

CUDA, Part Three

CS 3220 / CS 5220 Lecture 4-E

Jason Hibbeler

University of Vermont

Spring 2024

Topics

Events, and performance measurement

Device memory

Another 2-D problem

Atomic operations

Measuring Performance

As we develop, test, and modify CUDA code, we might want to ask whether a particular change was helpful

- what would “helpful” actually mean in this circumstance?
- one narrow way to answer the question would be: “did this change make my program run faster”?

With CPUs on multiuser system, it's hard to measure turnaround time (the time that a program takes to run)

- because of context switches
- and I/O latency

Measuring Performance

But on the GPU, the answer to “how long did my code take to run” is simpler

- the GPU is running code in a constrained environment
- it's not multitasking
- it's not doing I/O

On the GPU, we can measure throughput simply, by looking at elapsed time

- `stop_time - start_time`

Event

On the GPU, we can get the current time in two steps

1. create an event
2. record the event, which gives us a timestamp

```
cudaEvent_t start;  
cudaEventCreate(&start);  
cudaEventRecord(start, 0); // ignore the second parameter; use zero
```

Elapsed Time

And to get elapsed time, record two events:

```
cudaEvent_t start, stop;  
cudaEventCreate(&start);  
cudaEventCreate(&stop);  
cudaEventRecord(start, 0); // ignore the second parameter; use zero  
// now do some work on the GPU  
cudaEventRecord(stop, 0);
```

And then take the difference between the two timestamps

- it's actually a bit more complicated though

Elapsed Time

These calls are made outside of the kernel code

- but they're passed to the GPU
- they end up in a work queue on the GPU
- and the actual time that a work item in the work queue will be performed isn't predictable

Before, we were submitting only a single action to the GPU

- the call to the kernel code

But now, we'll be submitting several actions

- the event creates, the event records, and the kernel itself

Elapsed Time

Solution:

- synchronize on the stop timestamp event
- like this:

```
cudaEventRecord(stop, 0);  
cudaEventSynchronize(stop);
```

When we synchronize on an event, we know that all of the GPU work before the event has been completed

Elapsed Time

Finally, to get the actual elapsed time:

```
float elapsedTime;  
cudaEventElapsedTime(&elapsedTime, start, stop);
```

The resolution is OS dependent (for example, 0.5 ms)

And finally, to clean up:

```
cudaEventDestroy(start);  
cudaEventDestroy(stop);
```

Elapsed Time: Example

Here's what this looks like in a program:

```
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);

dotp<<<numBlocks,threadsPerBlock>>>( dev_U, dev_V, dev_partialSum, N);

cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);

float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
printf("elapsed time: %.4f ms\n", elapsedTime);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

Computing Elapsed Time

For head-to-head CPU vs. GPU comparisons:

- the fairest way to report how long it takes for a GPU algorithm to solve a problem is to include required memory transfers
- because after all, what we really care about is “how long did it take to solve my problem?”

But we can also cheat a little

- if we really want to know “how efficient is this algorithm on a GPU compared to the equivalent CPU algorithm”, then it's OK just to time the computational kernel (and not include the memory transfers)

note: the University requires me to tell you that not including the time of the memory transfers in the time we report for an algorithm is fudging a little

Memory

GPUs have enormous computational power

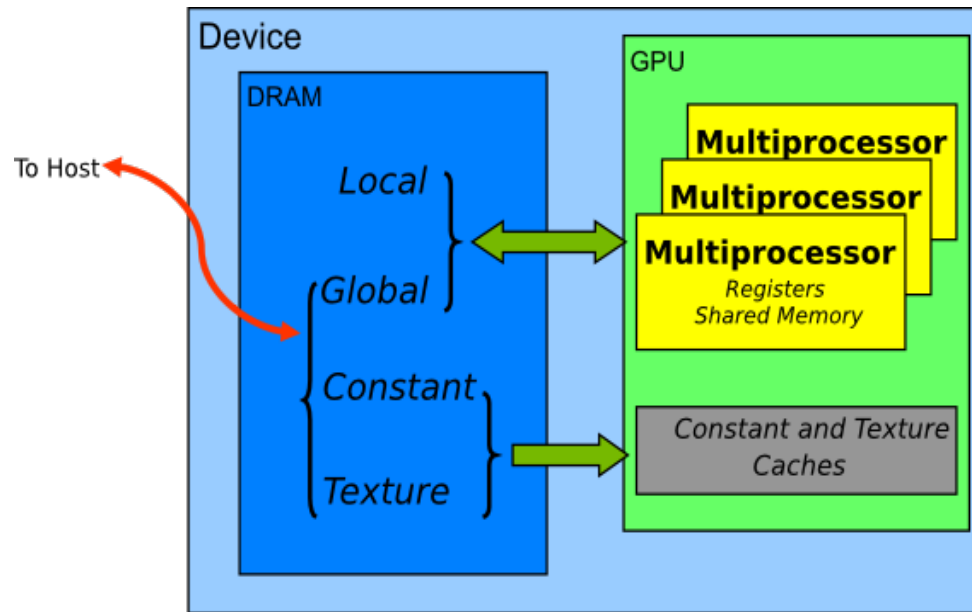
- with many hundreds of arithmetic logic units (ALUs) on each chip
- allowing the simultaneous processing of thousands of threads

The computation isn't the problem

- it's the memory bandwidth; getting data into and out of all of those ALUs

Device Memory

Memory spaces on the device (the GPU)



Local memory: contains the stack, local variables that cannot be held in registers, parameters, and return addresses for subroutines

Shared memory: a cache that can be partitioned as L1 + shared memory

Memory

The GPU does not use demand paging

- this means that every byte of virtual memory allocated to a thread must be backed by a byte of physical memory

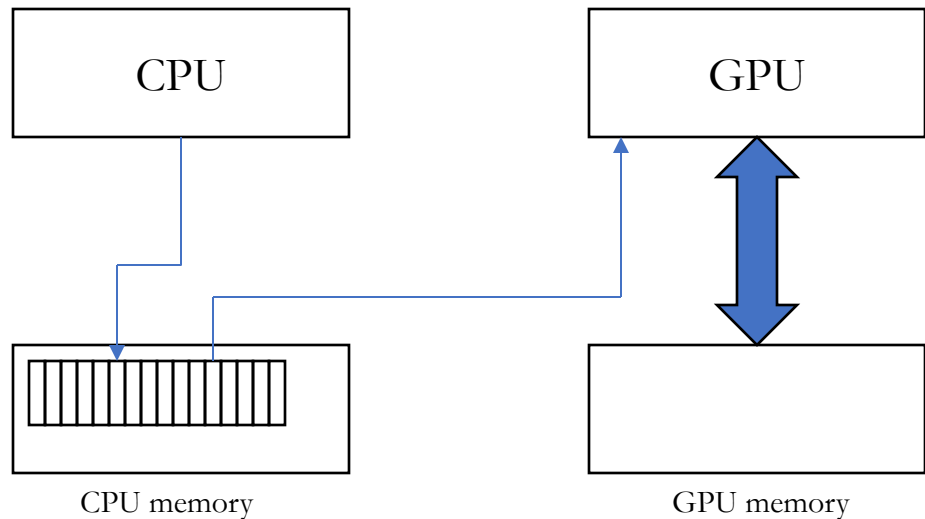
The GPU's memory is disjoint from the CPU's memory

- there is a way to map CPU memory to the GPU (“pinned memory”)
- this allows the GPU to read and write directly to the CPU's memory
- this can speed up data transfer between host and GPU

Memory: Command Buffer

Commands for the GPU are put into a shared buffer in the CPU's memory

- the GPU reads asynchronously from this buffer



Ideally, both GPU and CPU are kept busy

- one way to achieve this is to interleave memory copies and GPU execution
- a mechanism for doing this is *streams*

Newer GPUs have a *copy engine*

- can perform host/device memory transfers while the SMs are doing computations

GPU Architecture

Hardware architectures

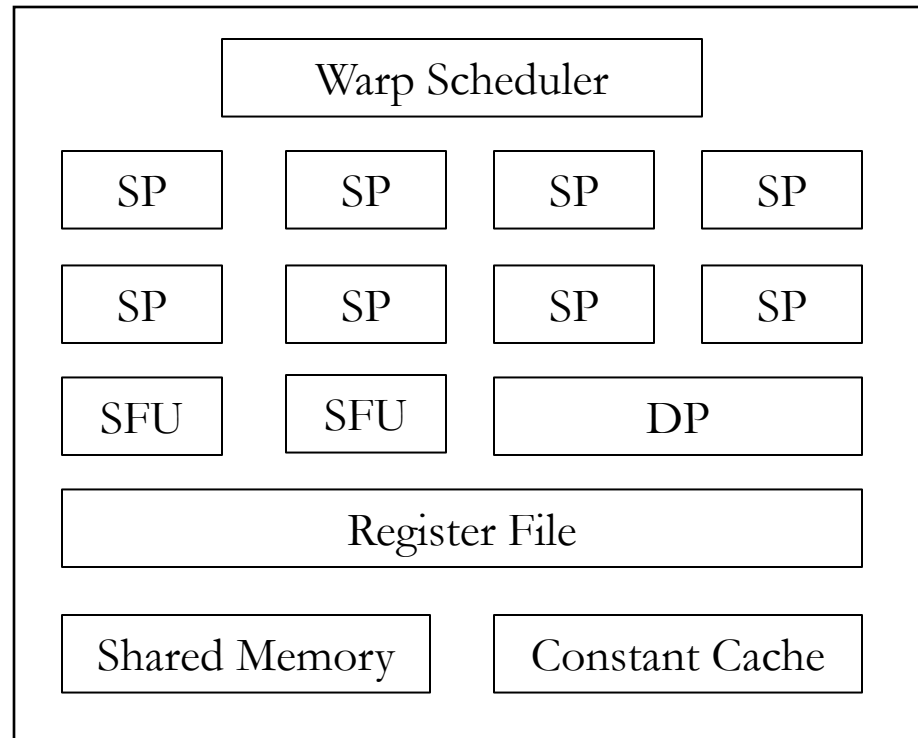
- Tesla: debuted in 2006, in the GeForce 8800 GTX (G80)
- Fermi: debuted in 2010, in the GeForce GTX 480 (GF100)
- Kepler: debuted in 2012, in the GeForce GTX 680 (GK104)
- Maxwell: debuted in 2014, in the GeForce GTX 750 (GM107)
- Pascal: debuted in 2016, in the GeForce GTX 10 series (GP100)
- Volta: debuted in 2017, in the V100 (GV100)
- Turing: debuted in 2018, in the GeForce RTX 20 series (TU102)
- Ampere: debuted in 2020, in the GeForce 30 series (GA100)

+ into the future (Nvidia is not standing still)

- Skynet: debuted in 2030, and the rest is history

Streaming Multiprocessor (SM)

SM from an older architecture (Tesla)



SP: single-precision functional unit

DP: double-precision functional unit

SFU: special-function unit (sqrt, log, sin, log, etc.)

Streaming Multiprocessor (SM)

Newer architectures have

- several warp schedulers per SM
- hundreds of SMs
- dozens of load/store units per SM

CUDA Software Architecture

The CUDA compiler, `nvcc`, extends a standard C/C++ compiler

- such as `gcc`

`nvcc` produces "ptx" code ("Parallel Thread eXecution")

- an intermediate representation of compiled GPU code
- ptx code is stable from architectural generation to generation
- but the underlying microcode might change

Example of ptx Code:

BB0_1:

