Lecture 11

More CUDA

In this episode...

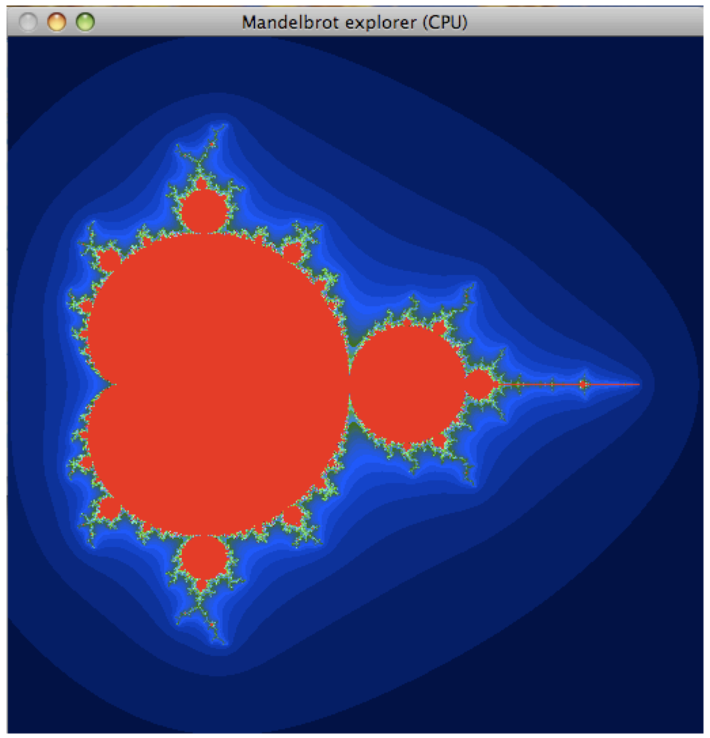
- Error checking
- Query device capabilities
- CUDA events
- More on CUDA memory:

Coalescing, Constant memory, Texture memory...

Lab 4

This week!

”Mandelbrot revisited” part, to follow up lab 1.



The story so far...

- CUDA and its language extensions
- The CUDA architecture
- Intro to memory
- Matrix multiplication example, using shared memory

CUDA and its language extensions

Kernel invocation myKernel<<<>>>()

__global__ __device__ __host__

cudaMalloc(), cudaMemcpy()

threadIdx, blockIdx, blockDim, gridDim

Using nvcc

The CUDA architecture

Blocks and threads

Grid-block-thread hierarchy

Indexing data with thread/block numbers

Intro to memory

global memory

shared memory

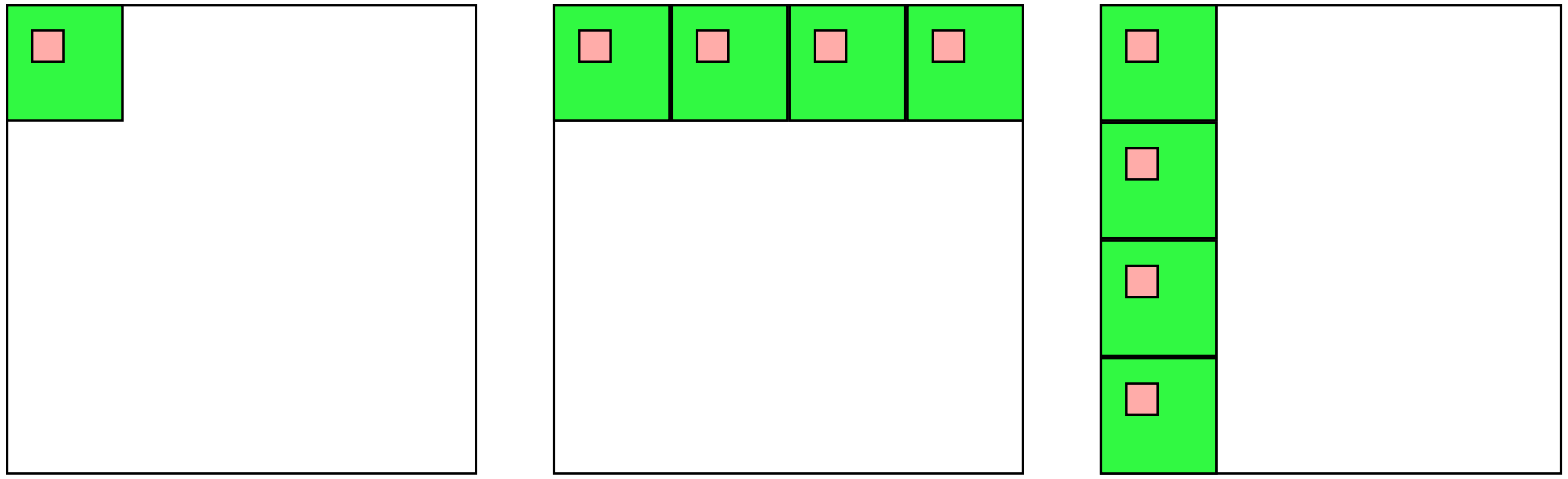
constant memory

local memory

texture memory/texture units

register memory

Matrix multiplication example, using shared memory



Huge speedup - my GPU went from questionable performance to clearly faster than CPU!

Over to today's e

Lecture questions:

1. Why can using constant memory improve performance?

2. What is CUDA Events used for?

3. What does coalescing mean and what should we do to get a speedup from coalescing?

4. Why can we not synchronize between blocks?

Error checking

- Functions returns error codes (but kernel launch does not)
- cudaGetLastError()
- cudaPeekLastError()
- cudaGetErrorString()

Asynchronous error checking

Asynchronous errors can not be returned by the function call!

Call cudaDeviceSynchronize() and check the latest error code.

Demo hello-error.cu

Extended "Hello World" with

- **cudaDeviceSynchronize()**
- **cudaGetLastError()**
- **cudaGetErrorString()**

and intentionally correct dimensions.

Some synchronization from last time:

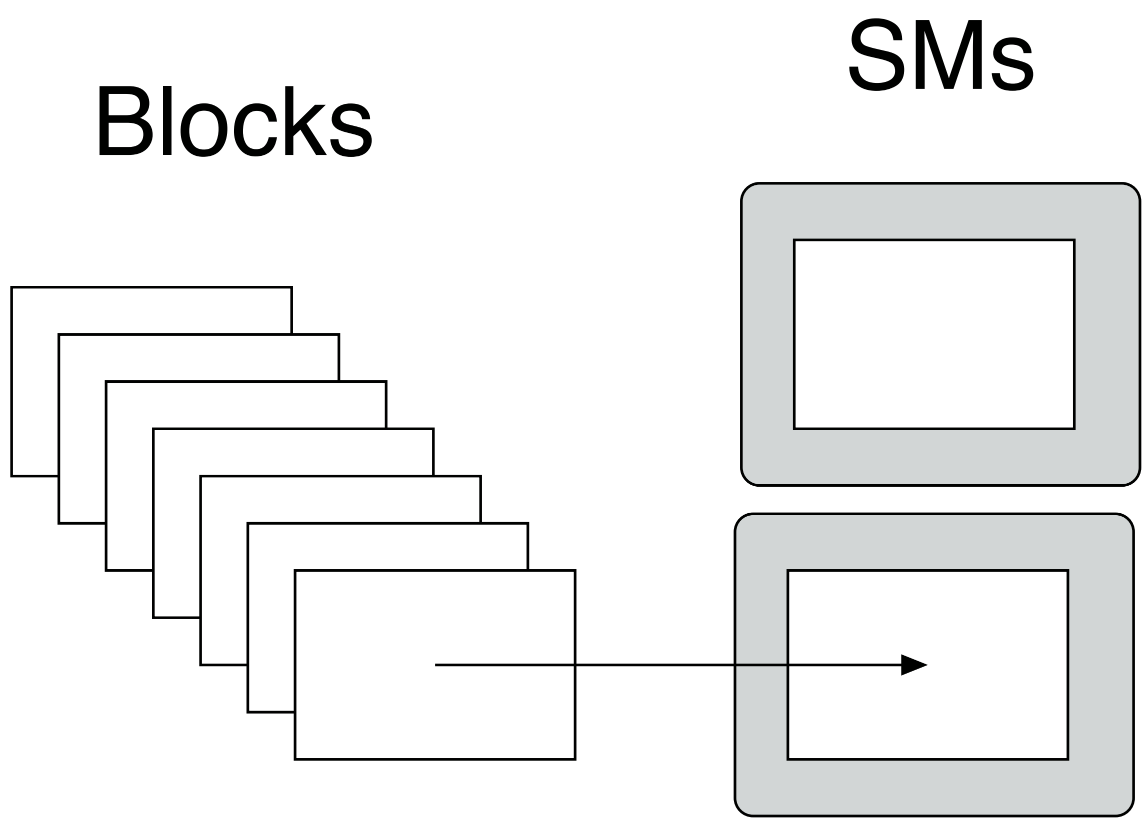
__syncthreads() is used inside a kernel.
Stop thread until all threads *in the block* reach the location!

cudaDeviceSynchronize() is used from the host. Wait until all current kernels finish.

cudaStreamSynchronize() waits until all kernels in a *stream* (group of kernels) finish.

No synchronization between blocks!

Why no synchronization between blocks?



Queue of blocks, one SM at a time.

More blocks than SMs!

Query devices

You can't trust all devices to have the same - or even similar - properties.

New boards may have totally different properties.

