



Information Coding / Computer Graphics, ISY, LiTH

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?**



Information Coding / Computer Graphics, ISY, LiTH

Lab 5

Sorting on the GPU

Will be expanded this year.

New version not yet available.



So what is it about?

Parallelize bitonic merge sort.

**Start from a fairly parallel friendly
implementation**

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

Some more work to make it run with larger data



Information Coding / Computer Graphics, ISY, LiTH

**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 in CUDA from Compute model 1.1.

To use it, specify model with

`-arch compute_11`

(or higher)



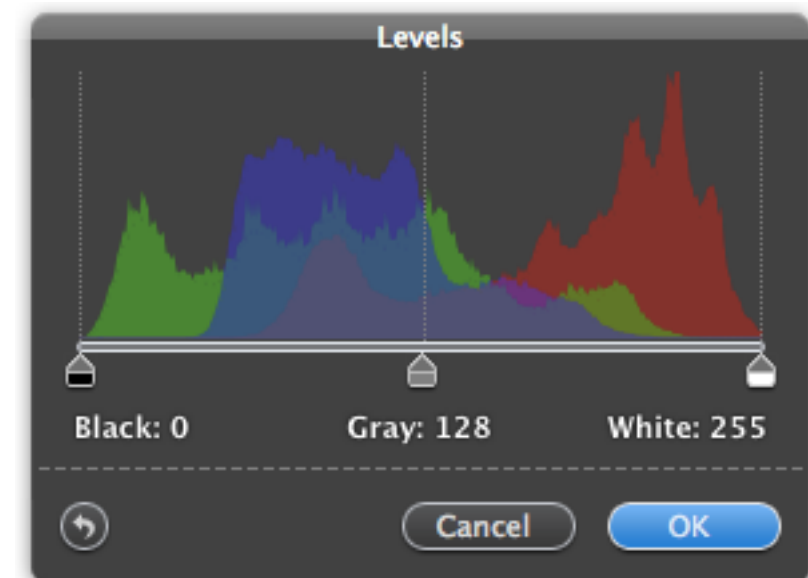
Example: Histogram

Simple method for gathering statistics about a set of data. Much data in, little out.

Common in image processing.



