Lecture 11

More CUDA

In this episode...

- Error checking
- Query device capabilities
- CUDA events
- More on CUDA memory:

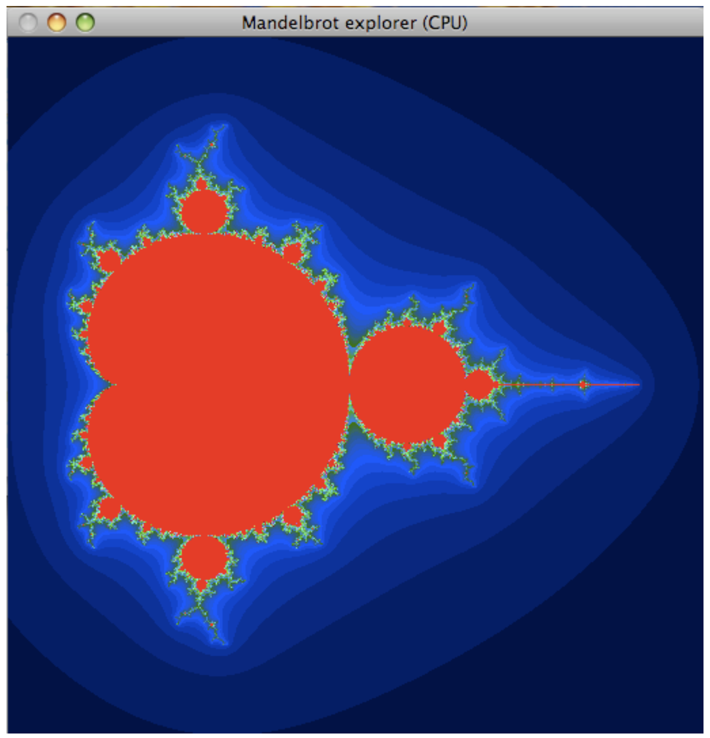
Coalescing, Constant memory, Texture memory...

- OpenGL integration
- Reduction (intro)

Lab 4

Starts monday! Tested and ready! (There were installation problems in Southfork, now resolved.)

”Mandelbrot revisited” part, to follow up lab 1.



The story so far...

- CUDA and its language extensions
- The CUDA architecture
- Intro to memory
- Matrix multiplication example, using shared memory

CUDA and its language extensions

Kernel invocation myKernel<<<>>>()

__global__ __device__ __host__

cudaMalloc(), cudaMemcpy()

threadIdx, blockIdx, blockDim, gridDim

Using nvcc

The CUDA architecture

Blocks and threads

Grid-block-thread hierarchy

Indexing data with thread/block numbers

Intro to memory

global memory

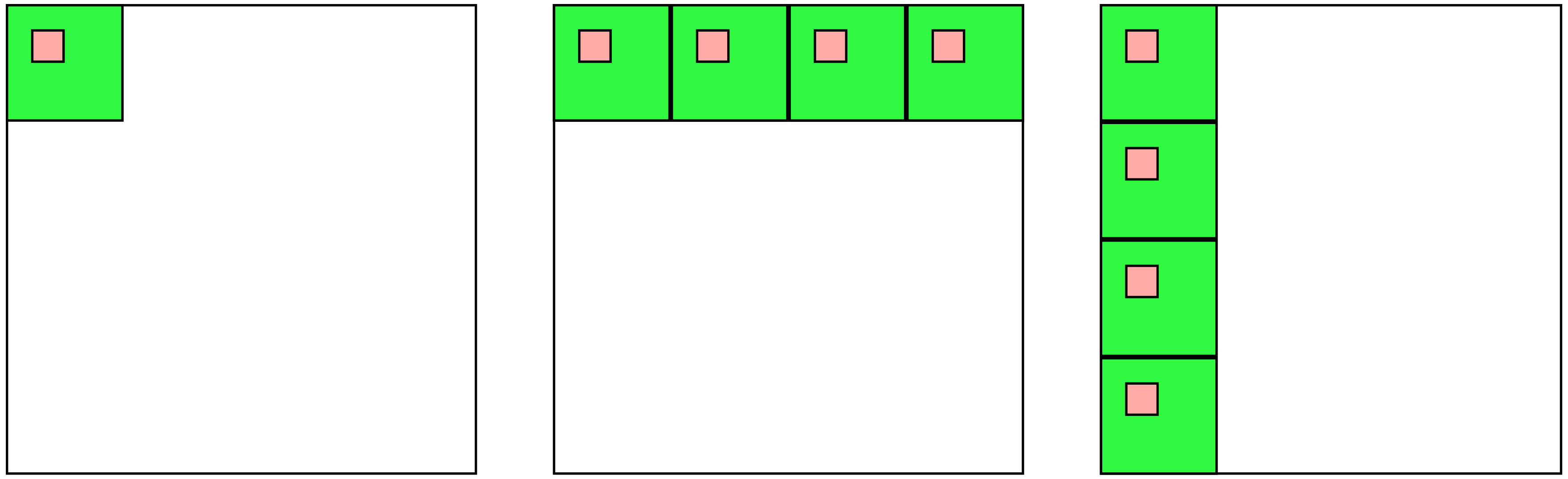
shared memory

constant memory

local memory

texture memory/texture units

Matrix multiplication example, using shared memory



Huge speedup - my GPU went from questionable performance to clearly faster than CPU!