Query CUDA for a list of features using cudaGetDeviceProperties()

Example query result (9400M)

```
---- Information for GeForce 9400M ----
Compute capability:  1.1
Total global memory (VRAM):  259712 kB
Total constant Mem:  64 kB
Number of SMs:  2
Shared mem per SM:  16 kB
Registers per SM:  8192
Threads in warp:  32
Max threads per block:  512
Max thread dimensions:  (512, 512, 64)
Max grid dimensions:  (65535, 65535, 1)
```

Example query result 2 (GT 750M)

```
---- Information for GeForce GT 750M ----
Compute capability:  3.0
Total global memory/VRAM:  2096704 kB
Total constant Mem:  64 kB
Number of Streaming Multiprocessors (SM):  2
Shared mem per SM:  48 kB
Registers per SM:  65536
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (2147483647, 65535, 65535)
```

What is important?

Compute capability - can this board at all work with our program?

Amount of shared memory - make sure we fit.

Max threads, max dimensions - make sure we fit.

Threads in warp: If you optimize on warp level.

Number of SMs: Lower bound for blocks

Compute capability

Essentially CUDA/architecture version number.

1.0: Original release.
1.1: Mapped memory, atomic operations.
1.3: Double support.
2.0: Fermi.
3.0: Kepler.
5.0: Maxwell.
6.0: Pascal. ← Olympen
7.5: Turing. ← Asgård
8.6 Ampère.

<table>
<tr><th>Feature Support</th><th colspan="6">Compute Capability</th></tr>
<tr><th>(Unlisted features are supported for all compute capabilities)</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x, 3.0</th><th>3.5</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)</td><td rowspan="2">No</td><td colspan="5" rowspan="2">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)</td><td colspan="2" rowspan="4">No</td><td colspan="4" rowspan="4">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)</td></tr>
<tr><td>Warp vote functions (Warp Vote Functions)</td></tr>
<tr><td>Double-precision floating-point numbers</td><td colspan="3">No</td><td colspan="3">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)</td><td colspan="4" rowspan="7">No</td><td colspan="2" rowspan="7">Yes</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())</td></tr>
<tr><td>__ballot() (Warp Vote Functions)</td></tr>
<tr><td>__threadfence_system() (Memory Fence Functions)</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
__syncthreads_or() (Synchronization Functions)</td></tr>
<tr><td>Surface functions (Surface Functions)</td></tr>
<tr><td>3D grid of thread blocks</td></tr>
<tr><td>Funnel shift (see reference manual)</td><td colspan="5">No</td><td>Yes</td></tr>
</table>

LiTH

More features of interest:

3.5: Dynamic parallelism

5.3: Half precision float

7.x: Tensor cores

	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110	Maxwell	Pascal	Turing
Compute Capability	2.0	2.1	3.0	3.5	5.0	6.0	7.5
Threads / Warp	32	32	32	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64	?	?	?
Max Threads / Multiprocessor	1536	1536	2048	2048	2048	2048	1024
Max Thread Blocks / Multiprocessor	8	8	16	16			
32-bit Registers / Multiprocessor	32768	32768	65536	65536	64k*	64k	64k
Max Registers / Thread	63	63	63	255	255	255	255
Max Threads / Thread Block	1024	1024	1024	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16K 48K	16K 48K	16K 32K 48K	16K 32K 48K			
Max X Grid Dimension	2^16-1	2^16-1	2^32-1	2^32-1			
Hyper-Q	No	No	No	Yes			
Dynamic Parallelism	No	No	No	Yes			

Compute Capability of Fermi and Kepler GPUs

Compute Capability	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**	**2.1**	**3.0**	**3.5**
SM Version	sm_10	sm_11	sm_12	sm_13	sm_20	sm_21	sm_30	sm_35
Threads / Warp	32	32	32	32	32	32	32	32
Warps / Multiprocessor	24	24	32	32	48	48	64	64
Threads / Multiprocessor	768	768	1024	1024	1536	1536	2048	2048
Thread Blocks / Multiprocessor	8	8	8	8	8	8	16	16
Max Shared Memory / Multiprocessor (bytes)	16384	16384	16384	16384	49152	49152	49152	49152
Register File Size	8192	8192	16384	16384	32768	32768	65536	65536
Register Allocation Unit Size	256	256	512	512	64	64	256	256
Allocation Granularity	block	block	block	block	warp	warp	warp	warp
Max Registers / Thread	124	124	124	124	63	63	63	255
Shared Memory Allocation Unit Size	512	512	512	512	128	128	256	256
Warp allocation granularity	2	2	2	2	2	2	4	4
Max Thread Block Size	512	512	512	512	1024	1024	1024	1024
Shared Memory Size Configurations (bytes)	16384	16384	16384	16384	49152	49152	49152	49152
[note: default at top of list]					16384	16384	16384	16384
							32768	32768
Warp register allocation granularities					64	64	256	256
[note: default at top of list]					128	128		

Table 14. Technical Specifications per Compute Capability

	Compute Capability												
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**	**7.5**	8.6
Warp size	32												~~32~~
Maximum number of resident blocks per multiprocessor	16				32							16	
Maximum number of resident warps per multiprocessor	64											32	
Maximum number of resident threads per multiprocessor	2048											1024	1536
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K								
Maximum number of 32-bit registers per thread block	64 K	32 K	64 K				32 K	64 K		32 K	64 K		
Maximum number of 32-bit registers per thread	63	255											
Maximum amount of shared memory per multiprocessor	48 KB			112 KB	64 KB	96 KB	64 KB		96 KB	64 KB	96 KB	64 KB	
Maximum amount of shared memory per thread block [27]	48 KB										96 KB	64 KB	96k
Number of shared memory banks	32												
Amount of local memory per thread	512 KB												
Constant memory size	64 KB												
Cache working set per multiprocessor for constant memory	8 KB							4 KB	8 KB				
Cache working set per multiprocessor for texture memory	Between 12 KB and 48 KB							Between 24 KB and 48 KB			32 - 128 KB	32 or 64 KB	

Do I care about Compute capability?

While learning CUDA - not much. Stick to the basics, it works on all.

But if you write professional CUDA code, of course.

CUDA Events

Timing!

Two ways of timing CUDA programs:

- **CPU timer. Synchronize at start and end.**
- **CUDA Events. Synchronize at end.**

Synchronize? Because CUDA runs asynchronously.

CUDA Events API

cudaEventCreate - initialize an event variable

cudaEventRecord - place a marker in the queue

cudaEventSynchronize - wait until all markers have received values

cudaEventElapsedTime - get the time difference between two events

CUDA memory

Coalescing

Constant memory

Texture memory

Pinned memory

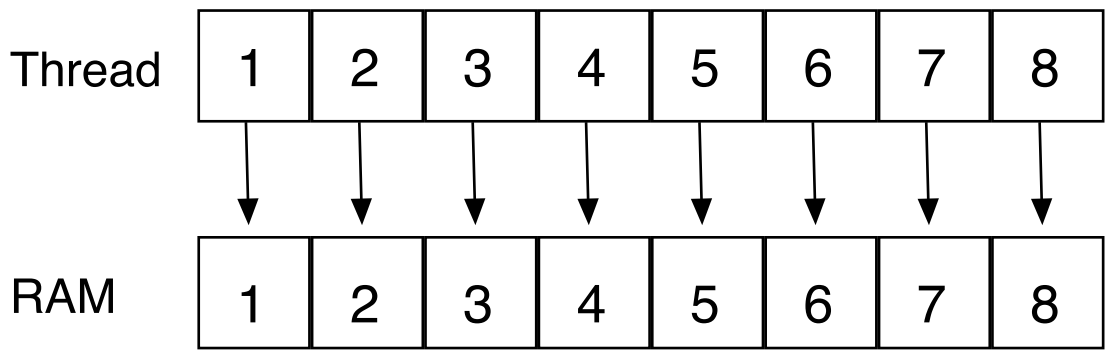
We already know…

- Global memory is slow.
- Shared memory is fast and can be used as ”manual cache”
- There were some other kinds of memory...

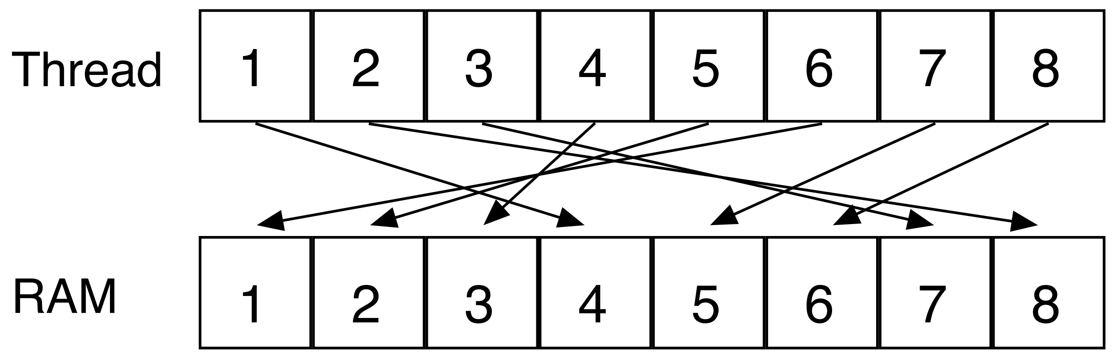
Coalescing

Always access global memory ”in order”

If threads access global memory in order of thread numbers, performance will be improved!



Good!



Bad!

WTF?

