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
```

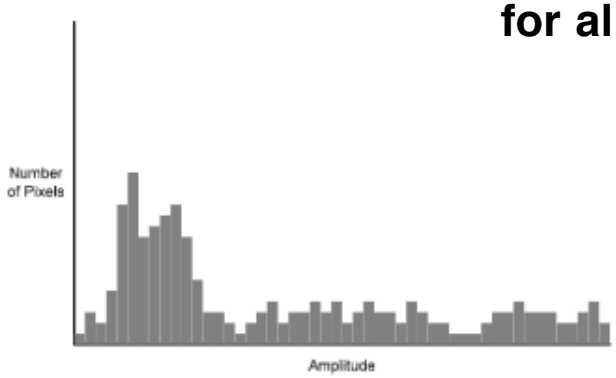
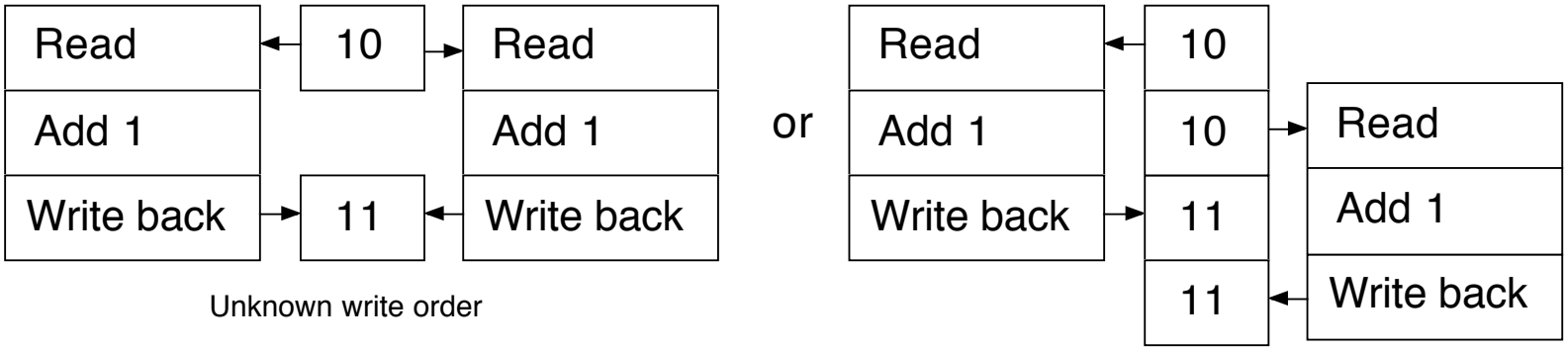


Figure 1: An example of an image histogram

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

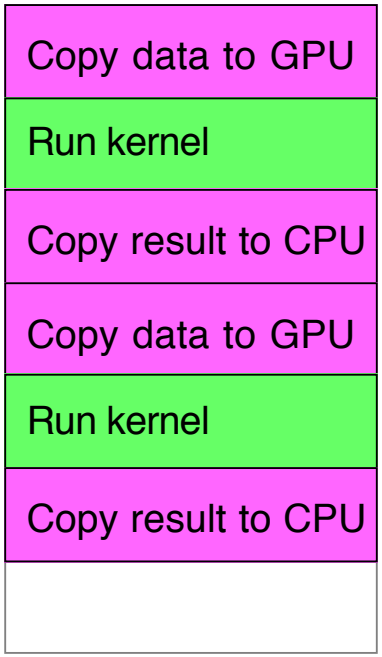
We usually only use the default CUDA stream.

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.

CUDA roundup

Some final comments

From NVIDIA Fermi Tuning Guide:

CUDA Best Practices
The performance guidelines and best practices described in the CUDA Programming Guide [2] and the CUDA Best Practices Guide [3] apply to all CUDA architectures. Programmers must primarily focus on following those recommendations to achieve the best performance.

The high-priority recommendations from those guides are as follows:

- Find ways to parallelize sequential code
- Minimize data transfers between the host and the device
- Adjust kernel launch configuration to maximize device utilization
- Ensure global memory accesses are coalesced
- Replace global memory accesses with shared memory accesses whenever possible
- Avoid different execution paths within the same warp.

Porting to CUDA

1. Parallel-friendly CPU algorithm.
2. Trivial (serial) CUDA implementation.
3. Split to blocks and threads.
4. Take advantage of shared memory.

CUDA emulation mode

CUDA programs can be compiled to CPU only versions.

--device-emulation

Lets you run CUDA (slowly) on non-NVidia hardware

Debugging easier (e.g. printf)

Summary of synchronization

__**synchthreads()** the basic in-kernel call, wait until all threads reach the command.

cudaDeviceSynchronize() complete all streams

cudaStreamSynchronize() complete a specific stream (of commands)

cudaEventSynchronize() waits until all events have occurred (been assigned times)

That’s all folks!

Next: Computing with shaders.