```
mul.wide.s32 %rd5, %r10, 4;
add.s64 %rd6, %rd2, %rd5;
add.s64 %rd7, %rd1, %rd5;
ld.global.f32 %f1, [%rd7];
ld.global.f32 %f2, [%rd6];
add.f32 %f3, %f2, %f1;
st.global.f32 [%rd7], %f3;
add.s32 %r10, %r10, %r2;
setp.lt.s32 %p2, %r10, %r5;
@%p2 bra BB0_1;
```

__global__

```
void add(int n, float *x, float *y) {
    int index = blockIdx.x * blockDim.x + threadIdx.x;
    int stride = blockDim.x * gridDim.x;

    for (int i = index; i < n; i = i + stride)
        y[i] = x[i] + y[i];
}
```

It's sort of RISC-like, with vector operations

Device Memory

Salient features of device memory

Memory	Location on/off chip	Cached	Access	Scope	Lifetime
Register	on	N/A	R/W	1 thread	thread
Local	off	yes**	R/W	1 thread	thread
Shared	on	N/A	R/W	all threads in block	block
Global	off	*	R/W	all threads + host	host allocation
Constant	off	yes	R	all threads + host	host allocation
Texture	off	yes	R***	all threads + host	host allocation

* Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

** Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

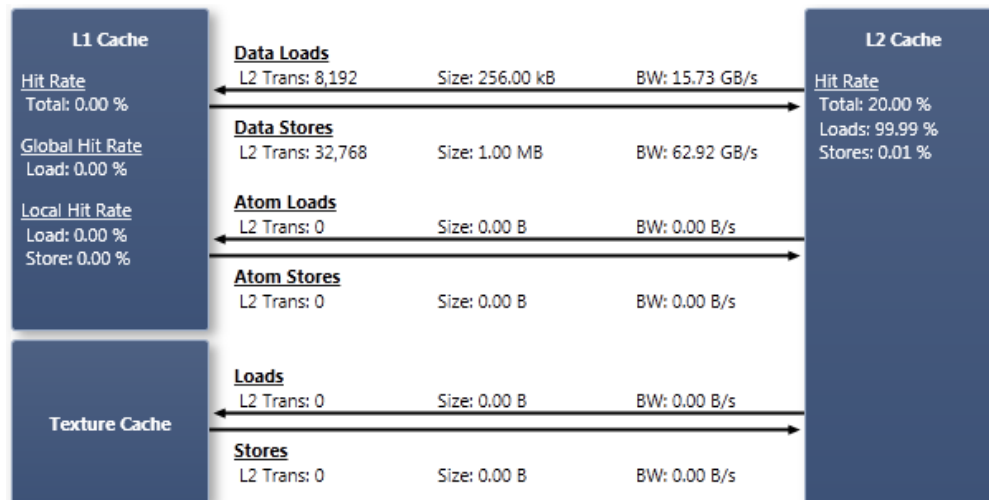
*** In the case of texture access, if a texture reference is bound to a linear array in global memory, then the device code can write to the underlying array.

Caches

SMs have an L1 and L2 cache

From Nvidia:

- Loads from the caches are made via transactions of a fixed size
- L1 transactions are 128 bytes, and L2 and texture transactions are 32 bytes



from [Nvidia docs](#)

Caches

Important strategy for optimizing memory usage:

- group loads and stores in order to access the necessary data in as few cache transactions as possible

For memory cached in both L1 and L2

- if every thread in a warp loads a 4-byte value from sparse locations which miss in the L1 cache
- then each thread will incur one 128-byte L1 transaction and four 32-byte L2 transactions
- this will cause the load instruction to reissue 32 times more than if the values would be adjacent and cache-aligned

If bandwidth between caches becomes a bottleneck:

- rearranging data or algorithms to access the data more uniformly can alleviate the problem

Coalescing

Access by the SMs to device global memory is expensive

Global memory loads and stores by threads of a warp are grouped by the device into as few transactions as possible (this is called “coalescing”)

- for best performance: ensure global memory accesses are coalesced whenever possible
- the access requirements for coalescing depend on the compute capability* of the device

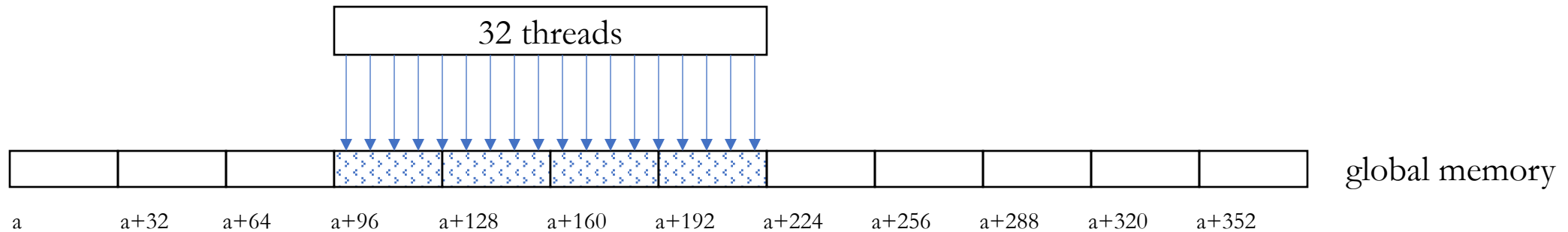
For devices of compute capability 6.0 or higher:

- the requirements can be summarized quite easily
- the concurrent accesses of the threads of a warp will coalesce into a number of transactions equal to the number of 32-byte transactions necessary to service all of the threads of the warp

*compute capability: roughly speaking, the hardware generation of the GPU; VACC GPUs have compute capability 7.x

Coalescing

Example: suppose the 32 threads of a warp access adjacent four-byte words (e.g., floats) in a memory region that is 32-byte aligned (so this is $4 \times 32 = 128$ bytes in total)



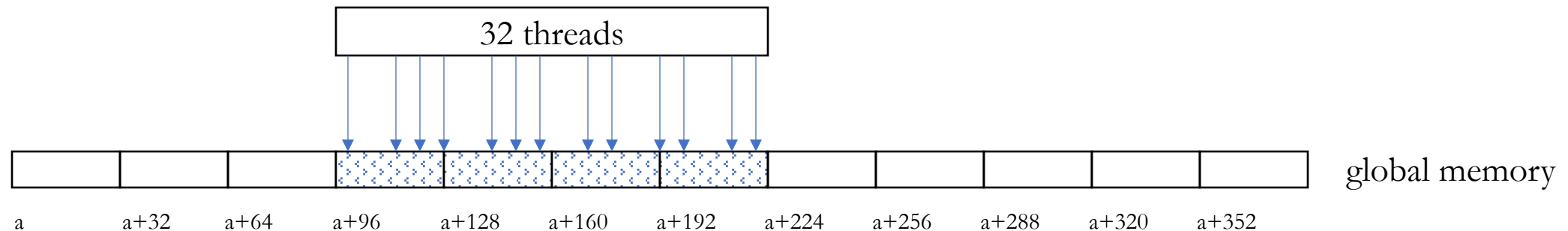
where a is a multiple of 32

This results in four 32-byte memory transactions

- and the accesses are fully coalesced: there are no unused bytes in the transactions

Coalescing

Suppose that only a subset of the threads of a warp access four-byte words in this 128-byte region



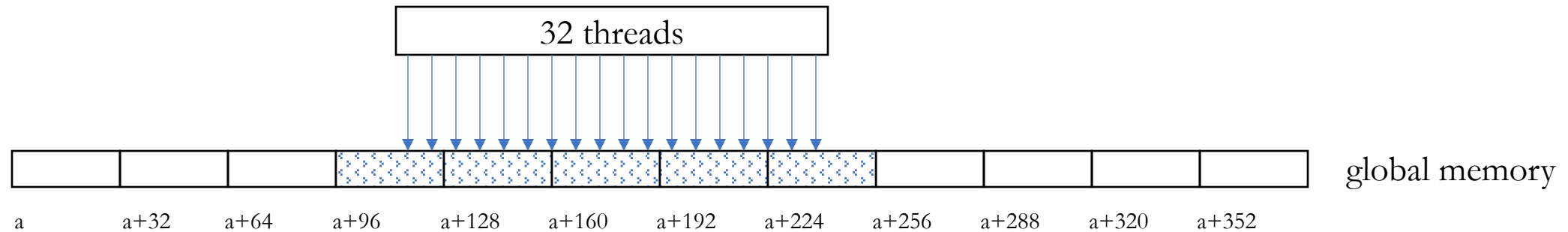
where a is a multiple of 32

This also results in four 32-byte memory transactions

- but the accesses are no longer coalesced: there are unused bytes in the transactions
- and the cost in bandwidth in a best-case scenario (according to an experiment that Nvidia shows) is $\sim 12\%$

Coalescing: Misaligned Access

If the threads of a warp access data that is not aligned on a 32-byte boundary:



where a is a multiple of 32

This results in five 32-byte memory transactions

- again, the accesses are no longer coalesced: there are unused bytes in the transactions

Coalescing

Achieve coalescing by choosing the thread-block size to be a multiple of 32

`cudaMalloc()` produces memory aligned on a 256-byte boundary

Memory Banks and Shared Memory

All threads in a thread block can access shared memory

- shared-memory access is an order of magnitude faster than global-memory access
- the shared memory is divided into equally-sized chunks called banks
- each bank can be accessed simultaneously
- so a memory load or store that accesses n banks can be performed simultaneously
- giving a bandwidth increase by the factor n over access to a single memory bank

So the key thing is understanding the structure of the memory banks

Memory Banks

On newer GPUs ($\geq 5.x$):

- each SM has by default 48 KB of shared memory
- successive 32-bit words are assigned to different banks: a hardware organization of memory
- and the memory bandwidth is 32 bits per clock cycle
- the warp size is 32 threads, and there are 32 banks

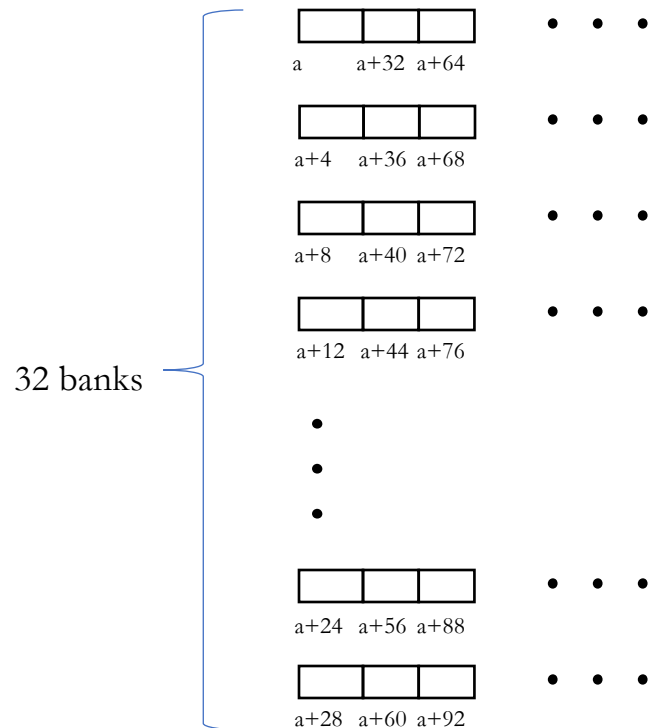
Bank conflict

- occurs when different threads in the same warp access different words that map to the same bank

Memory Banks

Successive 32-bit (four-byte) words are assigned to different banks

- and there are 32 banks



This then means:

- that if the 32 threads of a warp read 32 consecutive words
- then the reads can happen simultaneously

A computation having this structure would be:

- each thread `threadIdx.x` in a thread block accesses `A[threadIdx.x]`

CUDA Arrays

CUDA arrays are allocated in device global memory

- but they have a special (opaque) layout that is optimized for 2-D and 3-D locality
- this is different from the arrays that we used in the matrix-multiply example
- these are special data types supported in CUDA

Advice

- for applications having regular access patterns—especially those with little reuse—general memory is preferred
- but for applications with sparse access patterns—especially those with dimensional locality, such as computer vision—CUDA arrays provide benefit

Texture Memory

GPUs aren't used just for high-performance computing

- some people use them for high-performance graphics 😊

GPUs have special caching capabilities to treat part of the device's global memory as a “texture”

- i.e., something that can be rendered to a screen
- a texture is ultimately a 2-D array of values

The key aspects of this use of memory:

- it assumes a high degree of spatial locality—a thread will access memory near the memory that nearby threads access—but not necessarily "linear locality" in the way that the accesses to a 1-D array have locality
- and the access is read only

Texture Memory

Texture memory is global memory that is cached in special way

- to enable efficient read-only access to non-adjacent memory locations
- in computer graphics, textures involve wrapping a 2-D image around a 3-D object
- by doing a lot of lookups and bilinear interpolation

Also useful* for 2-D problems where a value is computed from north/south/east/west values

Technically, there is no “texture memory”

- it's device-global memory that can be efficiently cached

*in the old days

Texture Memory

Using Texture Memory

- again, “texture memory” is a misnomer
- it's really “texture-based caching of global memory”
- there are special CUDA functions to bind a texture reference to underlying device memory
- for example: `cudaBindTexture()` and `cudaBindTexture2D()`

Update

- newer versions of GPUs handle non-adjacent memory accesses much more efficiently than before
- one kind of computational problem (north/south/east/west) for which texture memory was traditionally recommended no longer needs it

Pinned Memory

On the host, it's possible to map device memory to the GPU

- in this case, the host memory must be “page-locked”, meaning that it cannot be paged out
- memory transfer from host to device is faster using this pinned host memory
- and the transfer from host to device can be asynchronous, since the host memory won't be paged out

Global Memory

Device global memory is attached directly to the GPU

- bandwidth is extremely high
- but not so high that we can ignore the latency
- this is the reason for understanding shared memory, memory banks, coalescing

Dynamic Allocation

Global memory is allocated through `cudaMalloc()` and `cudaFree()`

- memory allocation is expensive though

Pitched memory

- when allocating a 2-D array, pitched memory is allocated such that each row has the same alignment characteristics
- the pitch is the #bytes per row of the array
- pitch allocations take the width (in bytes) and height, pad the width to a suitable hardware-specific pitch, and pass back the base pointer and the pitch of the allocation


Pitched Allocation

For example, a 2-D array with a width of 352 bytes

- each row will be padded out to a multiple of some round number (512, when I tried this)
- this is the correct way to allocate a 2-D array