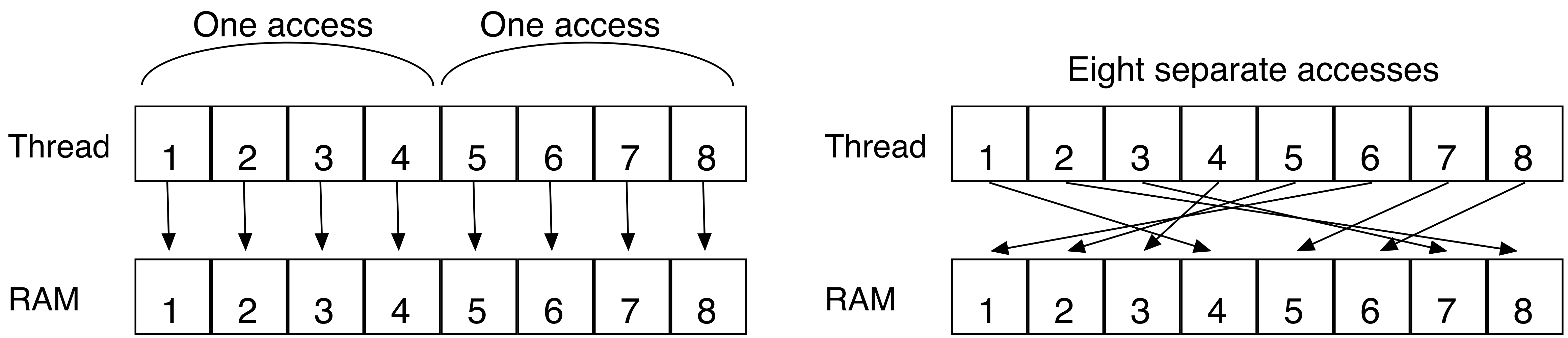
How can performance depend on what order I access my data??? Isn't it "random access"?

Yes... You can access in any order you want, but ordered access *helps* the GPU to read more data in one access!

Why? Because the GPU can get much data in a single transaction, and neighbor threads are tested for accessing the same area!

Coalescing

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



Good! **Bad!**

Coalescing on Fermi & later

Effect reduced by caches - but not removed.

Coalescing is still needed for maximum performance.

"A very important performance consideration... is the coalescing of global memory accesses." (CUDA C Best Practices Guide 2022)

Accelerating by coalescing

Pure memory transfers can be significantly faster by taking advantage of memory coalescing!

Example: Matrix transpose

No computations!

Only memory accesses.

Matrix transpose

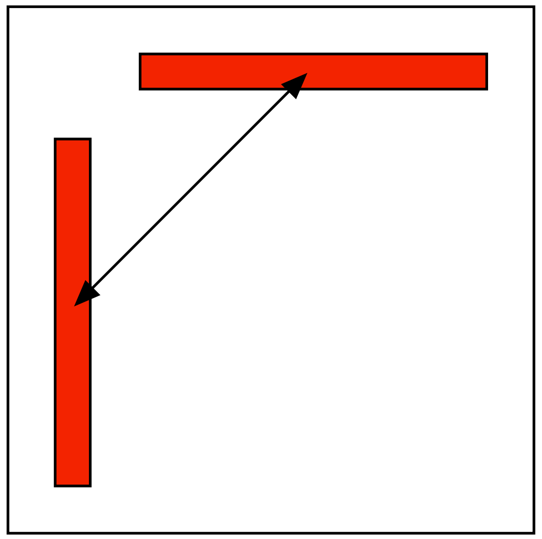
Naive implementation

```
__global__ void transpose_naive(float *odata, float* idata, int width, int height)
{
   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
   unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

   if (xIndex < width && yIndex < height)
   {
       unsigned int index_in  = xIndex + width * yIndex;
       unsigned int index_out = yIndex + height * xIndex;
       odata[index_out] = idata[index_in];
   }
}
```

How can this be bad?

Matrix transpose

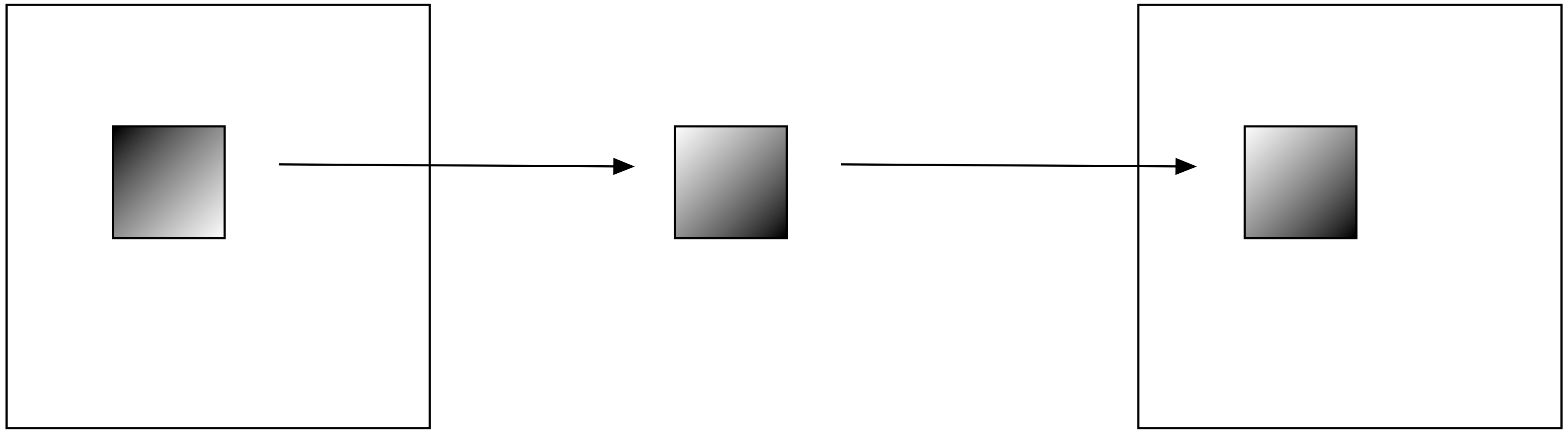
Coalescing problems



Row-by-row and column-by-column.
Column accesses non-coalesced!

Matrix transpose

Coalescing solution



Read from global memory to shared memory

In order from global, any order to shared

Write to global memory

In order write to global, any order from shared

Better CUDA matrix transpose kernel

```
__global__ void transpose(float *odata, float *idata, int width, int height)
{
 __shared__ float block[BLOCK_DIM][BLOCK_DIM+1];

 // read the matrix tile into shared memory
 unsigned int xIndex = blockIdx.x * BLOCK_DIM + threadIdx.x;
 unsigned int yIndex = blockIdx.y * BLOCK_DIM + threadIdx.y;
 if((xIndex < width) && (yIndex < height))
 {
  unsigned int index_in = yIndex * width + xIndex;
  block[threadIdx.y][threadIdx.x] = idata[index_in];
 }

 __syncthreads();

 // write the transposed matrix tile to global memory
 xIndex = blockIdx.y * BLOCK_DIM + threadIdx.x;
 yIndex = blockIdx.x * BLOCK_DIM + threadIdx.y;
 if((xIndex < height) && (yIndex < width))
 {
  unsigned int index_out = yIndex * height + xIndex;
  odata[index_out] = block[threadIdx.x][threadIdx.y];
 }
}
```

Shared memory for temporary storage

Read data to temporary buffer

Write data to global memory

Varying results

My demos tend to give varied results on my laptop GPU.
Yes, I am still searching...

Overall, I get

- **usually some speedup for coalescing**
- **no noticable speedup from avoiding bank conflicts**

Cache effect?

Let's try in the lab on full-scale GPUs!

Coalescing rules of thumb

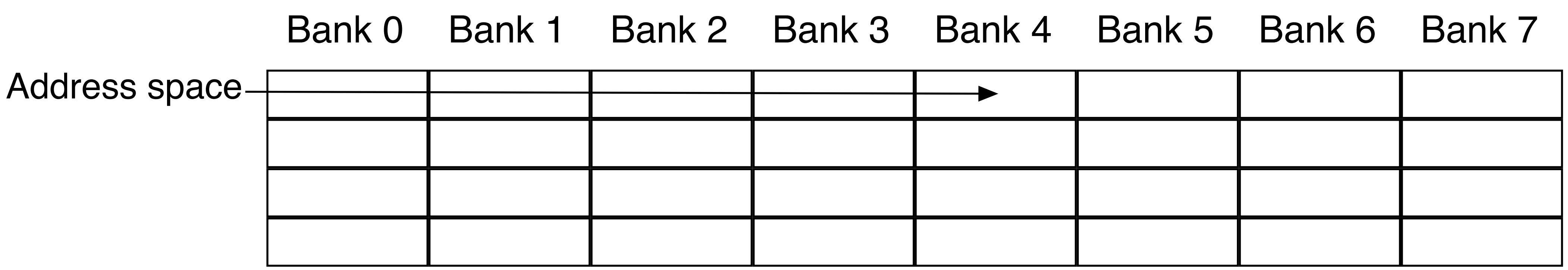
- The data block should start on a multiple of 64
- It should be accessed in order (by thread number)
- It is allowed to have threads skipping their item
- Data should be in blocks of 4, 8 or 16 bytes

Shared memory

Split into multiple memory banks (32). Fastest if you access different banks with each thread

Interleaved, 32 bits chunks

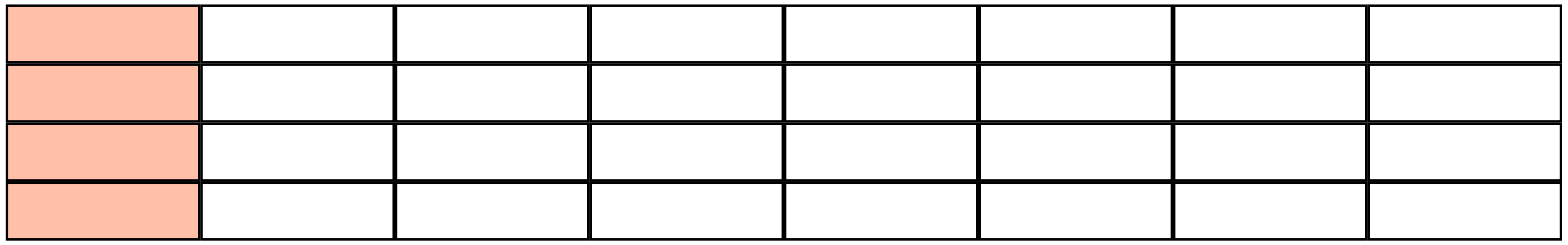
Thus: Address in 32-bit steps between threads for best performance



