



Lecture 12

Even more CUDA memory

Histogram

Sorting on GPU

Other problems



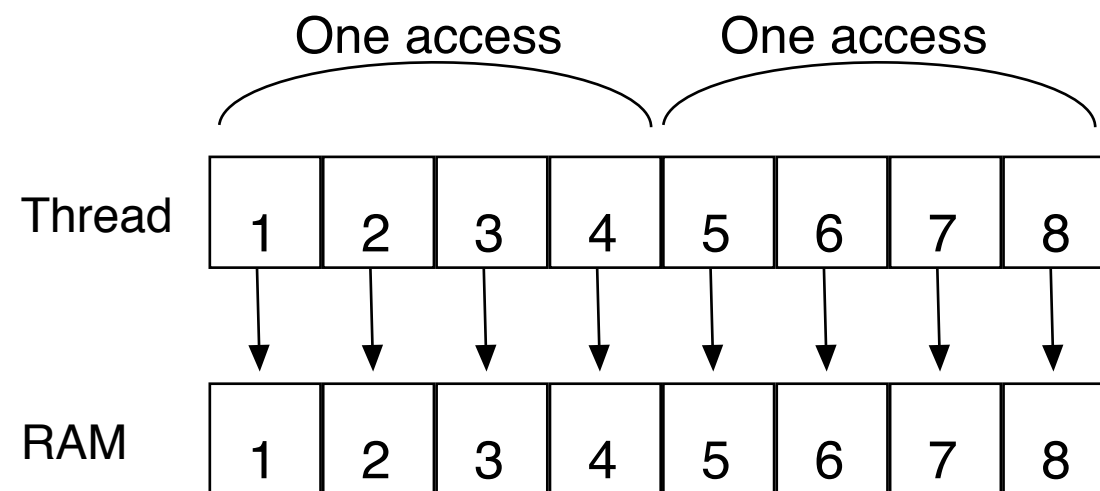
Last time

- Coalescing
- Constant memory
- Texture memory
- Reduction

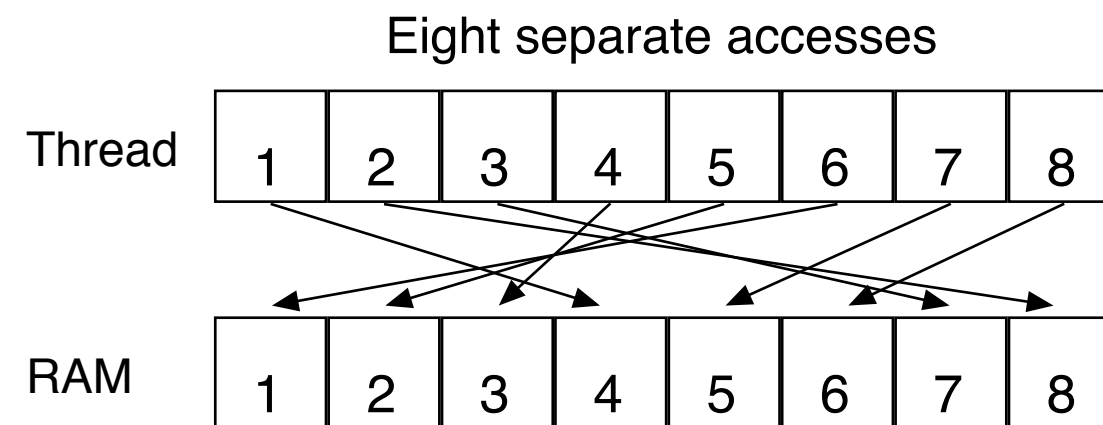


Coalescing - access in order

Example: Assume that we can get 4 data items per transaction.



Good!



Bad!



Reduction - many to few

Problems that are tricky to run in parallel.

Acceleration can be limited or nonexistent for small datasets.

Find minimum, maximum, average...



Tree-based reduction

etc

Break down the problem in small parallel parts.