```
cudaMallocPitch( (void **) &d_arr, &d_pitch, 352, 24);
```

r o w	padding
r o w	padding
r o w	padding
r o w	padding


d_pitch

Kernel Execution

CUDA kernels are launched asynchronously

- and in particular, control returns to the CPU before the GPU has completed the requested operation

CUDA kernels do not have return values

- they report their results back via device memory—which must be copied back to the CPU explicitly
- or through mapped host memory

This also means that error handling and reporting is somewhat indirect

- if a kernel encounters an error, then the error is reported back to the host at some point
- the safest way to check for errors is to synchronize, using either `cudaDeviceSynchronize()` or a with a synchronous memcpy call

Streaming Multiprocessor (SM)

A computing core

Key hardware resources and characteristics

- thousands of registers
- L1 cache (in newer architectures)
- moderate clock frequency
- no branch prediction*
- integer, floating-point, and transcendental-function arithmetic units

*at one point; but again, Nvidia keeps developing and releasing new iterations of the hardware

Integer Operations: Sample

`__brev()`: bit reverse - reverses the order of bits in a word

`__popc()`: population count - returns the number of nonzero bits

`__funnelshift_l(hi, lo, sh)`: funnel shift - concatenates [hi:lo] into a 64-bit quantity, shifts it left by (sh&31) bits, and returns the most significant 32 bits

Floating-Point Operations

Hardware implementation of a full suite of floating-point (FP) operations

- including trig, log, sqrt, reciprocal
- and even specialized functions for financial computations such as derivatives pricing

Another Algorithmic Example: the Heat Equation

A standard two-dimensional partial differential equation

- special case of Laplace's Equation

In two dimensions:

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

Interpretation: the flow of heat at a point (x, y) is proportional to the difference between the temperature at (x, y) and the temperature of the material surrounding (x, y)

The Heat Equation

Easy to discretize and solve by finite differencing:

$$A_{i,j}^{n+1} = \frac{1}{4} [A_{i,j-1}^n + A_{i-1,j}^n + A_{i+1,j}^n + A_{i,j+1}^n - 4 A_{i,j}^n]$$

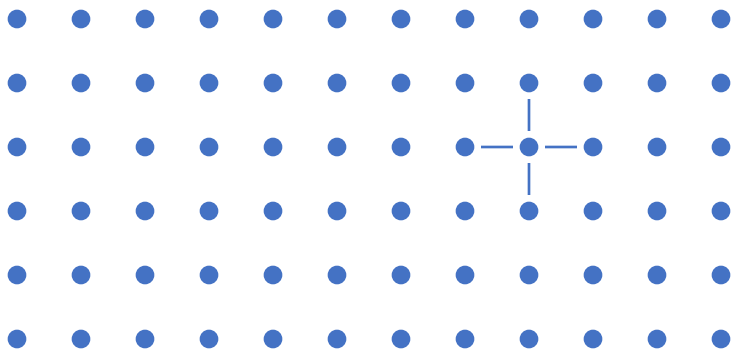
$A_{i,j}^n$: current values of A , the matrix that represents the values of the function $u(x, y)$ over a 2-D region

$A_{i,j}^{n+1}$: values of A at the next timestep

The Heat Equation

Example

- discretization of the 2-D heat equation on a finite $m \times n$ grid
- each $A_{i,j}$ is a value in a $m \times n$ array
- we can think of the array as a 2-D array; but ultimately it's laid out in a 1-D memory-address space
- the value of a grid point at the next time step is the weighted mean of the left, right, above, below neighboring values and the current value at that grid point (west/east/north/south)
- left and right values are adjacent in memory, but the above and below values are not!



$A_{i-1,j-1}$	$A_{i,j-1}$	$A_{i+1,j-1}$
$A_{i-1,j}$	$A_{i,j}$	$A_{i+1,j}$
$A_{i-1,j+1}$	$A_{i,j+1}$	$A_{i+1,j+1}$

$$A_{i,j}^{n+1} = \frac{1}{4} [A_{i,j-1}^n + A_{i-1,j}^n + A_{i+1,j}^n + A_{i,j+1}^n - 4 A_{i,j}^n]$$

The Heat Equation

And in fact there are many problems in which the new value of a mesh point is determined by a simple function of the values at neighboring mesh points

- image smoothing
- image edge detection
- finite differencing applied to various other PDEs

The Heat Equation

This will become an iterative update of an array T_{new} based on the values in an array T_{old}

$$T_{new} = T_{old} + \alpha(T_{old}^{north} + T_{old}^{south} + T_{old}^{east} + T_{old}^{west} - 4 T_{old})$$

	T_{old}^{north}	
T_{old}^{west}	T_{old}	T_{old}^{east}
	T_{old}^{south}	



	T_{new}	

and do this over each $T_{i,j}$ in a 2-D array
(with special conditions at the edges)

The Heat Equation

Example: hold the boundaries at a constant value, say zero

And hold certain interior points at a constant value, say c

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0																0
0																0
0																0
0																0
0																0
0							c	c	c							0
0							c	c	c							0
0																0
0																0
0																0
0																0
0																0
0																0
0																0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

The Heat Equation

Basic algorithm:

```
// allocate host memory, allocate device memory
// set host constant values: points in the mesh that are held at a
// fixed (const) temperature
// initialize grid #1 and grid #2 in device memory to all zeros
g = 0
for i = 1 to numIterations
    set the const values in grid #g
    update grid (g+1)%2 based on the values in grid g
    g = (g+1)%2
```

So in this way, we alternate between two grids (arrays)

- g is 0 and then 1 and then 0 and then 1 etc.

The Heat Equation

Like this:

```
__global__  
void( float *outSrc, const float * __restrict__ inSrc ) {  
    // some statements  
  
    outSrc[position] = inSrc[position] + alpha * ( inSrc[north] + inSrc[south] +  
        inSrc[east] + inSrc[west] - 4*inSrc[position] );  
}
```

And we'll make calls to the kernel in pairs

- switching the input and output arrays in each call

Heat Equation

The full kernel, to update an array of size $\text{DIM} \times \text{DIM}$

```
__global__
void( float *outSrc, const float * __restrict__ inSrc ) {
    int tidx = threadIdx.x + blockIdx.x * blockDim.x;
    int tidy = threadIdx.y + blockIdx.y * blockDim.y;
    int position = tidx + tidy*blockDim.x * gridDim.x;

    int west = position - 1;
    int east = position + 1;
    north = position - DIM;
    south = position + DIM;

    if (tidx > 0 && tidx < DIM-1 && tidy > 0 && tidy < DIM-1) {
        outSrc[position] = inSrc[position] + alpha * ( inSrc[north] + inSrc[south] +
                                                         inSrc[east] + inSrc[west] - 4*inSrc[position] );
    }
}
```

`const` and `__restrict__` give more information to the compiler about the use of this parameter and enable the compiler to optimize the use of this memory

Heat Equation

The full kernel, to update an array of size $\text{DIM} \times \text{DIM}$

```
__global__
void( float *outSrc, const float * __restrict__ inSrc ) {
    int tidx = threadIdx.x + blockIdx.x * blockDim.x;
    int tidy = threadIdx.y + blockIdx.y * blockDim.y;
    int position = tidx + tidy*blockDim.x * gridDim.x;

    int west = position - 1;
    int east = position + 1;
    north = position - DIM;
    south = position + DIM;

    if (tidx > 0 && tidx < DIM-1 && tidy > 0 && tidy < DIM-1) {
        outSrc[position] = inSrc[position] + alpha * ( inSrc[north] + inSrc[south] +
                                                         inSrc[east] + inSrc[west] - 4*inSrc[position]);
    }
}
```

these are not adjacent
memory accesses!

but with newer GPU
hardware, there's no
significant penalty

The Heat Equation

Here's the setup:

```
dim3 threads(16, 16);    // block size is 256 threads
dim3 blocks(N/16, M/16); // create enough blocks so that each block
                        // has 256 threads

updateKernel<<<blocks, threads>>>(d_mesh_1, d_mesh_2, d_pitch, M, N);
```

The Heat Equation

Memory allocation—of 2D arrays—is a little messier

- the call `cudaMallocPitch()` creates a row-major array in device memory with proper alignment for efficient access

```
size_t pitch;  
float *d_mesh_1, *d_mesh_2, *d_constVals;  
cudaMallocPitch( (void **) &d_mesh_1, &d_pitch, N*sizeof(float), M);  
cudaMallocPitch( (void **) &d_mesh_2, &d_pitch, N*sizeof(float), M);  
cudaMallocPitch( (void **) &d_constVals, &d_pitch, N*sizeof(float), M);
```

The Heat Equation

Copying a 2-D array from host memory to device memory:

```
cudaMemcpy2D(d_mesh_1, d_pitch, h_mesh_1, N*sizeof(float), N*sizeof(float), M,  
             cudaMemcpyHostToDevice);  
cudaMemcpy2D(d_mesh_2, d_pitch, h_mesh_2, N*sizeof(float), N*sizeof(float), M,  
             cudaMemcpyHostToDevice);  
cudaMemcpy2D(d_constVals, d_pitch, h_constVals, N*sizeof(float), N*sizeof(float), M,  
             cudaMemcpyHostToDevice);
```


Heat Equation

The full kernel, generalized to an $M \times N$ grid (M vertical elements, N horizontal elements)