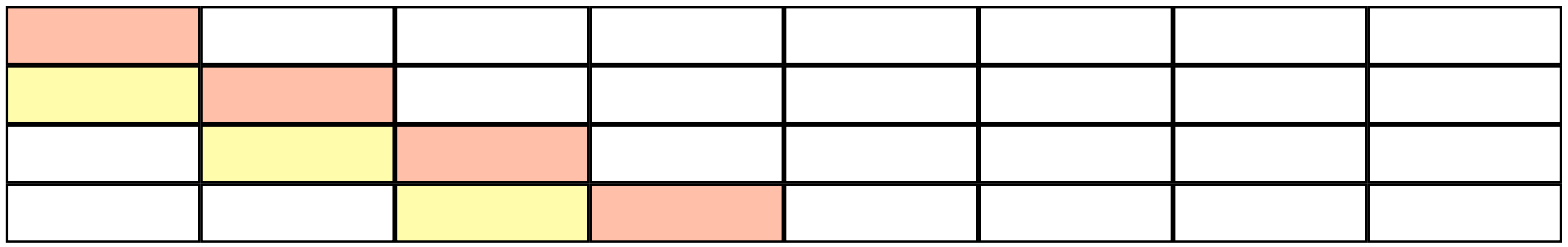
How can I get that?

Introduce a *padding*, an offset to make the memory accesses hit different banks

In steps of 8



In steps of 9



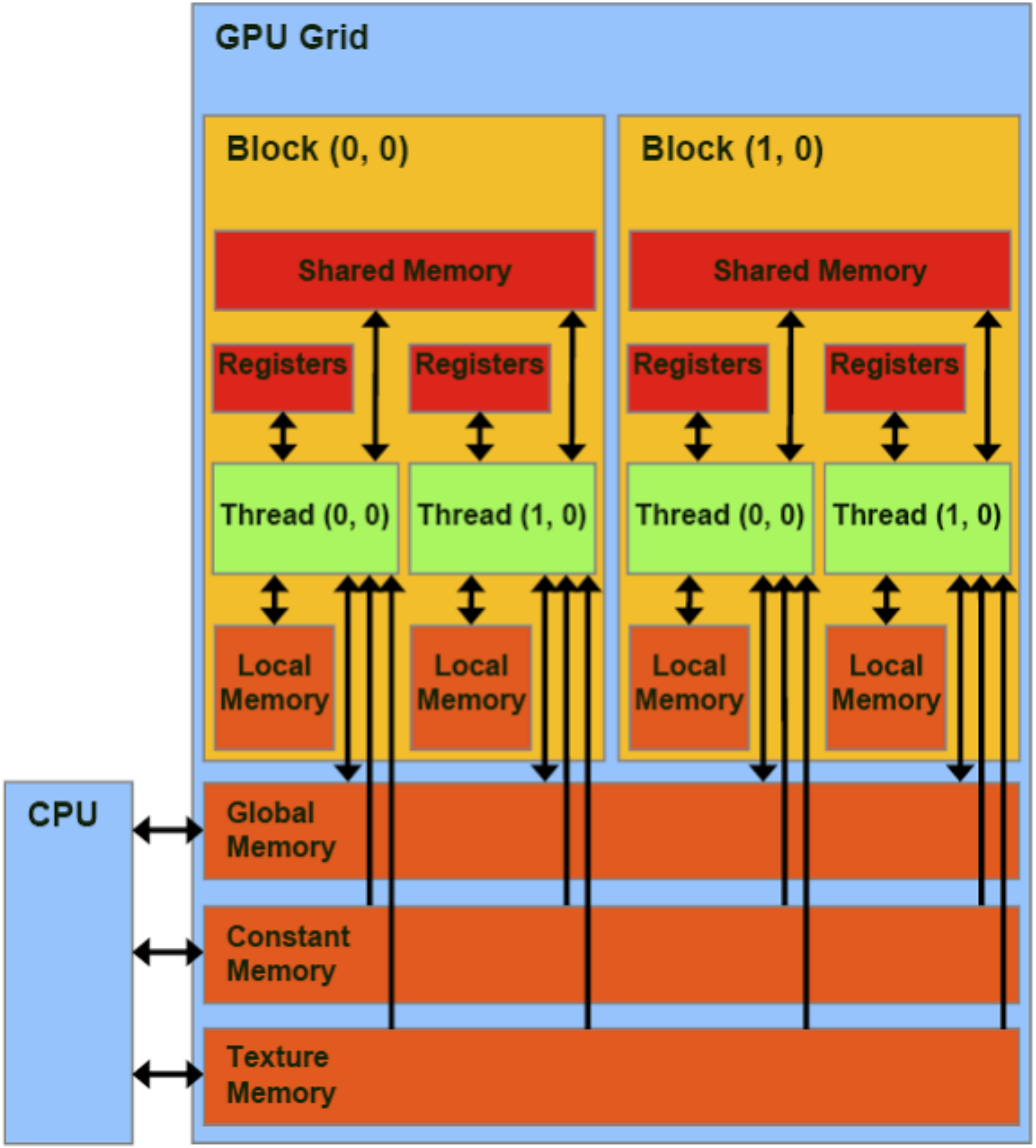
Constant memory

Sounds boring... but has its uses.

Read-only (for kernels)

__constant__ modifier

Use for input data, obviously


GPU Grid
Block (0, 0)
Block (1, 0)
Shared Memory
Shared Memory
Registers
Registers
Registers
Registers
Thread (0, 0)
Thread (1, 0)
Thread (0, 0)
Thread (1, 0)
Local Memory
Local Memory
Local Memory
Local Memory
CPU
Global Memory
Constant Memory
Texture Memory

Benefits of constant memory

- No cudaMemcpy needed! Just use it from kernel, write from CPU!
- For data read by all threads, significantly faster than global memory!
- Read-only memory is easy to cache.

Why faster access? When?

All (or many) threads reading the same data *simultaneously*.

One read can be broadcast to all ”nearby” threads.

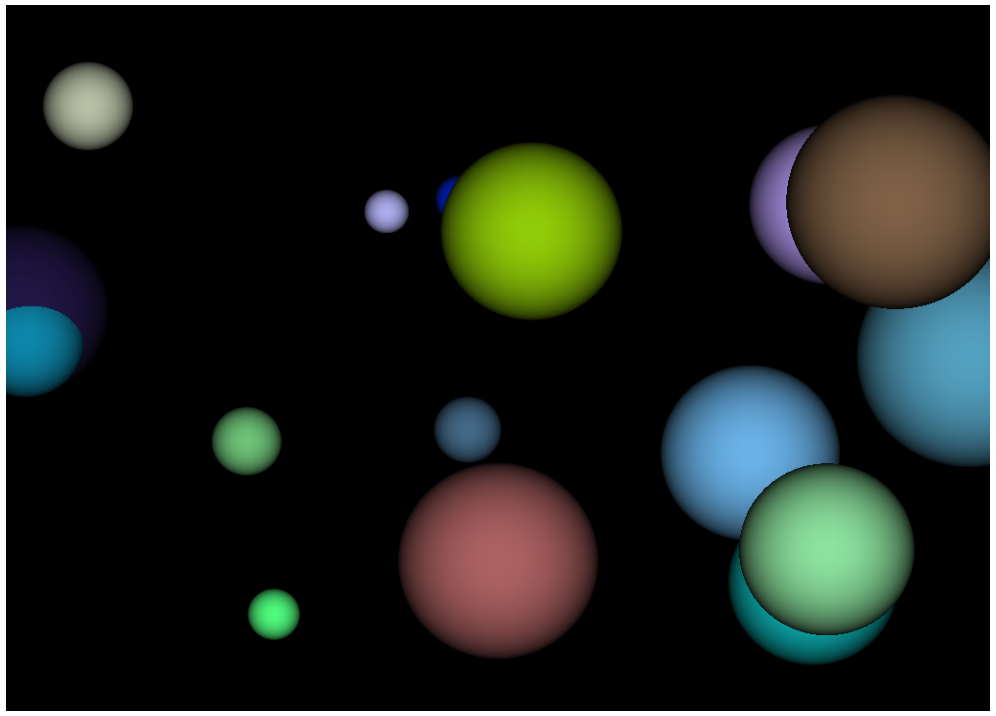
Nearby? All threads in same ”half-warp” (16 threads)

