Lecture 11 (#3 on GPU Computing)

More CUDA

In this episode...

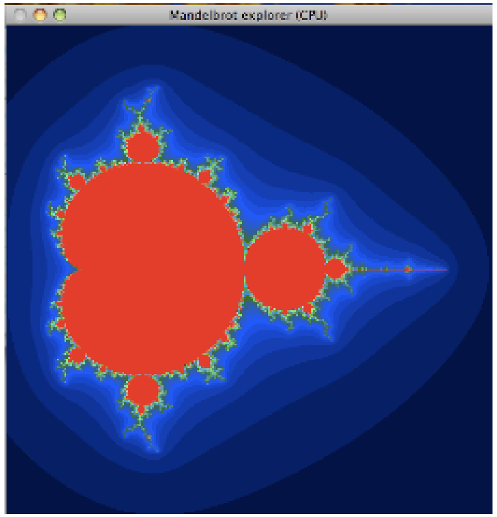
- Query device capabilities
- CUDA events
- More on CUDA memory:

Coalescing, Constant memory, Texture memory...

Lab 4

First version of the new lab available since yesterday (sunday)

Major change: ”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 involation 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

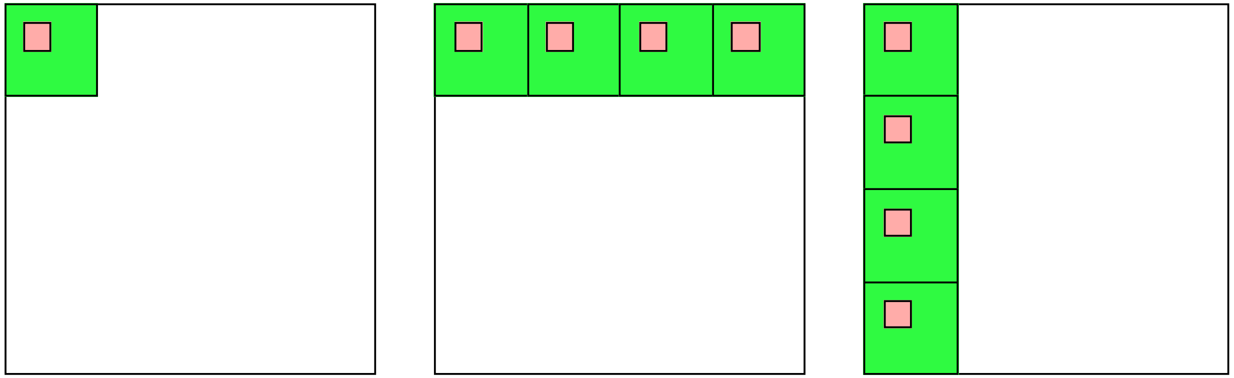
shared memory

constant memory

local memory

texture memory/texture units

Matrix multiplication example, using shared memory



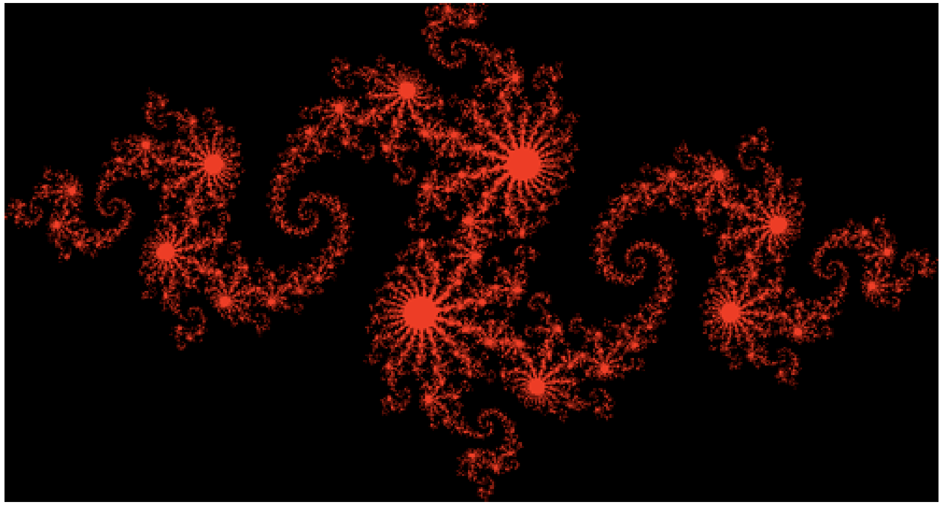
Huge speedup - my measly 9400M went from obvious loser to clearly faster than CPU!