```
__global__ void( float *outSrc, const float * __restrict__ inSrc, size_t pitch, int M, int N ) {
    int tidx = threadIdx.x + blockIdx.x * blockDim.x;
    int tidy = threadIdx.y + blockIdx.y * blockDim.y;
    int numEltsPerRow = d_pitch / sizeof(float); // this is #floats per padded row
    int pitchdiv = d_pitch / sizeof(float);
    int position = tidy * numEltsPerRow + tidx;

    int west = position - 1;
    int east = position + 1;
    north = position - pitchdiv;
    south = position + pitchdiv;

    if (tidx > 0 && tidx < N-1 && tidy > 0 && tidy < M-1) {
        outSrc[position] = inSrc[position] + alpha * ( inSrc[north] + inSrc[south] +
                                                         inSrc[east] + inSrc[west] - 4*inSrc[position] );
    }
}
```

On the CPU Side

The kernel runs repeatedly, with source and destination swapped each time

```
for (i=0; i<numIter; ++i) {
    setConstKernel<<<blocks, threads>>>(d_mesh_1, d_constVals, d_pitch, M, N);

    err = cudaGetLastError();
    if (err != cudaSuccess)
        break;

    updateKernel<<<blocks, threads>>>(d_mesh_1, d_mesh_2, d_pitch, M, N);

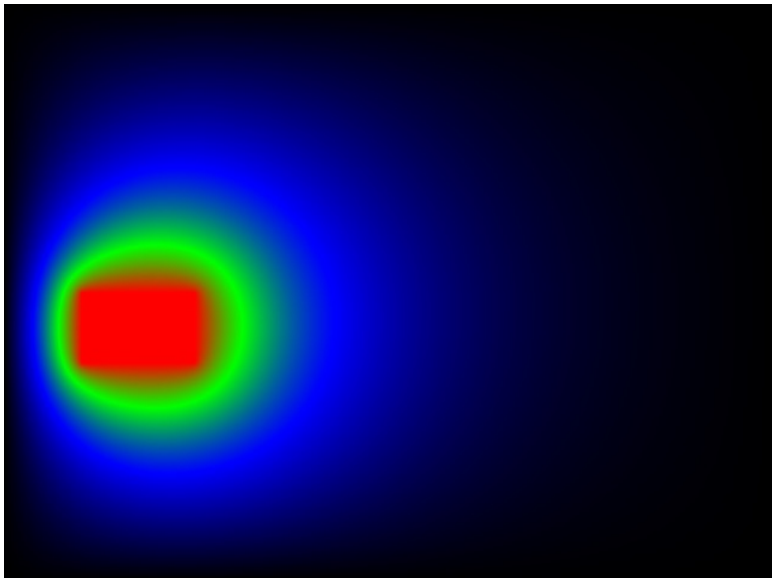
    err = cudaGetLastError();
    if (err != cudaSuccess)
        break;