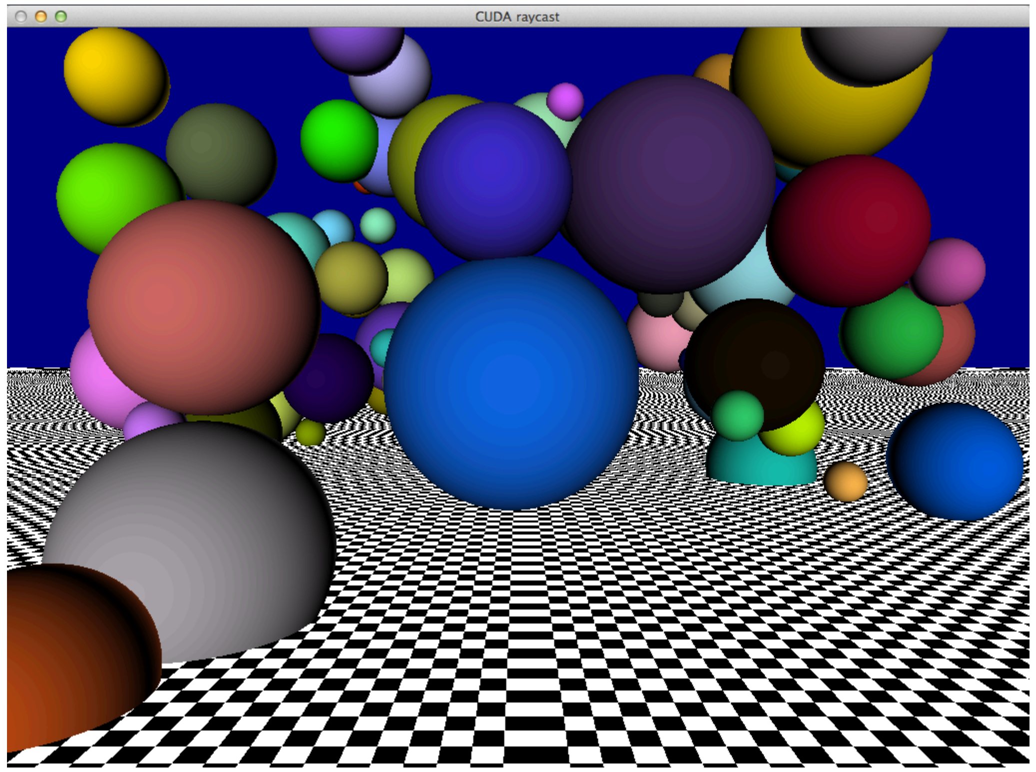
But no help if threads are reading different data!

Example of using constant memory: Ray-caster

Two demos, "Cuda by example" and "Attack in packs"

With and without using __const__





Ray-caster example

Every thread renders one pixel

Loop through all spheres, find closest with intersection

Write result to an image buffer.

Image buffer displayed with OpenGL.

Non-const: Uploads sphere array by cudaMemcpy()

Const: Declares array __const__, uses directly from kernel.
(Slightly simpler code!)

Ray-caster example

Resulting time:

Without using const: 31 ms

With const: 25 ms

Significant difference - for something that simplified the code!

Constant memory conclusions

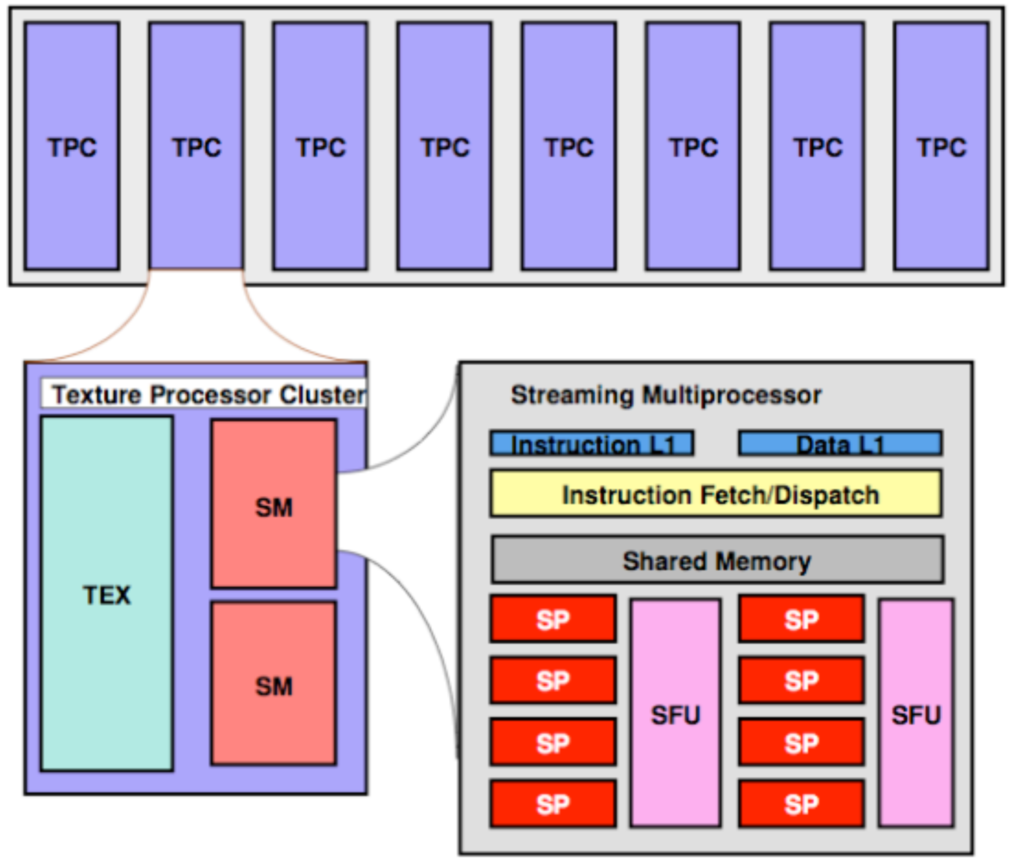
Relatively fast memory access - for the case when all threads read the same memory *simultaneously*!

Some advantage for code complexity.

NOT something we use for everything.

Texture memory/ Texture units

Using texture units to access memory



G80 processor hierarchy

Texture memory/ Texture units

Texture memory, yet another kind of memory (or memory access method)

But didn't we hide the graphics heritage...?

Access global memory through the texturing units. Lets CUDA take advantage of the strong points with texturing units.

Texture memory features

Read-only (writable using "surface objects").

Cached! Can be fast if data access patterns are good.

Texture filtering, linear interpolation.

Edge handling.

Especially good for handling 4 floats at a time (float4).

cudaBindTextureToArray() binds data to a texture unit.

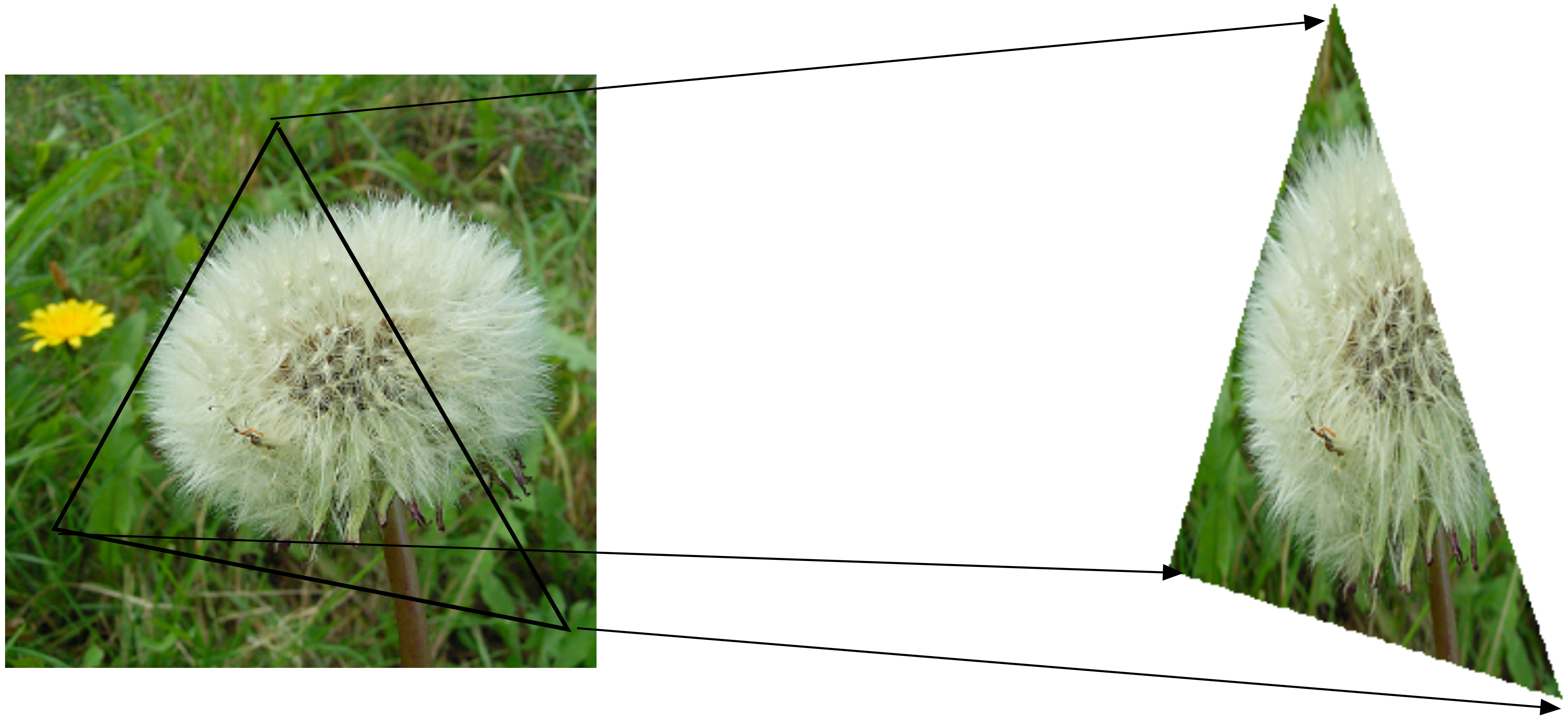
Texture memory for graphics

Texture data mostly for rendering textures

One texel used by 4 neighbor pixels (when not exact integer coordinates)

Designed for *spatial locality*

Varying access patterns - but neighbors are still neighbors!



Spatial locality for other things than textures

Image filters of local nature

Physics simulations with local updates, transfer of heat, liquids, pressure...

Big jumps, no gain!