Over to today’s episode:

Lecture questions:

1. Why can using constant memory improve performance?

2. What is CUDA Events used for?

3. What does coalescing mean and what should we do to get a speedup from coalescing?

4. How can you efficiently calculate the maximum of a dataset in parallel?

Error checking

- **Functions returns error codes (but kernel launch does not)**
  - **cudaGetLastError()**
  - **cudaPeekLastError()**

Asynchronous error checking

Asynchronous errors can not be returned by the function call!

Call cudaDeviceSynchronize() and check its returned error code.

More synchronization

No, synchronization isn't *that* simple.

__syncthreads()

cudaDeviceSynchronize()

cudaStreamSynchronize()

More synchronization

__syncthreads() is used inside a kernel.
Stop thread until all threds reach the location!

cudaDeviceSynchronize() is used from the host.
Wait until all current kernels finish.

cudaStreamSynchronize() waits until all kernels in a *stream* finish.

***No* synchronization between blocks!**

Query devices

You can't trust all devices to have the same - or even similar - data.

New boards may have totally different data.

Query CUDA for a list of features using cudaGetDeviceProperties()

Example query result (9400M)

```
---- Information for GeForce 9400M ----
Compute capability:  1.1
Total global memory (VRAM):  259712 kB
Total constant Mem:  64 kB
Number of SMs:  2
Shared mem per SM:  16 kB
Registers per SM:  8192
Threads in warp:  32
Max threads per block:  512
Max thread dimensions:  (512, 512, 64)
Max grid dimensions:  (65535, 65535, 1)
```

Example query result 2 (GT 650M)

```
---- Information for GeForce GT 650M ----
Compute capability:  3.0
Total global memory/VRAM:  523968 kB
Total constant Mem:  64 kB
Number of Streaming Multiprocessors (SM):  2
Shared mem per SM:  48 kB
Registers per SM:  65536
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

What is important?

Compute capability - can this board at all work with our program?

Amount of shared memory - make sure we fit.

Max threads, max dimensions - make sure we fit.

Threads in warp: A lower bound for performance.

Number of SMs: Lower bound for blocks

Compute capability

Essentially CUDA/architecture version number.

1.0: Original release.
1.1: Mapped memory, atomic operations.
1.3: Double support.
2.0: Fermi.
3.0: Kepler.
5.0: Maxwell.
6.0: Pascal.

LiTH

<table>
<tr><th>Feature Support</th><th colspan="6">Compute Capability</th></tr>
<tr><th>(Unlisted features are supported for all compute capabilities)</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x, 3.0</th><th>3.5</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)</td><td rowspan="2">No</td><td colspan="5" rowspan="2">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)</td><td colspan="2" rowspan="4">No</td><td colspan="4" rowspan="4">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)</td></tr>
<tr><td>Warp vote functions (Warp Vote Functions)</td></tr>
<tr><td>Double-precision floating-point numbers</td><td colspan="3">No</td><td colspan="3">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)</td><td colspan="4" rowspan="7">No</td><td colspan="2" rowspan="7">Yes</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())</td></tr>
<tr><td>__ballot() (Warp Vote Functions)</td></tr>
<tr><td>__threadfence_system() (Memory Fence Functions)</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
__syncthreads_or() (Synchronization Functions)</td></tr>
<tr><td>Surface functions (Surface Functions)</td></tr>
<tr><td>3D grid of thread blocks</td></tr>
<tr><td>Funnel shift (see reference manual)</td><td colspan="5">No</td><td>Yes</td></tr>
</table>

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1
Hyper-Q	No	No	No	Yes
Dynamic Parallelism	No	No	No	Yes

Compute Capability of Fermi and Kepler GPUs

Compute Capability	1.0	1.1	1.2	1.3	2.0	2.1	3.0	3.5
SM Version	sm_10	sm_11	sm_12	sm_13	sm_20	sm_21	sm_30	sm_35
Threads / Warp	32	32	32	32	32	32	32	32
Warps / Multiprocessor	24	24	32	32	48	48	64	64
Threads / Multiprocessor	768	768	1024	1024	1536	1536	2048	2048
Thread Blocks / Multiprocessor	8	8	8	8	8	8	16	16
Max Shared Memory / Multiprocessor (bytes)	16384	16384	16384	16384	49152	49152	49152	49152
Register File Size	8192	8192	16384	16384	32768	32768	65536	65536
Register Allocation Unit Size	256	256	512	512	64	64	256	256
Allocation Granularity	block	block	block	block	warp	warp	warp	warp
Max Registers / Thread	124	124	124	124	63	63	63	255
Shared Memory Allocation Unit Size	512	512	512	512	128	128	256	256
Warp allocation granularity	2	2	2	2	2	2	4	4
Max Thread Block Size	512	512	512	512	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16384	16384	16384	16384	49152	49152	49152	49152
[note: default at top of list]					16384	16384	16384	16384
							32768	32768
Warp register allocation granularities					64	64	256	256
[note: default at top of list]					128	128		

Do I care about Compute capability?

While learning CUDA - not much. Stick to the basics, it works on all.

But if you write professional CUDA code, of course.

CUDA Events

Timing!

Two ways of timing CUDA programs:

- **CPU timer. Synchronize at start and end.**
- **CUDA Events. Synchronize at end.**

Synchronize? Because CUDA runs asynchronously.

CUDA Events API

cudaEventCreate - initialize an event variable

cudaEventRecord - place a marker in the queue

cudaEventSynchronize - wait until all markers have received values

cudaEventElapsedTime - get the time difference between two events