    float *tmp = d_mesh_1;
    d_mesh_1 = d_mesh_2;
    d_mesh_2 = tmp;
}
```

Heat Equation

Convenient way to view results:

- write out a png of the values in the array after a certain number of iterations
- with colors to show scaled values from 0 to c
- here, this models a rectangular constant heat source with heat diffusing through the mesh, with the boundaries held at zero



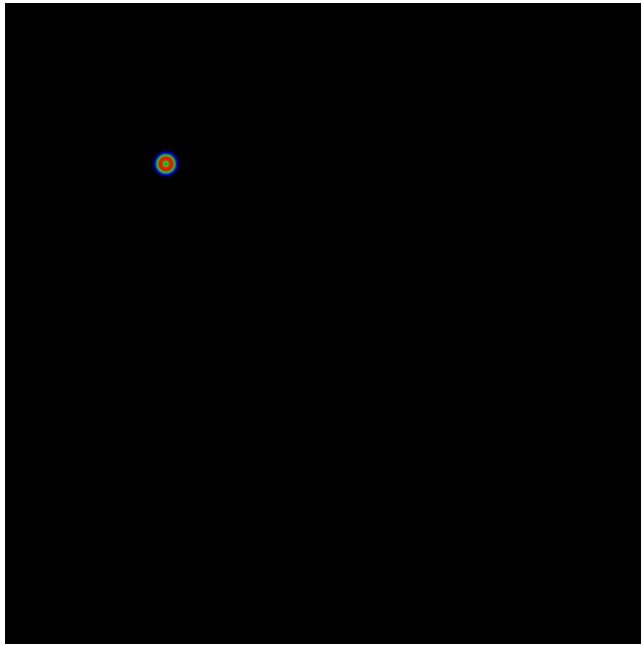
this is 100,000 iterations on a 480×640 mesh

- so the mesh has 307200 mesh points
- the calculation at each mesh point involves four additions and two multiplications
- so this represents something like 184,320,000,000 calculations
- and this took less than one second of GPU time

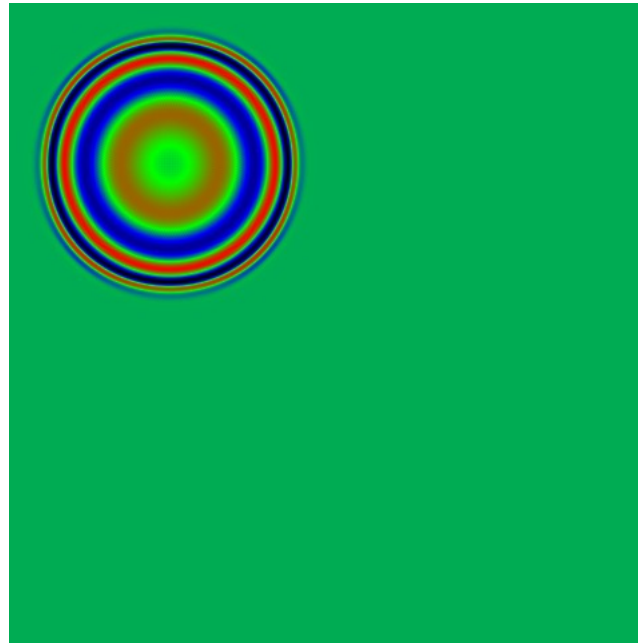
Wave Equation

Another simple linear 2-D partial differential equation

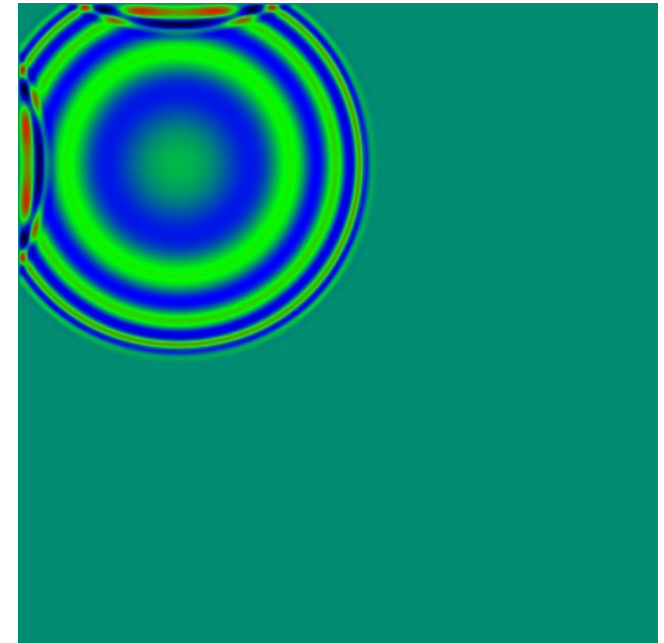
- easily solved by finite differencing; here's the solution on a 480×480 mesh



10 timesteps

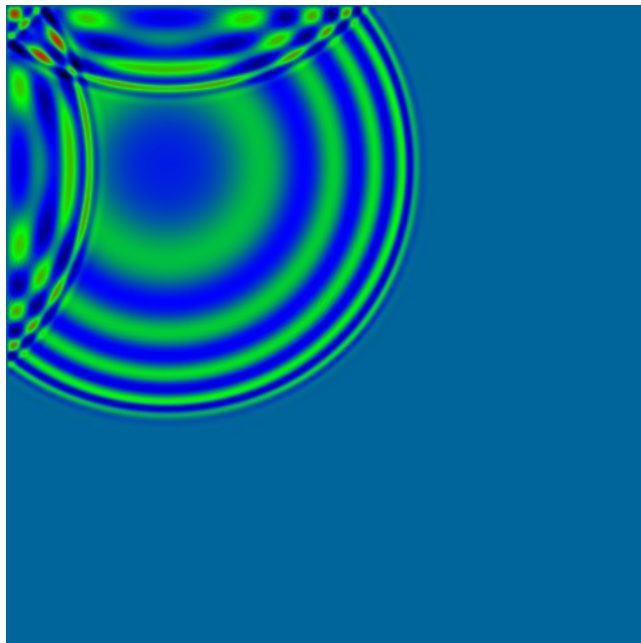


100 timesteps

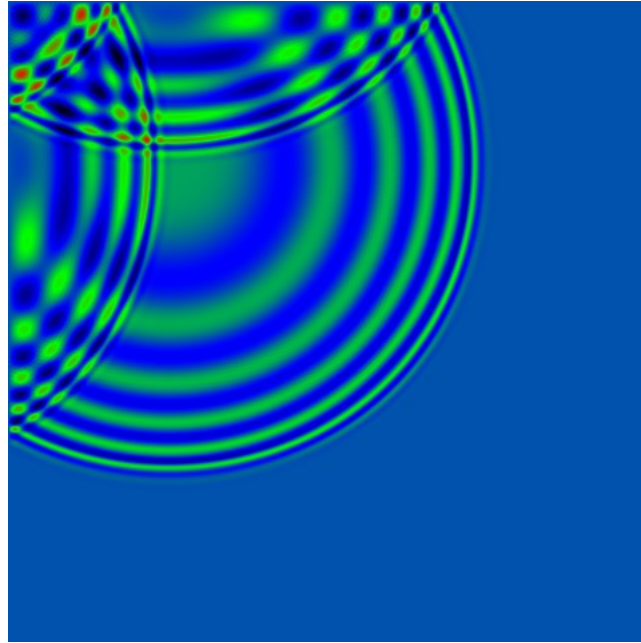


300 timesteps

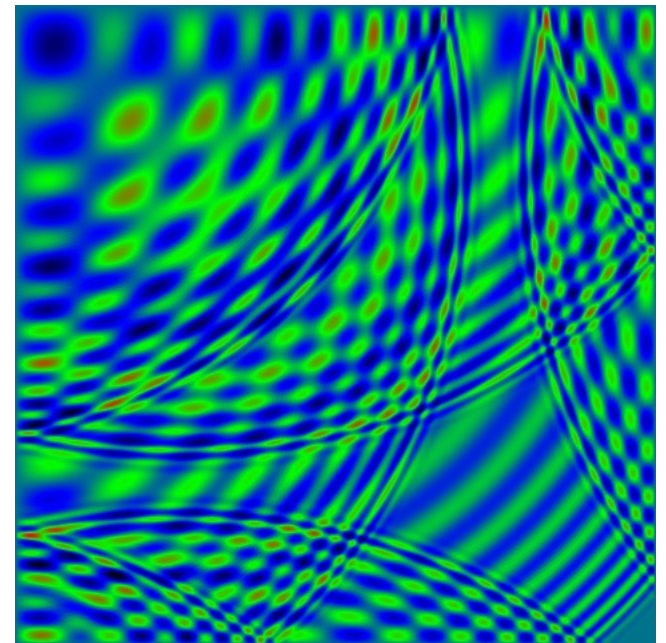
Wave Equation



400 timesteps

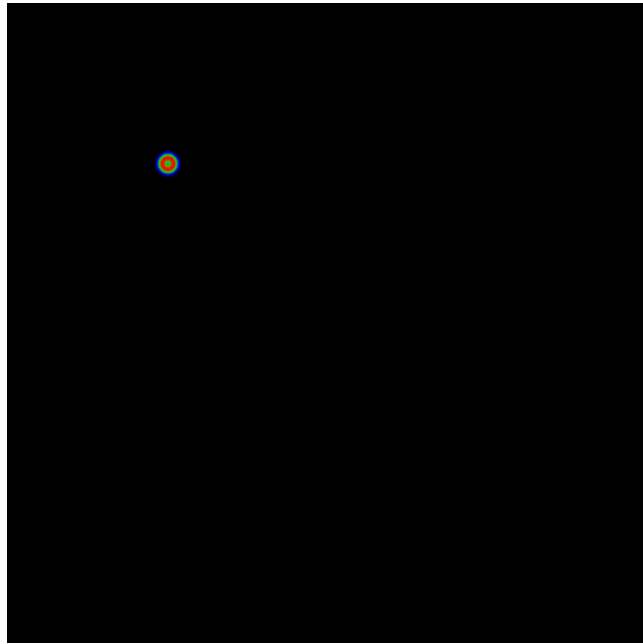


500 timesteps

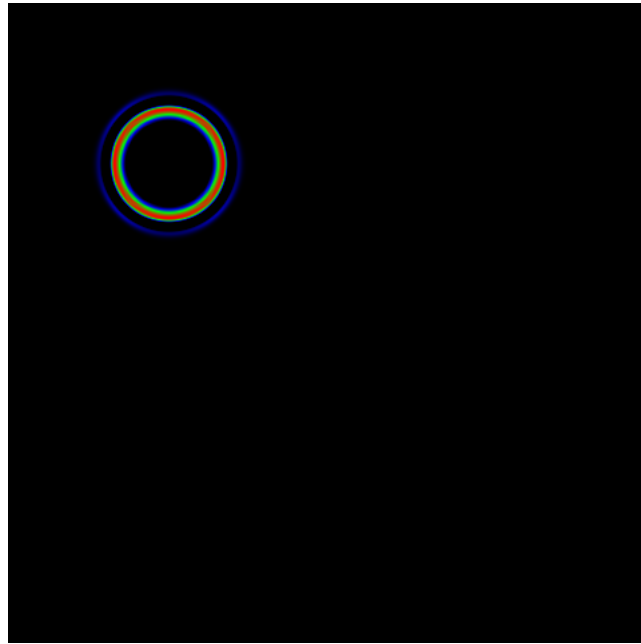


1000 timesteps

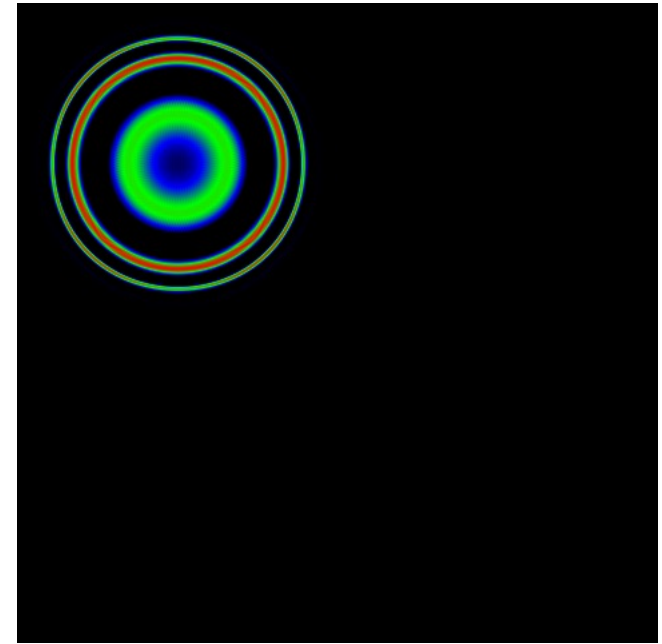