Using texture memory in CUDA

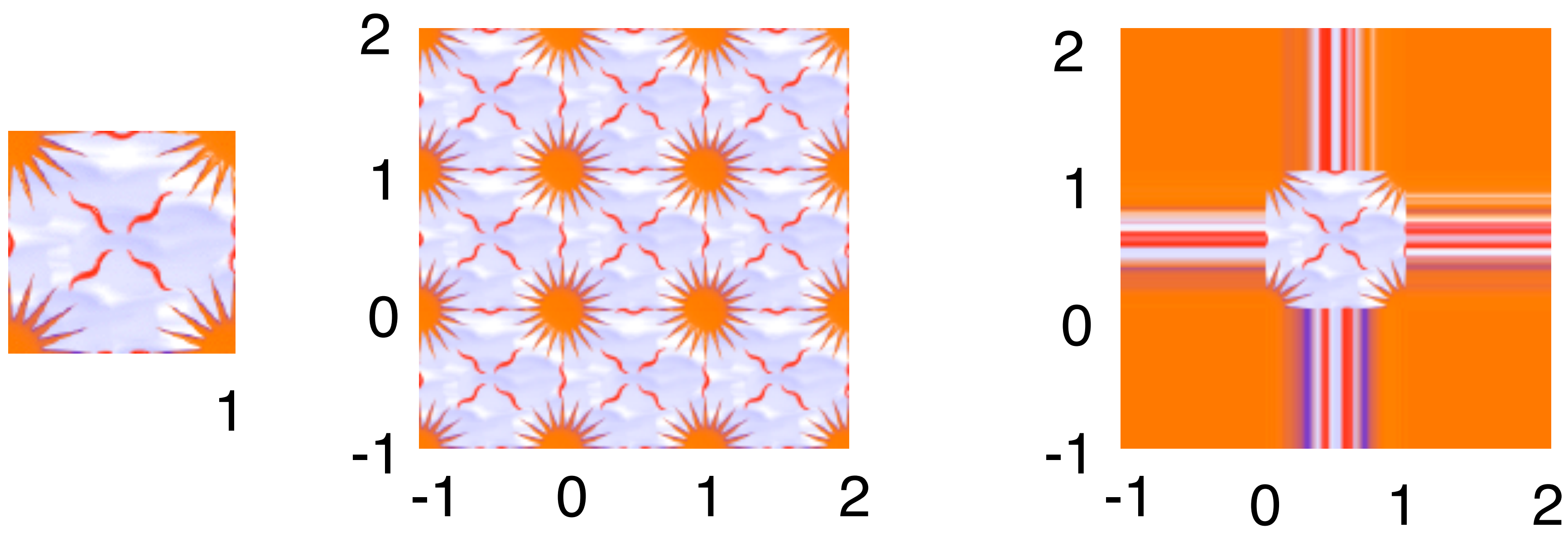
Allocate with cudaMalloc

Bind to texture unit using cudaBindTexture2D()

Read from data using tex2D()

Drawback: Just like in OpenGL, messy to keep track of which texture unit/texture reference is which data.

Clamp and repeat



Texture access needs no boundary checks!

Clamp and repeat

You are used to this

ERROR	ERROR	ERROR	ERROR
ERROR	1	2	ERROR
ERROR	3	4	ERROR
ERROR	ERROR	ERROR	ERROR

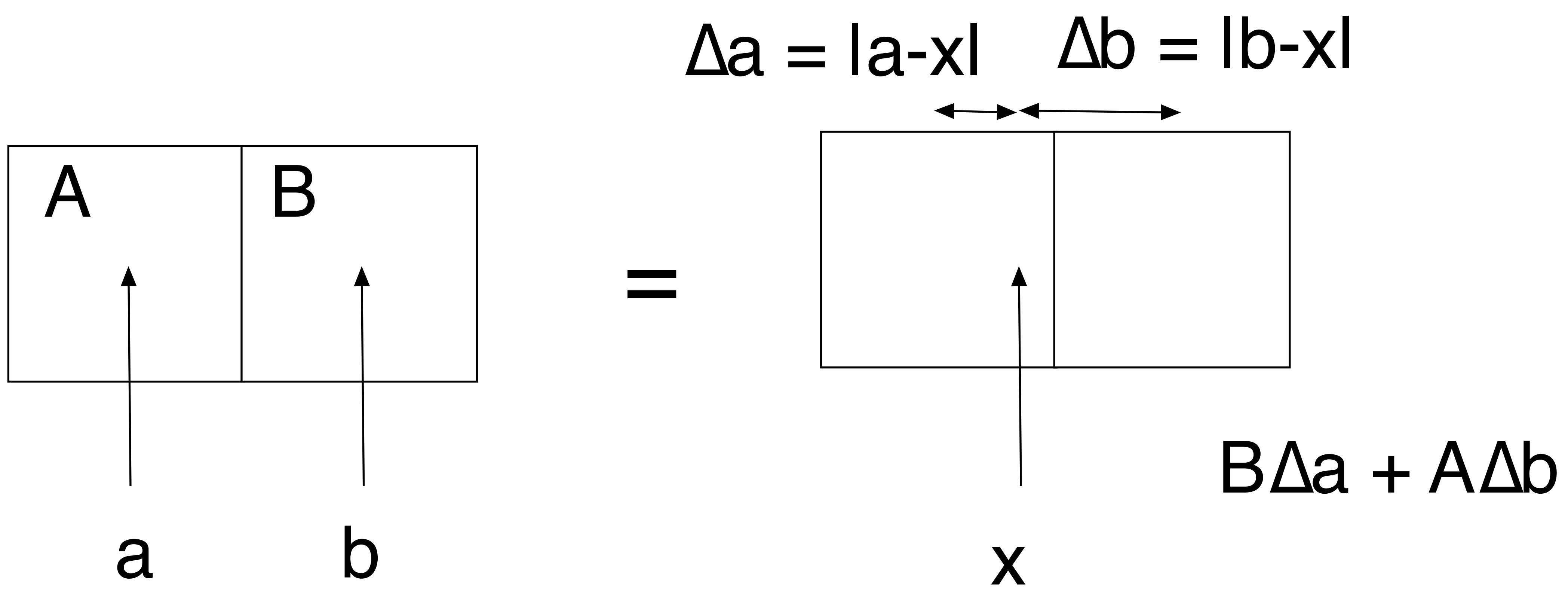
Now you can get this

4	3	4	3
2	1	2	1
4	3	4	3
2	1	2	1

or this

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Interpolation

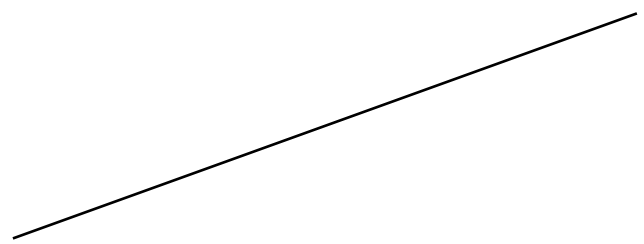


Texture accesses and calculations hardware accelerated!

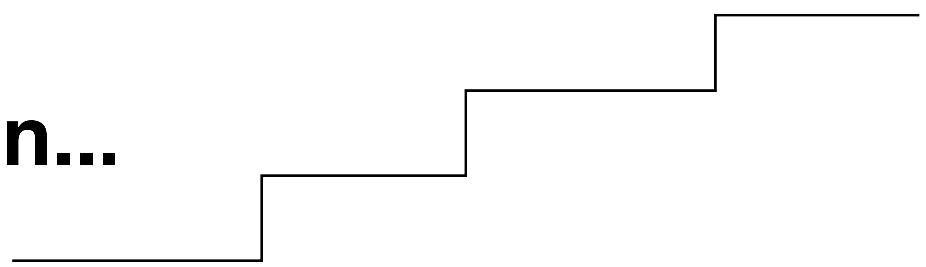
Hardware interpolation too good to be true...

The interpolation trick sounds kind of useful (for some cases)... but isn't as useful as it seems.

Why? It is meant for interpolating between texels, visually. Small errors is not a problem then! May have low precision, like 10 steps.

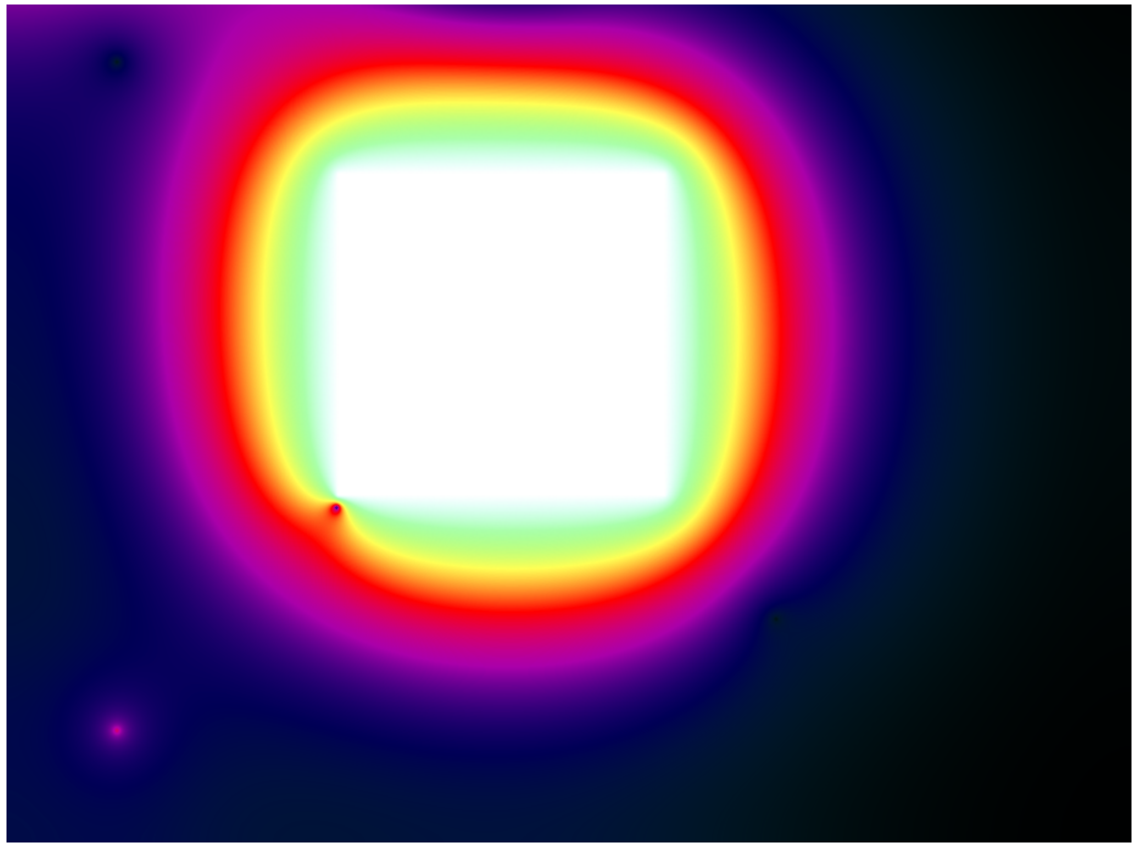


Not as fun then...