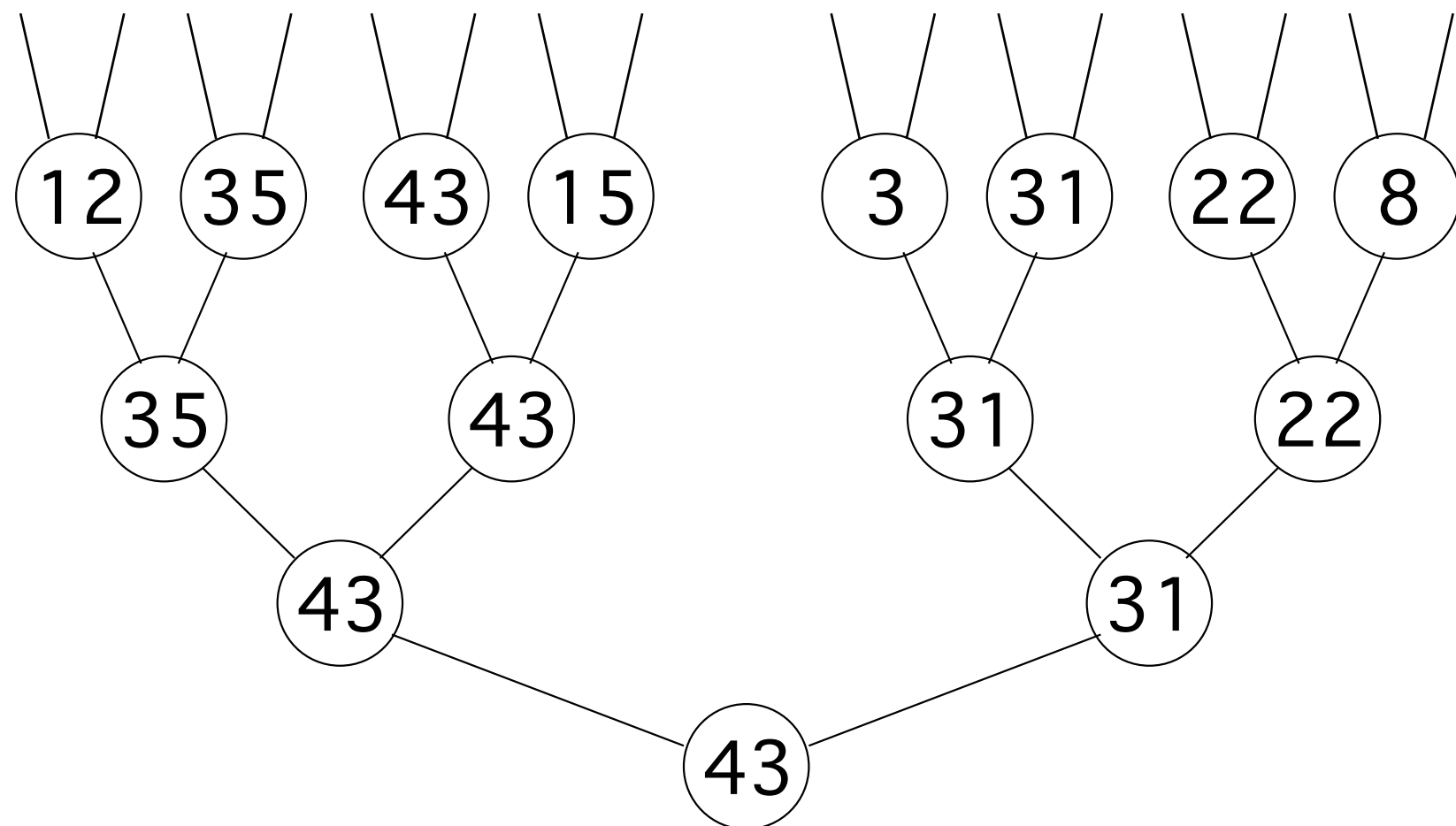
16

8

4

2

1





Divergent branching =

"if" statements:

Branches can be bad in GPU code!

Why?



Divergent branching in SIMD:

All branches execute *all code*! Data masked with result of "if".

Warp-level problem!