Wave Equation: Just the Positive Values



10 timesteps

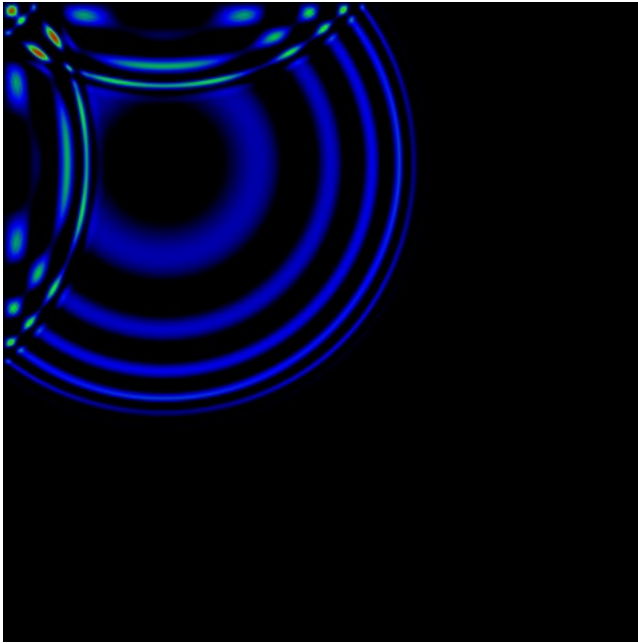


100 timesteps

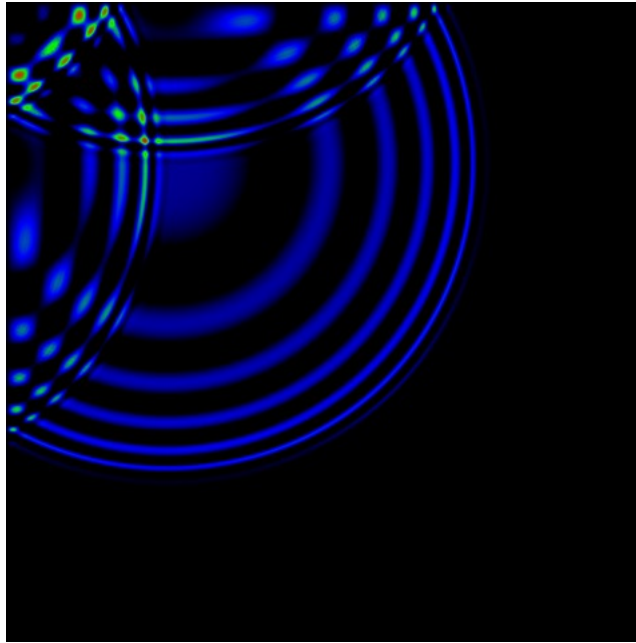


300 timesteps

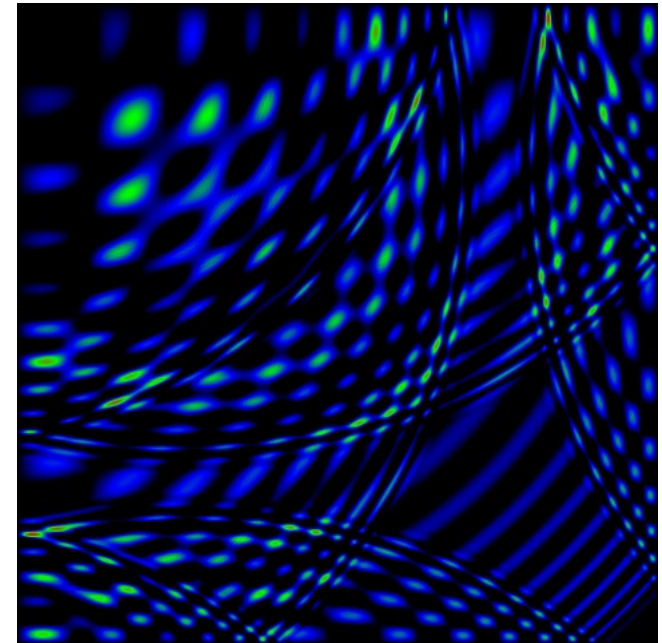
Wave Equation: Just the Positive Values



400 timesteps



500 timesteps



1000 timesteps

Constant Memory

For some computations, there might be data that doesn't need to change during the course of a kernel's execution

- an Nvidia GPU chip has a small amount of memory (64K) that can be accessed by all of the threads in a thread block on an SM
- this memory must be read only: this lets the SM cache it locally

However:

- access to different addresses in constant memory by different threads in a warp are serialized
- so constant memory works best when the threads in a warp access only a few distinct locations

And:

- if all threads in a warp access the same constant location, then the latency is the same as for a register access

Constant Memory

Suppose I have:

- a “room” of 256×256 grid points
- and a small number of objects in the room at specified grid points
- and for each grid point, I want to know “what’s my mean distance to all of the objects?”

Here, the objects are fixed

- put their coordinates in constant memory

Constant Memory

Declare constant memory in a kernel with `__constant__`

For example:

```
// x and y values for NUM_FIXED_OBJECTS points  
__constant__ float coefficients[2*NUM_FIXED_OBJECTS];
```

Constant Memory

Sample problem:

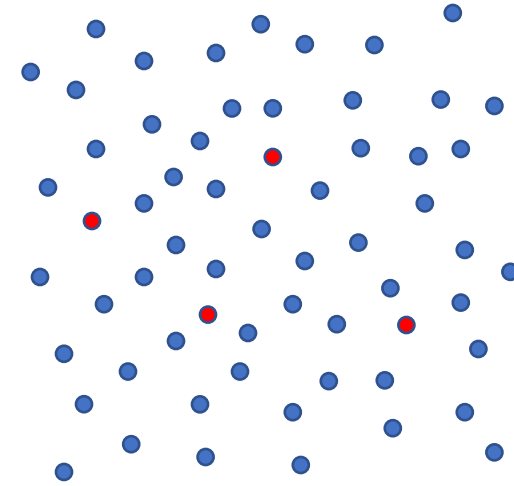
- $N \times N$ grid points (with floating-point x and y values)
- four fixed points

For each of the $N \times N$ grid points:

- compute the mean distance of that point to each of the fixed points

The fixed points have constant values for the problem (they're fixed!)

- so we can put them in constant memory



Constant Memory

Define constant memory this way:

```
__constant__ Object *d_objects[NUM_OBJECTS];
```

and copy to constant memory this way:

```
cudaMemcpyToSymbol( d_objects, h_objects, NUM_OBJECTS * sizeof(Object) );
```

Constant Memory: Performance

NOT including the memory copies

#grid points	#static objects	Elapsed time
512 x 512	16	CPU: 45 ms
512 x 512	16	GPU, non-constant objects: 0.097 ms
512 x 512	16	GPU, constant objects: 0.075 ms

including the memory copies

#grid points	#static objects	Elapsed time
512 x 512	16	CPU: 45 ms
512 x 512	16	GPU, non-constant objects: 4.75 ms
512 x 512	16	GPU, constant objects: 4.42 ms

each mem copy takes about 2 ms

Constant Memory: Performance

This requires:

- $512 \times 512 \times 16 \times 2$ floating-point additions: $(x_i - x_j)$ and $(y_i - y_j)$
- $512 \times 512 \times 16 \times 2$ floating-point products: $(x_i - x_j)^2$ and $(y_i - y_j)^2$
- $512 \times 512 \times 16$ floating-point additions: $(x_i - x_j)^2 + (y_i - y_j)^2$
- $512 \times 512 \times 16$ square-root operations: $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
- $512 \times 512 \times 16$ additional adds
- 512×512 divisions

$$512 \times 512 \times 16 = 4194304$$

Histograms

Histogram:

- a count of how many times a particular value appears in a set of values
- it's the first step of several algorithms

We'll do the following problem:

- given an array of length N containing integers in the range 0 to $m - 1$
- how many times does each value $0, 1, 2, \dots, m - 1$ appear?
- so in this case, the histogram is an integer array of size m
- each entry `array[i]` will then be the number of times that i appears in the big array

Atomic Operations

The rule: if more than a single thread will be writing to the same location:

- then the write operation must be done atomically

CUDA provides a function to do this

- `atomicAdd()`

Histograms

With CUDA:

- this is easily parallelizable: have one CUDA thread look at each array element
- but the tricky thing is that each CUDA thread needs to increment an entry in the histogram array
- and since many threads will be doing this simultaneously, we need to use an atomic addition

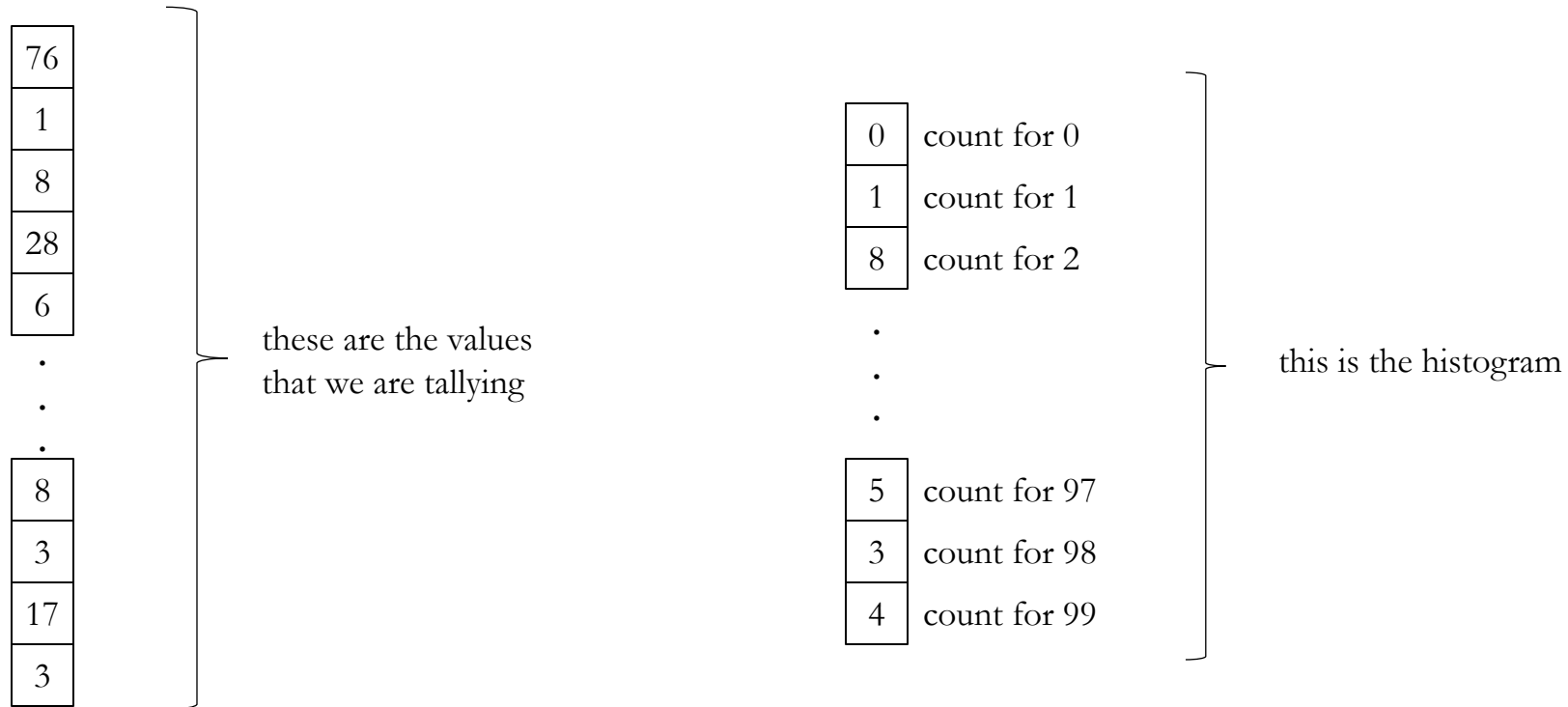
Atomic operation:

- implies locking (for mutual exclusion)
- if not done carefully, we can have **lock contention**: excessive time spent waiting for a lock because of the number of threads that want it and the frequency with which they want it
- rather than doing actual work

Structure of the Histogram

Looks at each element and increment the appropriate tally

- here, $m = 100$



Histogram: CPU Version

Here's the simple loop:

```
// initialize counts
for (k=0; k<m; ++k)
    h_hist[k] = 0;

// do the actual counting
for (i=0; i<N; ++i)
    h_hist[h_values[i]] = h_hist[h_values[i]] + 1;
```

No atomic operation needed, since this is a serial computation

Histogram: CUDA Kernel

```
__global__  
void histKernel(int n, int *d_values, int *d_histogram) {  
    int i, stride;  
  
    i = threadIdx.x + blockIdx.x * blockDim.x;  
    stride = blockDim.x * gridDim.x  
  
    while (i < n) {  
        atomicAdd( &(d_histogram[d_values[i]]), 1 );  
        i = i + stride;  
    }  
}
```

Lock Contention

Each thread on the GPU will execute this statement:

```
atomicAdd( &(d_histogram[d_values[i]]), 1 );
```

The `atomicAdd()` operation has a lock under the hood

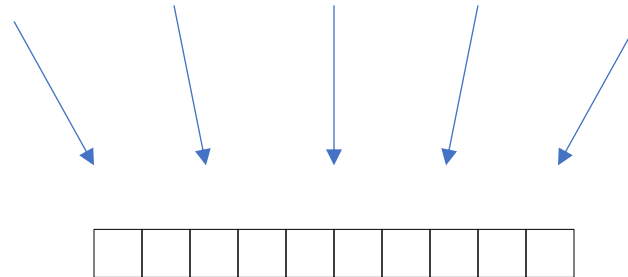
- if one thread is currently executing `atomicAdd()` on a particular memory location, then other threads that want to access that memory location must wait on the lock
- in this case, thousands of threads will be hitting a small number of memory locations (assuming m is small)
- which means there will be tremendous lock contention

Lock Contention

Suppose we have a huge array, and each element is an integer in the range 0 to 9

- each thread looks at one element
- and increments the appropriate entry in the global histogram
- result: huge degree of lock contention!

2	3	8	3	5	2	0	3	0	9	4	5	8	1	3	3	2	6	1	4	3	2	1	2	4	1	3	2	0	7	3	1	2	4	3	2	5	6	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



this is the histogram

Reducing Lock Contention

Simplifying assumption

- suppose m is 256
- and the block dimension is also 256 (although, in general, it could be bigger)

Now, have each thread block write to shared local storage

- each thread block keeps a local mini-histogram, just of the values in its region of the array

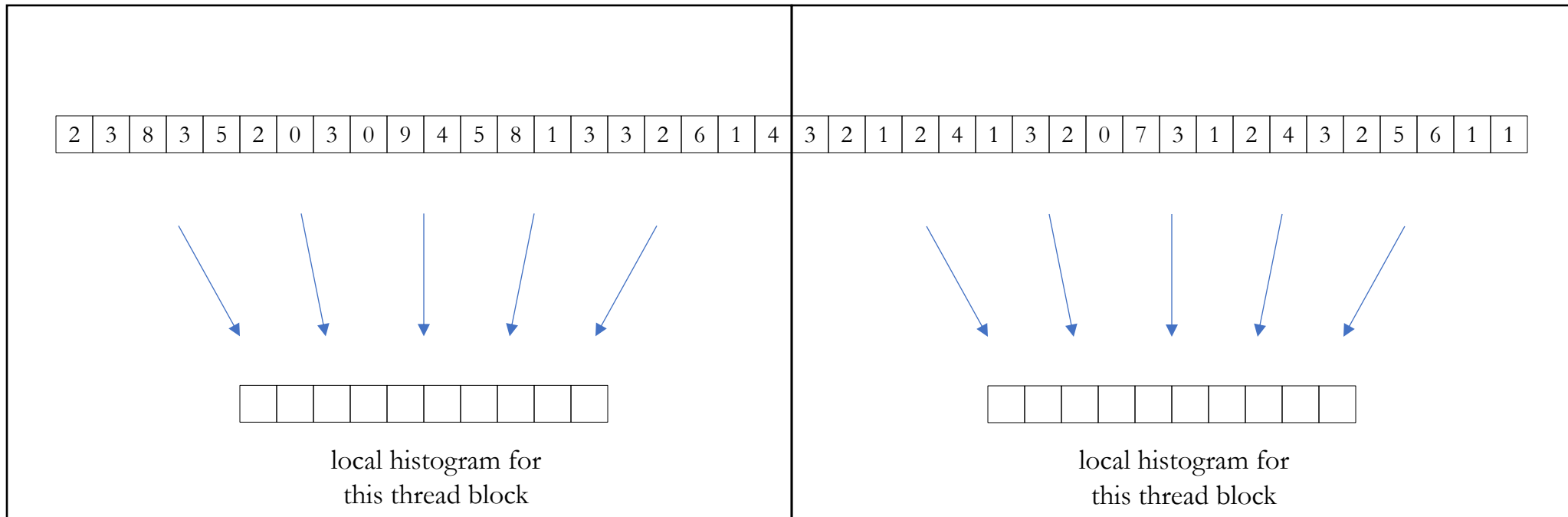
Then, each thread block will merge its results into the global histogram

- but there are far fewer thread blocks than threads
- so the degree of contention will be much smaller

Reducing Lock Contention

Create a local shared histogram for each thread block

- reduced the degree of lock contention greatly
- at the end, we'll merge all of the local histograms



Modified Histogram Kernel

```
__global__
void histKernelBetter(int n, int *d_values, int *d_histogram) {
    __shared__ int localhist[m]; // m is the range of the values in the array
    int i, stride;

    localhist[threadIdx.x] = 0; // each thread initializes its count to zero
    __syncthreads();

    i = threadIdx.x + blockIdx.x * blockDim.x;
    stride = blockDim.x * gridDim.x; // this is the total number of threads we have

    while (i < n) {
        atomicAdd( &(amp;localhist[d_values[i]]), 1 );
        i = i + stride;
    }

    __syncthreads();
    // now accumulate the results from all threads in this thread block
    atomicAdd( &(d_histogram[threadIdx.x]), localhist[threadIdx.x] );
}
```

Modified Histogram Kernel

Each thread block has 256 threads

- thread #0: looks at `values[0]`; updates `localHist[values[0]]` atomically
- thread #1: looks at `values[1]`; updates `localHist[values[1]]` atomically
- ...
- thread #255 looks at `values[255]`; updates `localHist[values[255]]` atomically

So in the worst case, there will be 256-way contention for a lock

- by the 256 threads in a thread block

Modified Histogram Kernel

But this is way better than lock contention among $256 \times b$ threads

- where b is the number of thread blocks

And then the last step involves worst-case contention among the thread blocks

- thread #0 in each thread block updates `histogram[0]` atomically
- thread #1 in each thread block updates `histogram[1]` atomically
- ...
- thread #255 in each thread block updates `histogram[255]` atomically

So again, the worst-case lock contention now involves b threads

Performance Results

#elements	Elapsed time
$256 \times 256 \times 256 \times 8 = 134217728$	CPU: 607 ms
$256 \times 256 \times 256 \times 8 = 134217728$	GPU, first implementation: 53.2 ms
$256 \times 256 \times 256 \times 8 = 134217728$	GPU, second implementation: 8.61 ms

×

Furthermore, by modifying the number of blocks, we can get even better performance:

- #blocks = 160: 1.66 ms
- #blocks = 320: 1.09 ms
- #blocks = 640: 0.86 ms (with memory copies, ~180 ms)

Conclusions

1. GPUs can provide immense speedup—many orders of magnitude
2. In order to use a GPU effectively, it's necessary to know that structure of the problem
3. In order to use a GPU effectively, it's necessary to understand how GPUs use memory
4. In order to use a GPU effectively, it's almost always necessary to write a specialized algorithm

References

[The CUDA Programming Guide](#)

[Nvidia's intro to CUDA tutorial](#)

[In-depth discussion of parallel reduction](#)

[Nvidia's CUDA Best Practices Guide](#)

CUDA by Example, by J Sanders and E Kandrot. Addison Wesley, 2011

The CUDA Handbook, by N. Wilt, Addison Wesley, 2013