Demo using texture memory

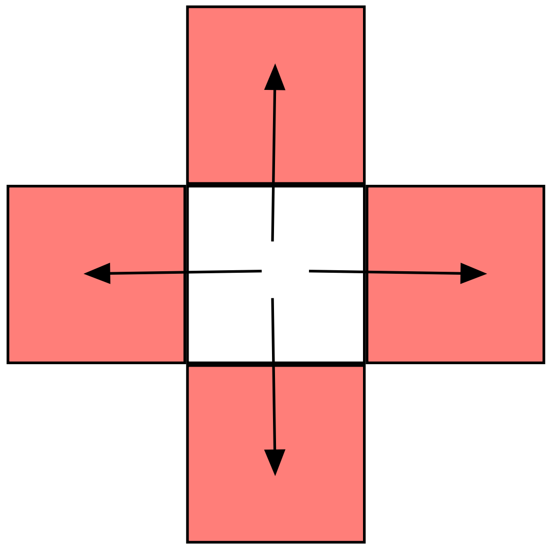
Heat transfer demo



Demo using texture memory

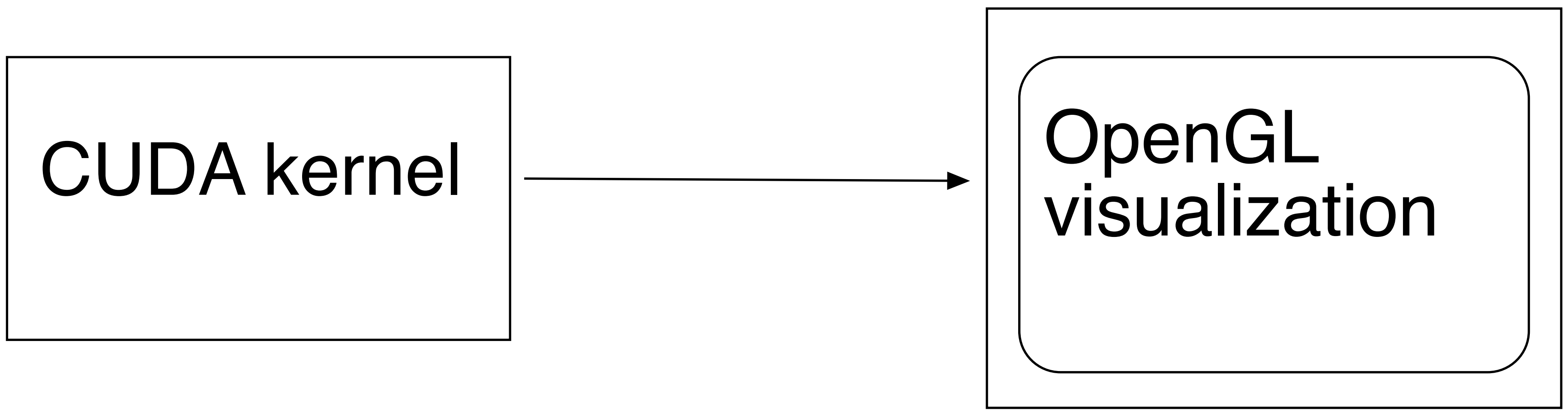
Heat transfer demo

Makes local operations modelling heat dissipation



CUDA-OpenGL Interoperability

Visualize results with OpenGL

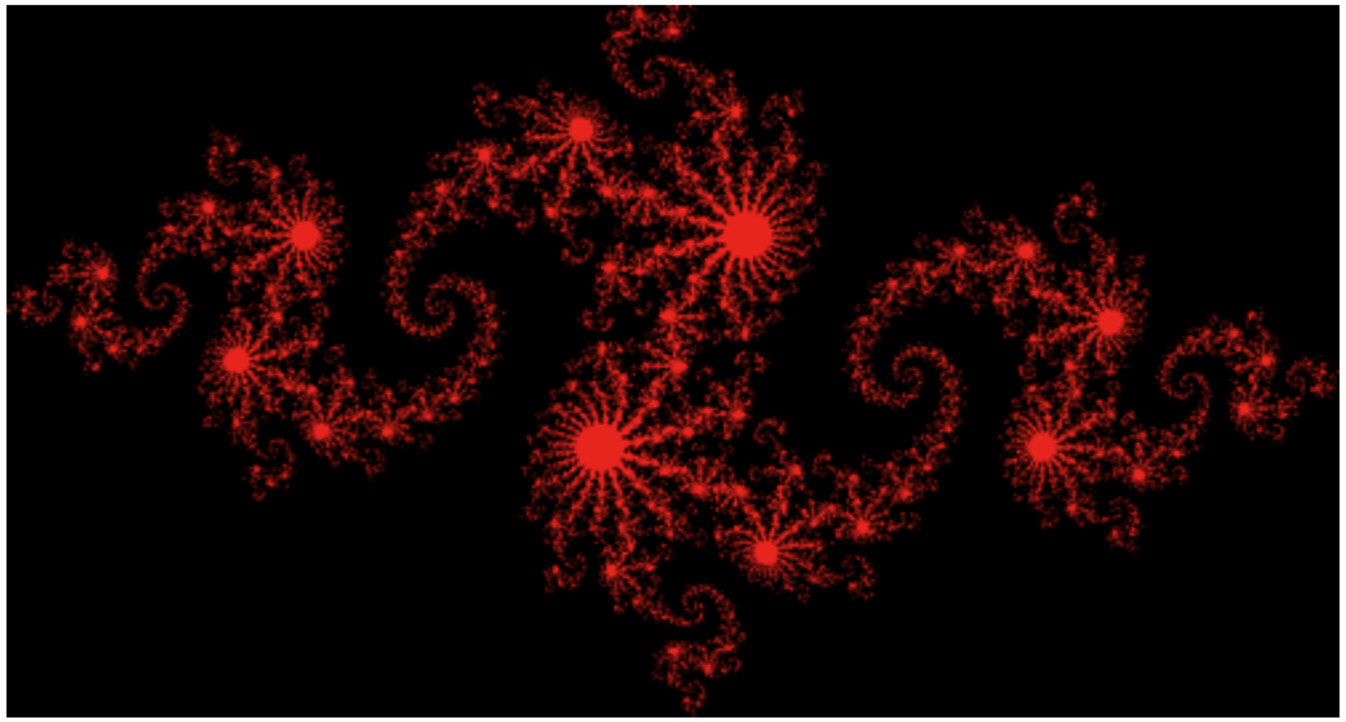


CUDA and graphics

Simplest way: Pass output from CUDA, typically to an OpenGL texture.

Example: Julia set, Lab 4 Mandelbrot, ray caster...

