Lecture 12

A few more CUDA issues

Sorting on GPU

The Fast Fourier Transform

OpenGL interoperability

Lecture questions

1) What is the challenge in parallizing the FFT?

2) In what way does bitonic sort fit the GPU better than many other sorting algorithms?

3) What is the advantage of using CUDA OpenGL interoperability?

Lab 5

All new lab on sorting on the GPU

Prototype done, tested, looks good

Instructions pretty sh*tty

Will be available monday - maybe earlier in preliminary version

So what will it be?

Parallellize bitonic merge sort.

Start from a fairly parallel friendly implementation

Very easy to parallellize for small data sets (i.e. up to 512-1024)

Some more work to make it run with larger data

Not much use for shared memory in lab 4 and 5

Lab 6 is focused entirely on shared memory - but in OpenCL

More memory

Atomics

Pinned memory

Mapped memory

Atomic operations

A special memory access method, for avoiding conflicts and race conditions.

Available from Compute model 1.1.

To use it, specify model with

-arch compute_11

Example: Histogram

Simple method for gathering statitics about a set of data.

Common in image processing.

for all elements i in a[]
h[a[i]] += 1

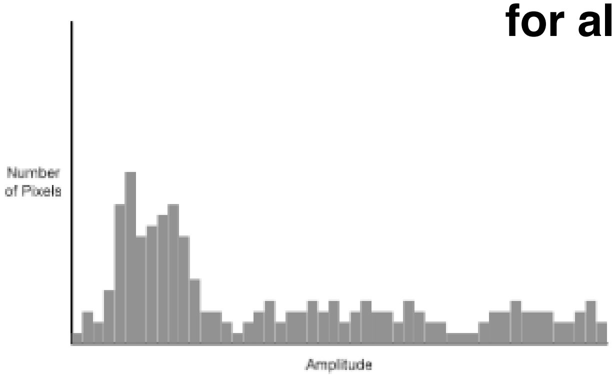
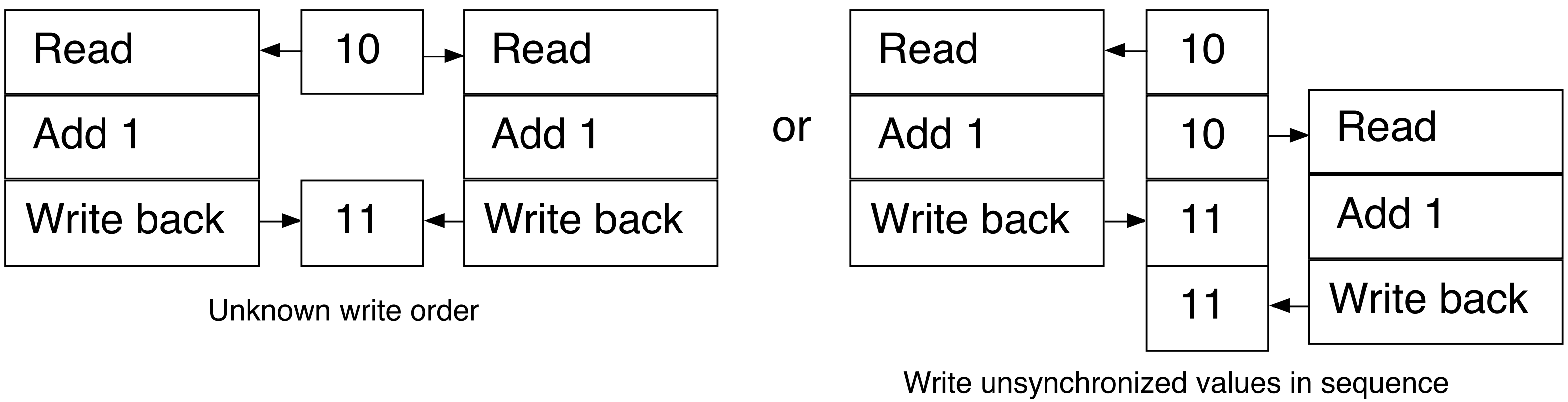


Figure 1: An example of an image histogram

Histogram memory conflicts

If you try to parallelize this operation, threads will write at the same place.