Can not be avoided within warps if a single thread gets a different result from others. Can be avoided if all threads in warp take same branch

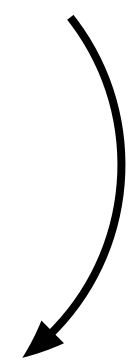


Divergent warp

```
if X then 10010110
|
|   and with 10010110
|
else
|
|   and with 01101001
|
endif
```

Non-divergent warp

```
if X then 11111111
|
|
|
else
|
|
|
endif
```



this episode





Lecture questions

- 1) In what way does bitonic merge sort fit the GPU better than many other sorting algorithms?
- 2) Bitonic merge sort is $N \log^2 N$ and QuickSort is $N \log N$. Why they will still have similar complexity on the GPU?
- 3) What is the reason to use pinned memory?
- 4) What problem does atomics solve?



More memory

Managed memory

Atomics

Pinned memory



Managed memory

Makes read/write memory as easy as constant!

New, simpler Hello World!

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void hello(char *a, int *b)
{
    a[threadIdx.x] += b[threadIdx.x];
}

__managed__ char a[N] = "Hello \0\0\0\0\0\0";
__managed__ int b[N] = {15, 10, 6, 0, -11, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int main()
{
    printf("%s", a);
    dim3 dimBlock( blocksize, 1 );
    dim3 dimGrid( 1, 1 );
    hello<<<dimGrid, dimBlock>>>(a, b);
    cudaDeviceSynchronize(); // Synchronize