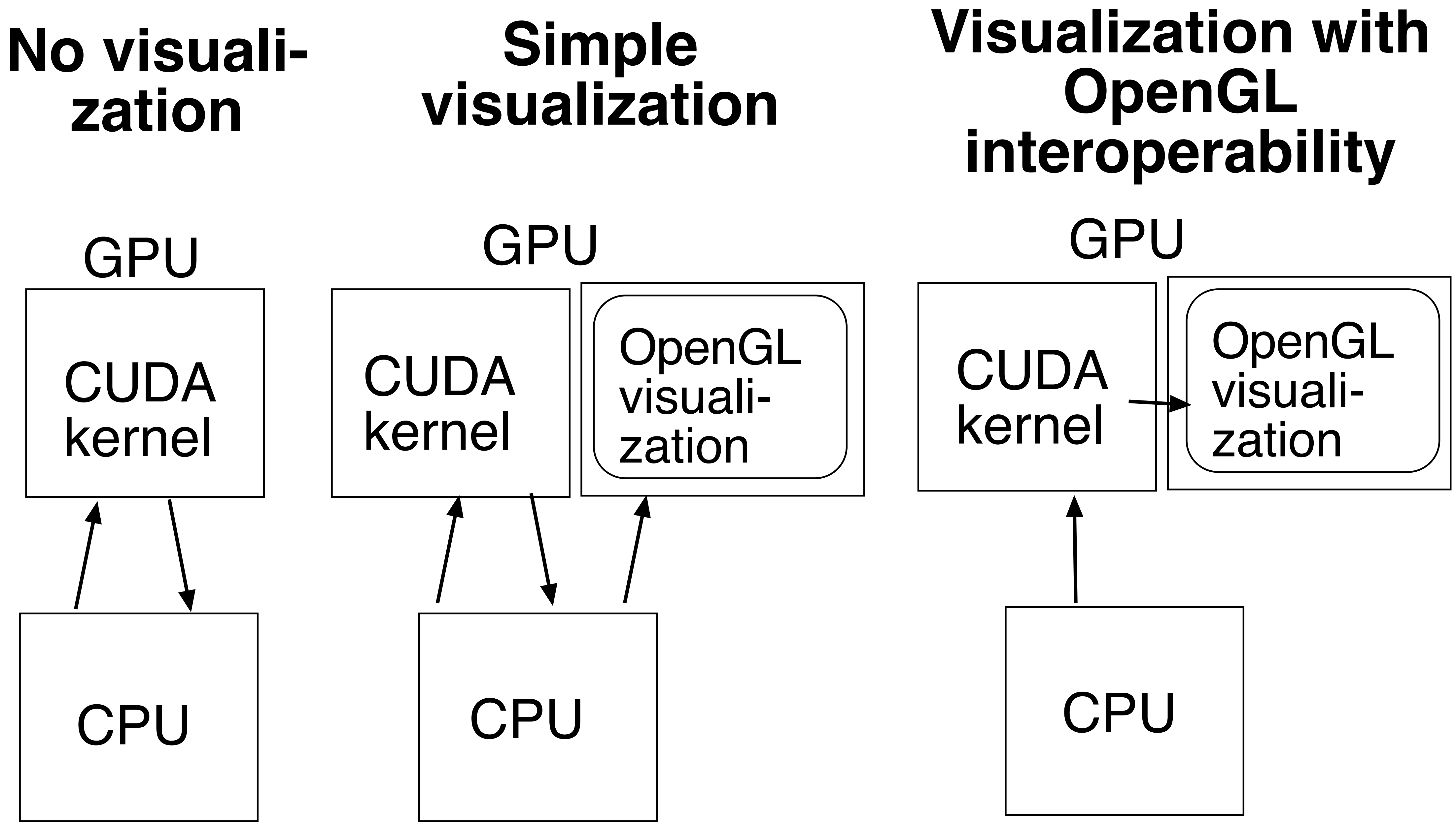
Good for visualizing results. Better methods exist, without having to move data to CPU and back.



CUDA-OpenGL Interoperability

- Integrate for better performance!
- Possible to visualize without leaving GPU

An output which is not the CPU


No visuali-
zation
Simple
visualization
Visualization with
OpenGL
interoperability
GPU
CUDA
kernel
CPU
GPU
CUDA
kernel
OpenGL
visuali-
zation
CPU
GPU
CUDA
kernel
OpenGL
visuali-
zation
CPU

Steps for interoperability

- Decide what data CUDA will process
- Allocate with OpenGL
- Register with CUDA
- Map buffer to get CUDA pointer
- Pass pointer to CUDA kernel
- Release pointer
- Use result in OpenGL graphics

- **Allocate with OpenGL**
- **Register with CUDA**

Allocate VBO (vertex buffer)

```
glGenBuffers(1, &positionsVBO);
glBindBuffer(GL_ARRAY_BUFFER, positionsVBO);
unsigned int size = NUM_VERTS * 4 * sizeof(float);
glBufferData(GL_ARRAY_BUFFER, size, NULL, GL_DYNAMIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Register with CUDA

```
cudaGraphicsGLRegisterBuffer(&positionsVBO_CUDA, positionsVBO,
cudaGraphicsMapFlagsWriteDiscard);
```

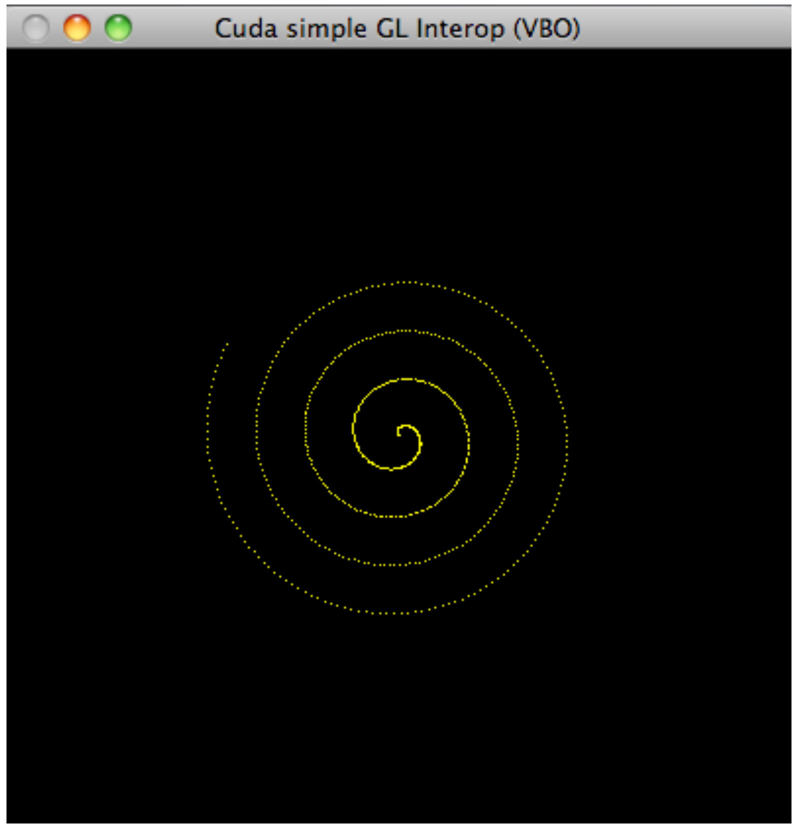
- **Map buffer to get CUDA pointer**
- **Pass pointer to CUDA kernel**
- **Release pointer**

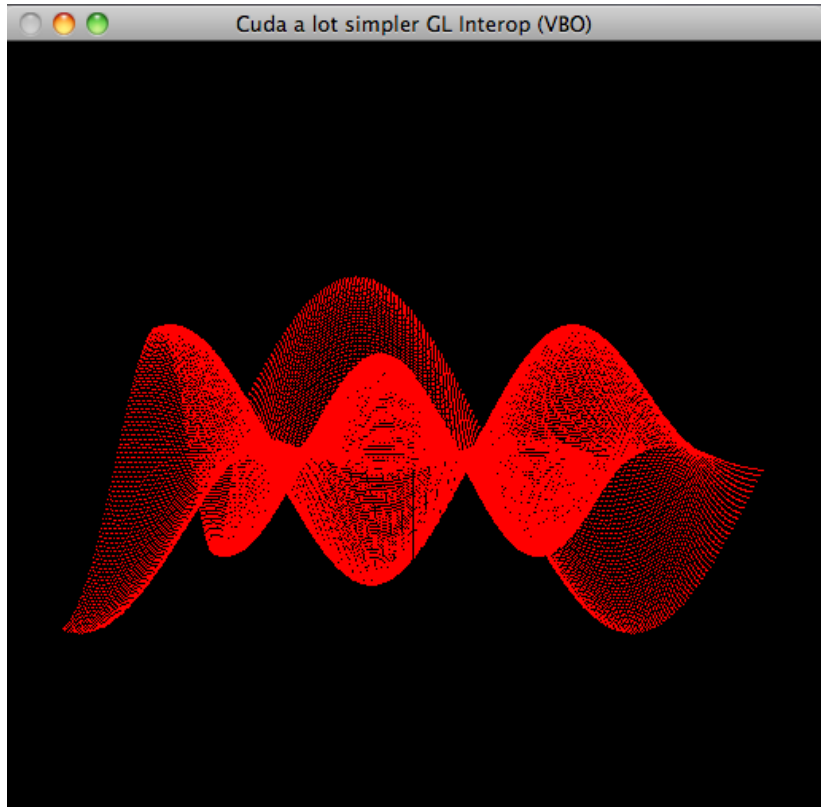
```
cudaGraphicsMapResources(1, &positionsVBO_CUDA, 0);
size_t num_bytes;
cudaGraphicsResourceGetMappedPointer((void**)&positions, &num_bytes,
positionsVBO_CUDA);printError(NULL, err);

// Execute kernel
dim3 dimBlock(16, 1, 1);
dim3 dimGrid(NUM_VERTS / dimBlock.x, 1, 1);
createVertices<<<dimGrid, dimBlock>>>(positions, anim, NUM_VERTS);

// Unmap buffer object
cudaGraphicsUnmapResources(1, &positionsVBO_CUDA, 0);
```

Simple CUDA kernel for producing vertices for graphics

```
// CUDA vertex kernel
__global__ void createVertices(float4* positions, float time, unsigned int num)
{
 unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;

 positions[x].w = 1.0;
 positions[x].z = 0.0;
 positions[x].x = 0.5*sin(kVarv * (time + x * 2 * 3.14 / num)) * x/num;
 positions[x].y = 0.5*cos(kVarv * (time + x * 2 * 3.14 / num)) * x/num;
}
```

Simple examples:





Just vertices - but you can draw surfaces, compute textures, use any OpenGL effects (light, materials)

But should we use CUDA for OpenGL?

Great for visualizing

Faster than going over CPU

Slower than plain OpenGL for graphics!

and OpenGL has CUDA-like functionality built-in!
(Compute Shaders.) (Later lecture)

Conclusions

CUDA can be coupled closer to OpenGL than the simple way we have done before!

Moving data back and forth is wastefui, there is performance to gain!

Some interesting alternatives exist as well.

Lecture questions:

1. Why can using constant memory improve performance?

2. What is CUDA Events used for?

3. What does coalescing mean and what should we do to get a speedup from coalescing?

4. Why can we not synchronize between blocks?

That's all folks!

Next: Sorting on the GPU.