Non-atomic operations will read h[a[i]], add 1, and write back.



Solution: Atomics

Read - modify - write in *one* operation!

Guaranteed not to be subject to racing.

atomicAdd, atomicSub, atomicExch, atomicMin, atomicMax, atomicInc, atomicDec, atomicCAS, atomicAnd, atomicOr, atomicXor

More types in fermi

For a cost: Slower than other operations.

Global memory only (1.1)

Example: Find maximum

for all elements i in a[]
maxValue := max(maxValue, a[i])

Easy? Parallel? No!

All threads will write to the same memory element!

Use atomics? Very slow! All write at the same time, will have to wait - we get sequential performance.

Solution: Split problem in parts, each section finds a local maximum. Merge later.

Pinned memory

Page-locked memory

So far: malloc() and cudaMalloc()

New call: cudaHostAlloc()

Allocated page-locked memory! Fixed physical location!

Pinned memory

Page-locked memory is a limited resource!

If you don't use it: CUDA copies internally to page-locked memory, then DMA to GPU. Transfer time goes up!

Pinned memory, streams, overlapping computation

Pinned memory is part of the optimization with overlapping computations

Not only slight speedup of the data transfer.

cudaMemcpyAsynch(), can copy locked memory asynchonously

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

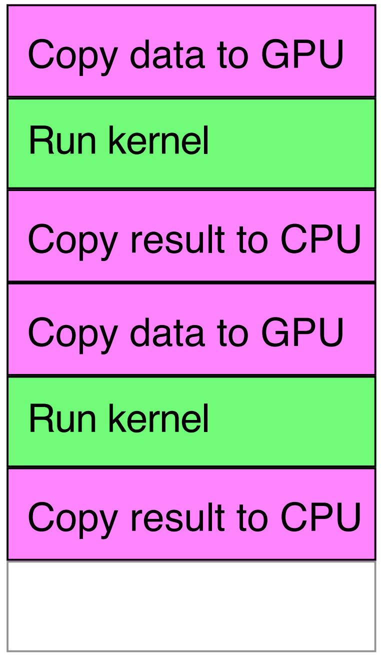
We usually only use the default CUDA stream.

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

Not all devices...

Asynchronous data copying as well as concurrent execution is not guaranteed...

so make a device query!

CU_DEVICE_ATTRIBUTE_ASYNCH_ENGINE_COUNT: Can we copy pinned memory asynch?

CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS: Can we run multiple kernels?

Mapped memory

Mapped memory shared between CPU and GPU, no transfer needed.

Must be page-locked.

Data transfers overlapping kernel execution possible without multiple streams.

Debugging CUDA

Let's get a bit more efficient when your code doesn't work

- Catch error codes
- printf() from kernels
- cudagdb

Catch those error codes

```
// Check for errors everywhere
err = cudaMalloc( (void**)&ad, csize );
// If the GPU won't even take our data we are toasted
if (err) printf("cudaMalloc %d %s\n", err, cudaGetErrorString(err));
...
dim3 dimBlock( blocksize, 1 );
dim3 dimGrid( 1, 1 );
hello<<<dimGrid, dimBlock>>>(ad, bd);
// Most important thing to check? Did the kernel run at all?
err = cudaPeekAtLastError();
if (err) printf("cudaPeekAtLastError %d %s\n", err, cudaGetErrorString(err));
```

and pass them to cudaGetErrorString() for an explanation

printf() from kernels

Yes - printf() if legal in a kernel since Compute Capability 2.0

But don't try to print 100000 messages per second...

More advanced debugger tools

There are more tools to help you out there!

cudagdb

Variant of the GDB debugger

Allows breakpoints and single-stepping CUDA kernels!