CUDA and graphics

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

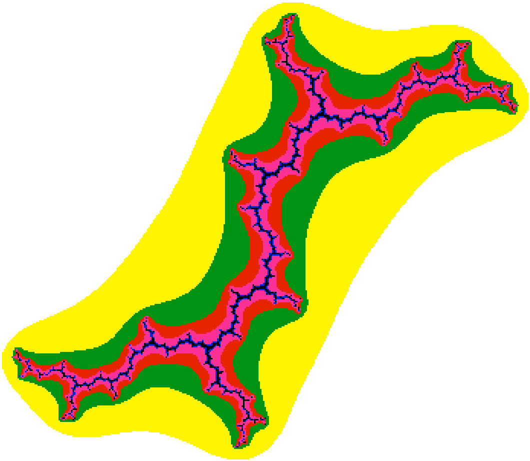
Example: Julia set.

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



Self-squaring fractals, the Julia set

$$z_{k+1} = z_k^2 + \lambda$$



Julia set for
$\lambda = (0, 1) = 0 + j$

Start with position in complex space.

Apply complex function recursively

Inspect distance to origin

Perfectly parallel algorothm

Over to today’s episode:

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?

Query devices

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

New boards may have totally different data.

Query CUDA for a list of features using cudaGetDeviceProperties()

Example query result

```
---- 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)
```

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: A lower bound for performance.

