneous Programming



- GPU as co-processor in host application
- GPU and CPU (device and host) maintain their own separate memory spaces in DRAM memory
- Application manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime
- Kernel calls are asynchronous
- Overlapping of kernel execution and memory transfer possible
- CC 2.0: concurrent kernels

Timing CUDA Code

Many CUDA calls are asynchronous, use

`cudaThreadSynchronize()`

when timing kernel calls, or use events:

```
cudaEvent_t start , stop ;
float time;                                     // time in ms
cudaEventCreate(&start );
cudaEventCreate(&stop );
cudaEventRecord( start , 0 );
kernel<<<grid , threads>>>(d_odata , d_idata );
cudaEventRecord( stop , 0 );
cudaEventSynchronize(stop );
cudaEventElapsedTime(&time , start , stop );
cudaEventDestroy( start ); cudaEventDestroy( stop );
```

Memory Management (1)

Allocate/deallocate global memory:

```
cudaMalloc(void ** devPtr, size_t size)
cudaFree(void * devPtr)
```

Copy memory to the device, from the device, on the device:

```
cudaMemcpy(void * dst, const void * src,
           size_t count, enum cudaMemcpyKind kind)
```

Shared memory static allocation inside kernel:

```
__shared__ float memory[BLOCK_SIZE]
```

Pinned host memory for fast host <-> device transfers

```
cudaHostAlloc(void ** ptr, size_t size, unsigned int flags)
cudaFreeHost(void * ptr)
```

Memory Management (2)

Constant memory

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
```

Asynchronous memory transfers

```
cudaMemcpyAsync(void * dst, const void * src, size_t count,
enum cudaMemcpyKind kind, cudaStream_t stream)
```

Overlapping memory transfer and computation

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]); float* hostPtr;
cudaMallocHost((void**)&hostPtr, 2 * size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                   size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
        (outputDevPtr + i * size, inputDevPtr + i * size, size);
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size,
                   outputDevPtr + i * size, size, cudaMemcpyDeviceToHost,
                   stream[i]);
cudaThreadSynchronize();
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

Error Handling

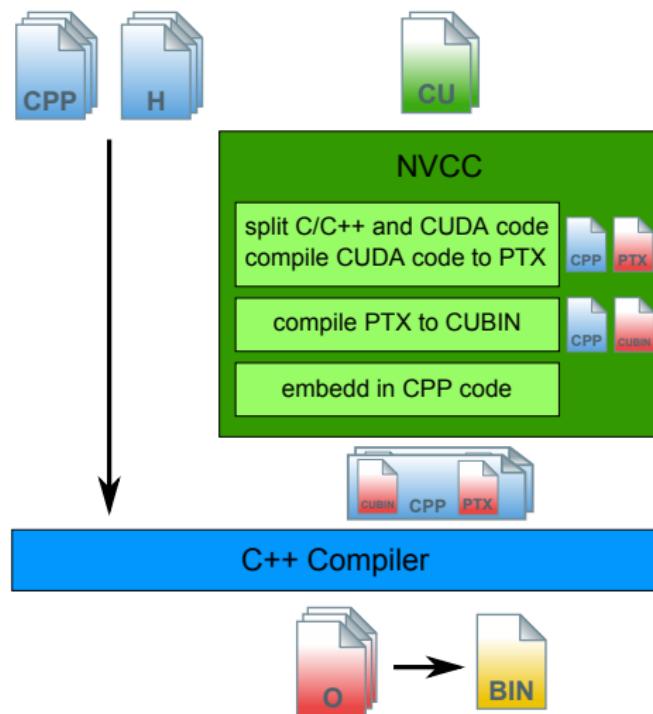
- All runtime functions return an error code
- Runtime maintains a per thread error variable that can be accessed with

```
cudaError_t cudaGetLastError(void)  
const char* cudaGetErrorString(cudaError_t error)
```

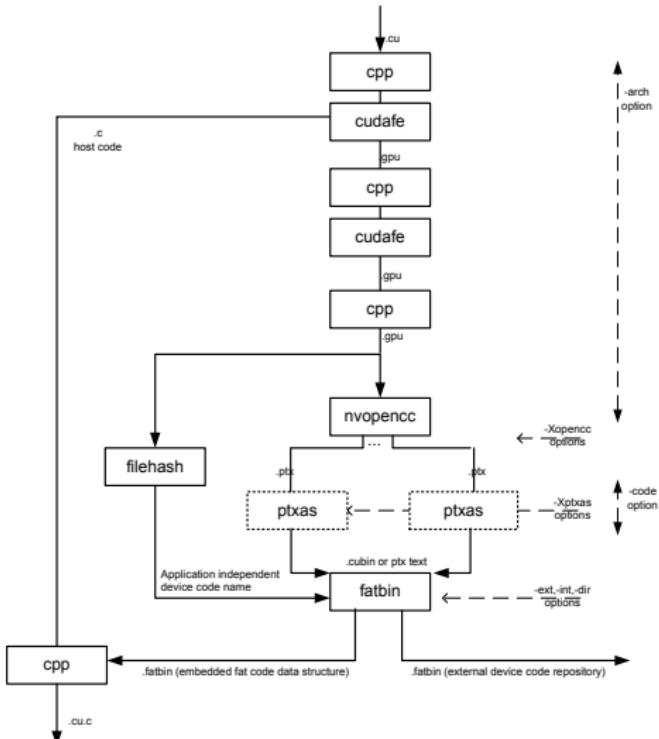
- Beware of asynchronous errors, most of them can not be reported directly, use combination of

```
cudaError_t cudaGetLastError(void)  
cudaThreadSynchronize()
```

NVCC



NVCC Details



Optimizing CUDA Code

- What should be the goal of the optimization?
- Answer: reach peak performance of device
- 2 possible metrics:
 - Instruction throughput (e.g. GFLOP/s) for compute-bound kernels
 - Data throughput (e.g. GB/s) for memory-bound kernels

Examples for GeForce 480 GTX

Instruction Throughput

- Not trivial to determine the number of instructions per kernel
- The CUDA Visual Profiler helps (provides instruction throughput ratio)
- Calculation of max GFLOP/s:
 - Processor Clock 1401 MHz
 - CUDA Cores 480
 - fmad instruction counts as 2 flops
 - $1.4 \text{GFLOP/s} * 480 * 2 = 1344.96 \text{GFLOPs/s}$

Data throughput

- Calculation of max GFLOP/s:
 - Memory bus width: 384bit
 - Memory clock: 924MHz
 - $1848 \text{MHz} * 384 \text{bit} * 2 = 177.4 \text{GB/s}$

Performance Optimization Strategies

- Focus first on finding ways to parallelize sequential code
- Use bandwidth of your computation as a metric to measure performance
- Minimize data transfer between the host and the device
- Ensure global memory accesses are coalesced whenever possible
- Minimize the use of global memory and prefer shared memory access where possible
- Avoid branching within warps

References (1)

-  Wen-mei W. Hwu David Kirk.
ECE498 course at University of Illinois: CUDA Threading
Hardware, CUDA Memory Hardware.
online, 5 2010.
-  Kayvon Fatahalian.
From shader code to a teraflop: How shader cores work.
online, 12 2009.
-  Kunle; Hofstee H. Peter Keckler, Stephen W.; Olukotun, editor.
Multicore Processors and Systems.
Springer, 2009.
-  NVIDIA.
CUDA Best Practices Guide, 3.0 edition, March 2010.

References (2)



NVIDIA.

CUDA C Programming Guide, 3.0 edition, March 2010.