    printf("%s\n", a);
    return EXIT_SUCCESS;
}
```



Managed memory

- Managed memory must be declared `__managed__`
- Memory accessible both from CPU and GPU. Risk for racing!
- Copy to GPU or copy to `__managed__`, same thing.
- Do not expect performance penalty (but always be ready for surprises).



Atomic operations

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

Available in CUDA from Compute model 1.1 (which means everywhere).

Specify compute model with

`-arch compute_11`

but you probably don't have to. (I didn't.)

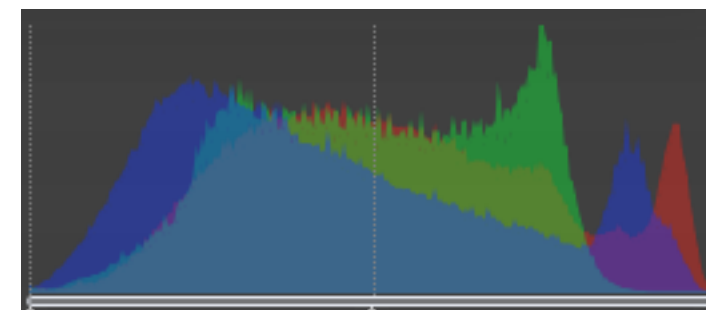
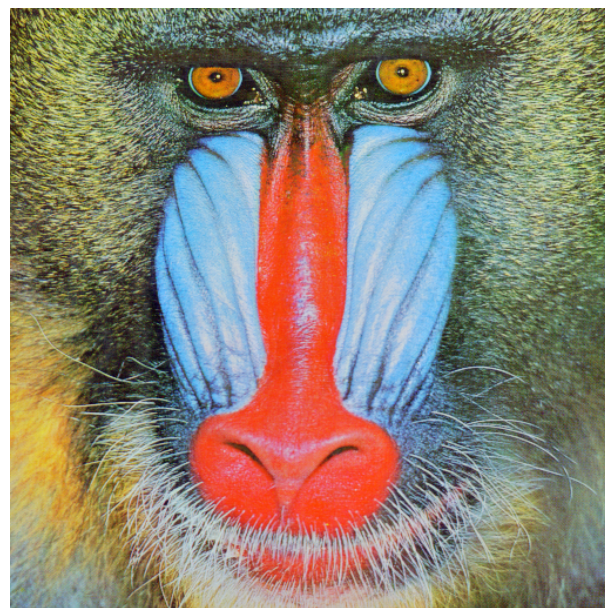


Example: Histogram

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 in parallel

Each thread reads $a[\text{threadIdx}]$

and perform $h[a[\text{threadIdx}]] = h[a[\text{threadIdx}]] + 1$

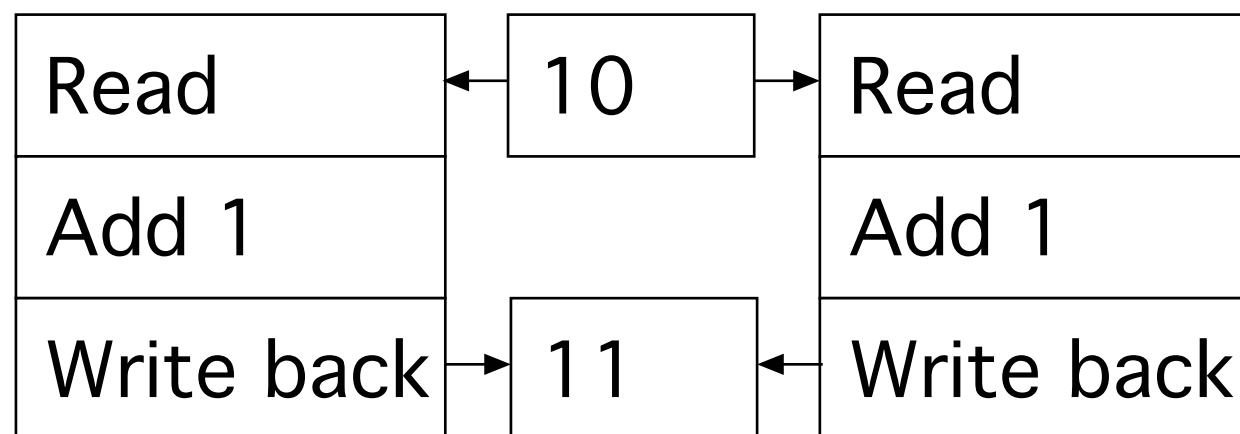
...not



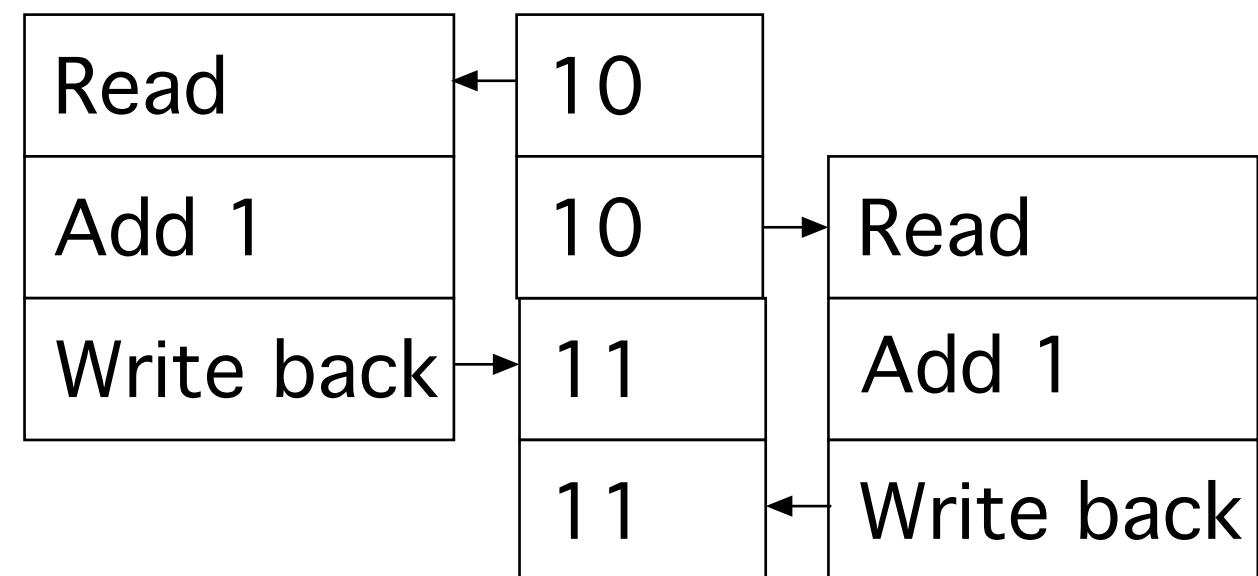
Histogram memory conflicts

If you try to parallelize these 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

Supported for both global and shared memory.



But it comes for a cost!

Slower than other operations

Simpler but slower than reduction solutions!



How would I do histograms?

Atomics are fairly OK... 256 lanes of parallelism.

Split to parts for separate blocks.

Produce one histogram for each part.

Merge result, possibly in several reduction steps.



Example: Find maximum

```
for all elements i in a[]  
maxValue = max(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!