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

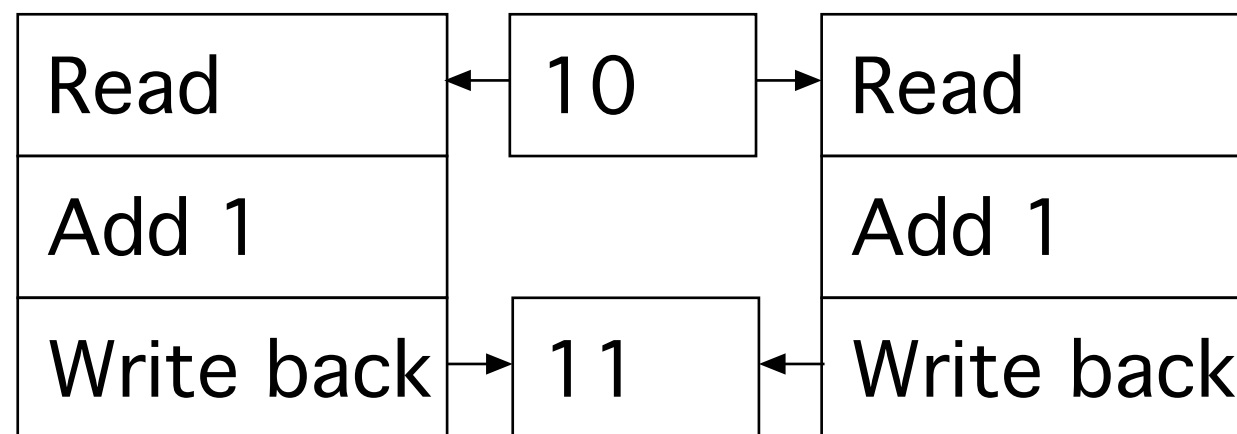




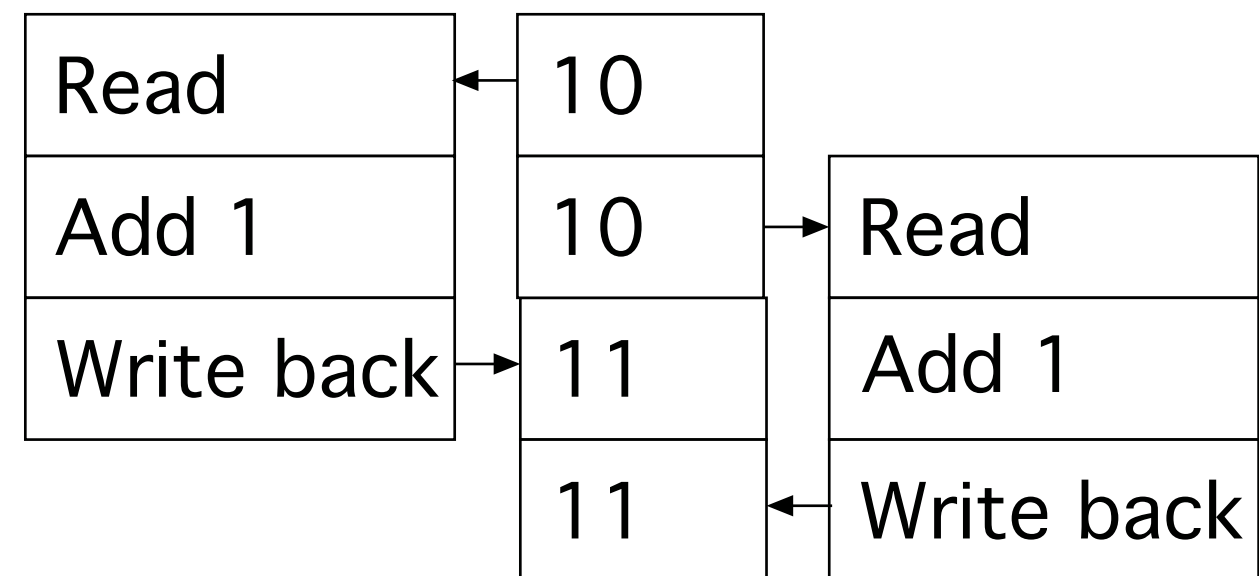
Histogram memory conflicts

If you try to parallelize this operations, multiple threads will write simultaneously at the same item

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



Unknown write order



Write unsynchronized values in sequence



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 and up



Information Coding / Computer Graphics, ISY, LiTH

But it comes for a cost!

Slower than other operations

Global memory only as of Compute Capability 1.1

Simpler but slower than reduction solutions!



Example: Find maximum

**for all elements i in $a[]$
 $\text{maxValue} = \max(\text{maxValue}, a[i])$**

Easy? Yes! Parallel? No!

**All threads will write to the same memory
element!**

**Use atomics? Very slow! All write at the same
time, must wait -> sequential performance!**

Solution: Use reduction instead!



Atomic conclusions

Simplifies some operations

Serializes conflicting operations

Can hurt performance! Don't overuse!



Pinned memory

Can boost performance for memory transfer

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!

For non-pinned memory, CUDA copies it internally to page-locked memory, then DMA to GPU. Transfer time goes up!



Pinned memory, streams, overlapping computation

**Pinned memory is part of an optimization
approach with overlapping computations**

No longer just a slight speedup of data transfer!

**`cudaMemCpyAsync()` can copy locked memory
asynchronously!**



Multiple streams

CUDA commands are placed in a queue, a *stream*!

These are the same queues as you can post CUDA events to.

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 transferred.

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

Copy data to GPU

Run kernel

Copy result to CPU

Copy data to GPU

Run kernel

Copy result to CPU



Dual stream computation

While one stream runs a kernel, the other stream performs data copying,

More time for computing, in this example kernels are 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_ASYNC_ENGINE_COUNT:
Can we copy memory asynch?**

**CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS:
Can we run multiple kernels?**



Information Coding / Computer Graphics, ISY, LiTH

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.

See also *zero-copy memory*.

Mapped memory seems convenient but may not be a performance advantage.



Debugging CUDA

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

- **Catch error codes**
- **printf() from kernels**
 - **cuda-gdb**



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



Information Coding / Computer Graphics, ISY, LiTH

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...**



Information Coding / Computer Graphics, ISY, LiTH

More advanced debugger tools

There are more tools to help you out there!

cuda gdb

Variant of the GDB debugger

**Allows breakpoints and single-stepping
CUDA kernels!**