Lecture 11 (#3 on GPU Computing)

More CUDA

In this episode...

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

Coalescing, Constant memory, Texture memory...

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 involation 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

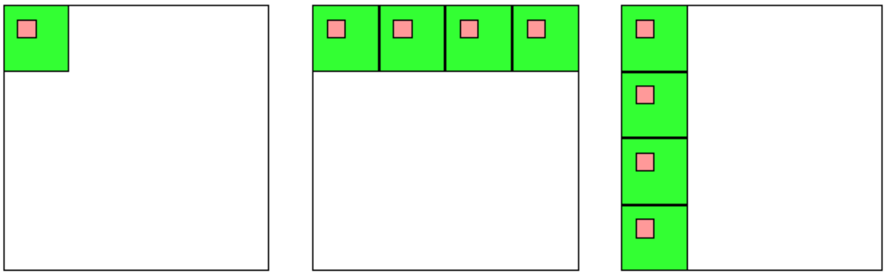
shared memory

constant memory

local memory

texture memory/texture units

Matrix multiplication example, using shared memory



Huge speedup - my measly 9400M went from obvious loser 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?

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

```
---- 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)
```

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.
3.5: Kepler.

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

CUDA Events and Streams

CUDA commands are placed in a queue - a *stream*

Commands are executed, and when a marker is encountered, it is given a time value

We usually only use the default CUDA stream.

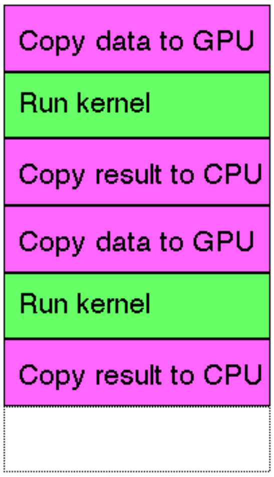
Multiple CUDA streams can be used to overlap work - especially computing and data transfers

---

Single stream computation

The kernel can not run until the data is transfered.

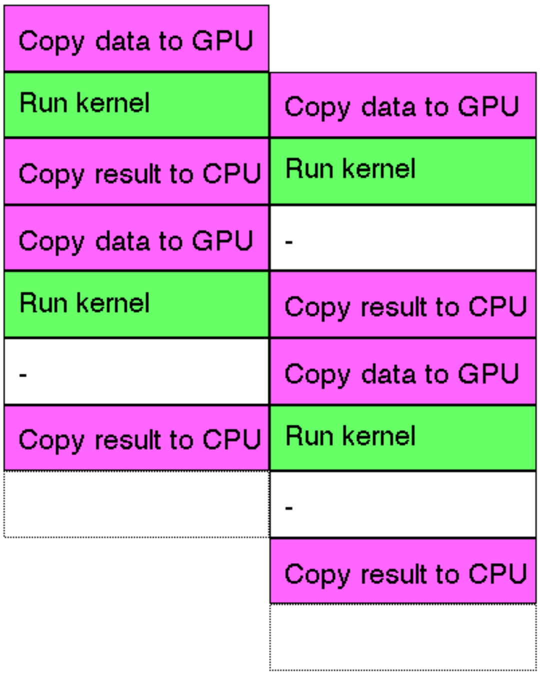
For this example: 2/3 data transfer, 1/3 computation



Dual stream computation

One stream runs a kernel while the other performs data copying.

More time for computing, kernels running 1/2 of the time instead of 1/3.



Copy data to GPU	
Run kernel	Copy data to GPU
Copy result to CPU	Run kernel
Copy data to GPU	-
Run kernel	Copy result to CPU
-	Copy data to GPU
Copy result to CPU	Run kernel
	-
	Copy result to CPU