More exotic optimizations and tools

Pinned memory

Multiple streams

Not where you start but let's not ignore the options.



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!

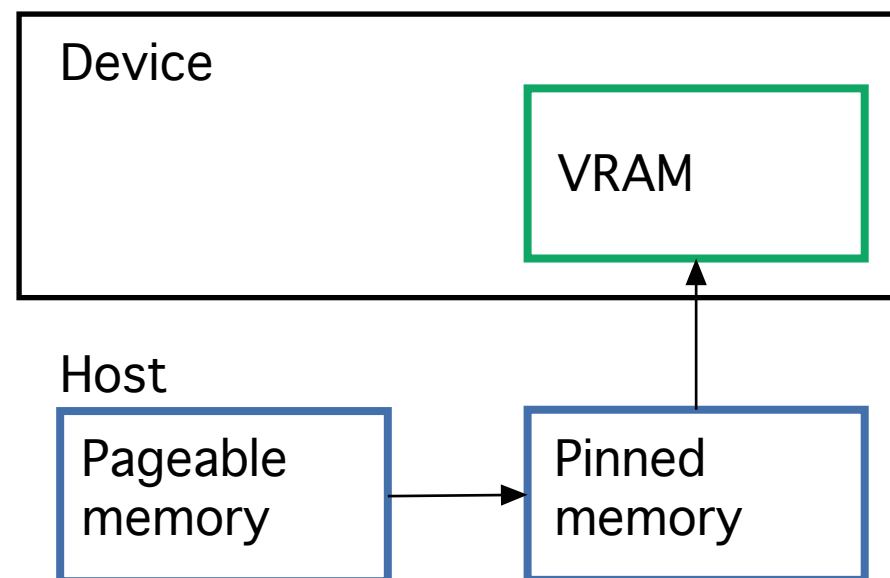


Pinned memory

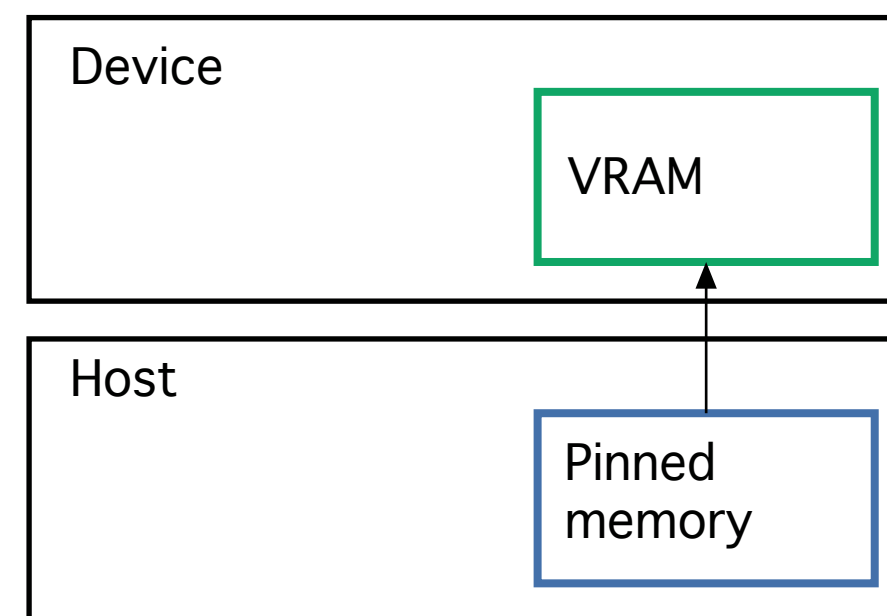
Page-locked memory is a limited resource!

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

Picture based on an NVidia article



Normal, pageable data transfer



Pinned data transfer



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, $\frac{2}{3}$ data transfer, $\frac{1}{3}$ 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?



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!

`cuda-gdb`

Variant of the GDB debugger

Allows breakpoints and single-stepping CUDA kernels!