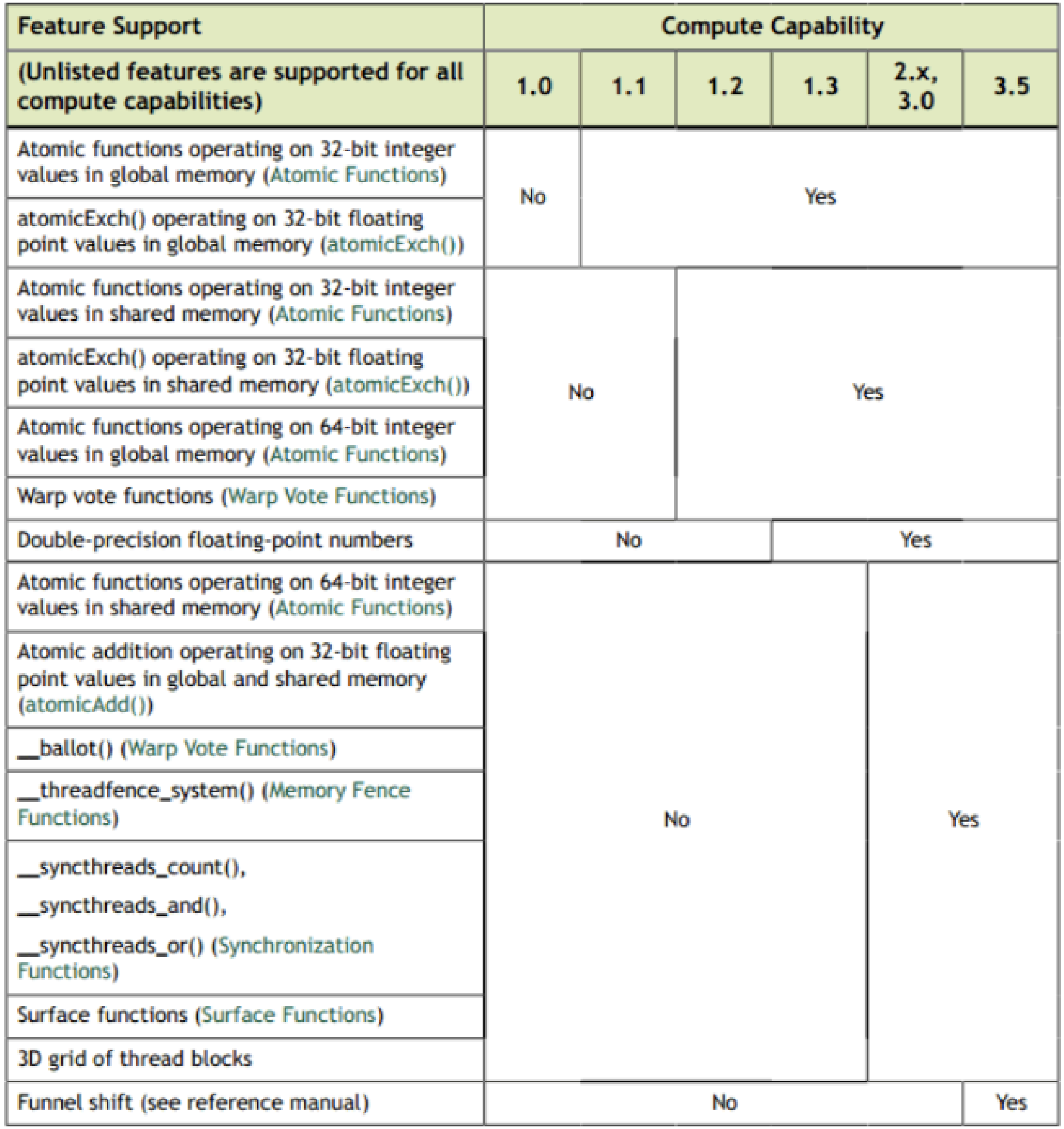
Number of SMs: Lower bound for blocks

Compute capability

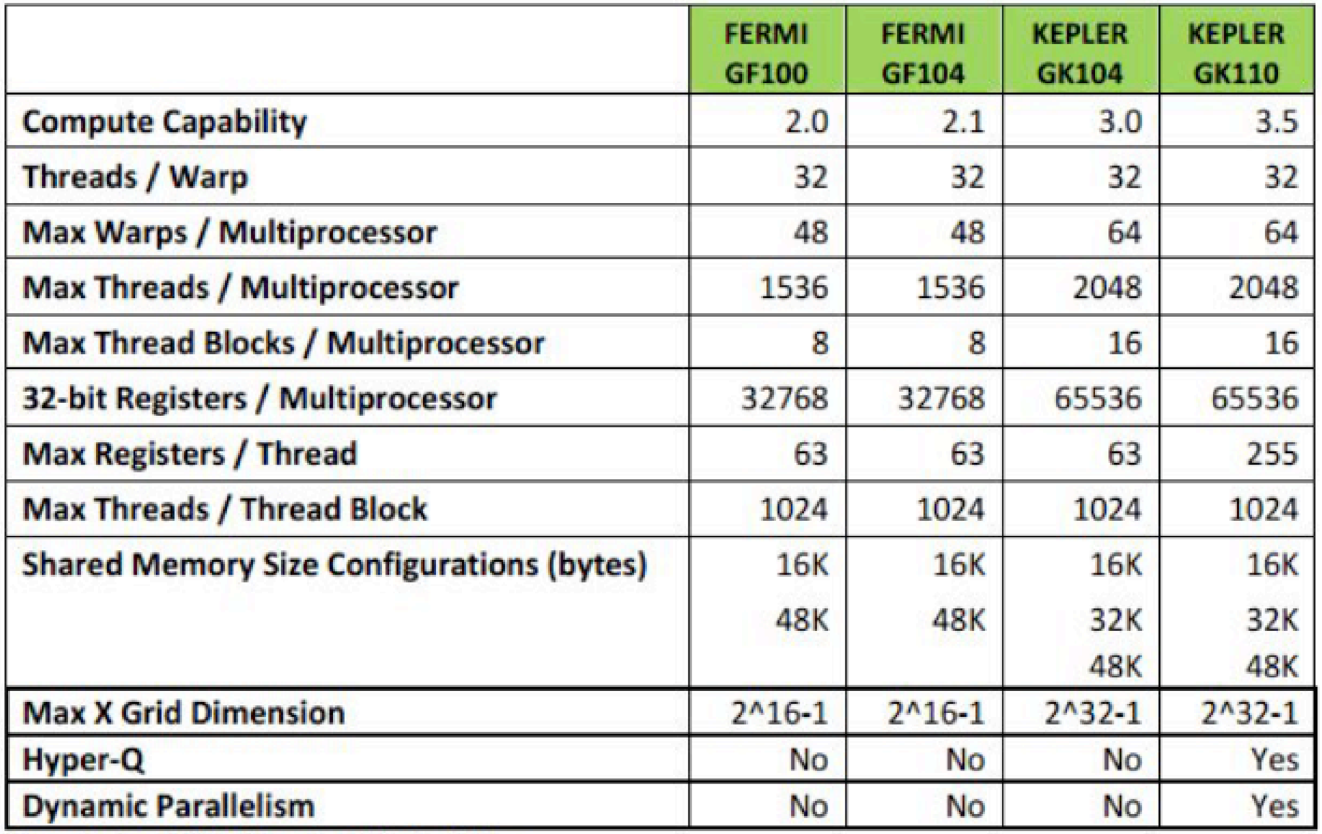
Essentially CUDA/architecture version number.

1.0: Original release.
1.1: Mapped memory, atomic operations.
1.3: Double support.
2.0: Fermi.
3.0: Kepler.
5.0: Maxwell.

-iTH

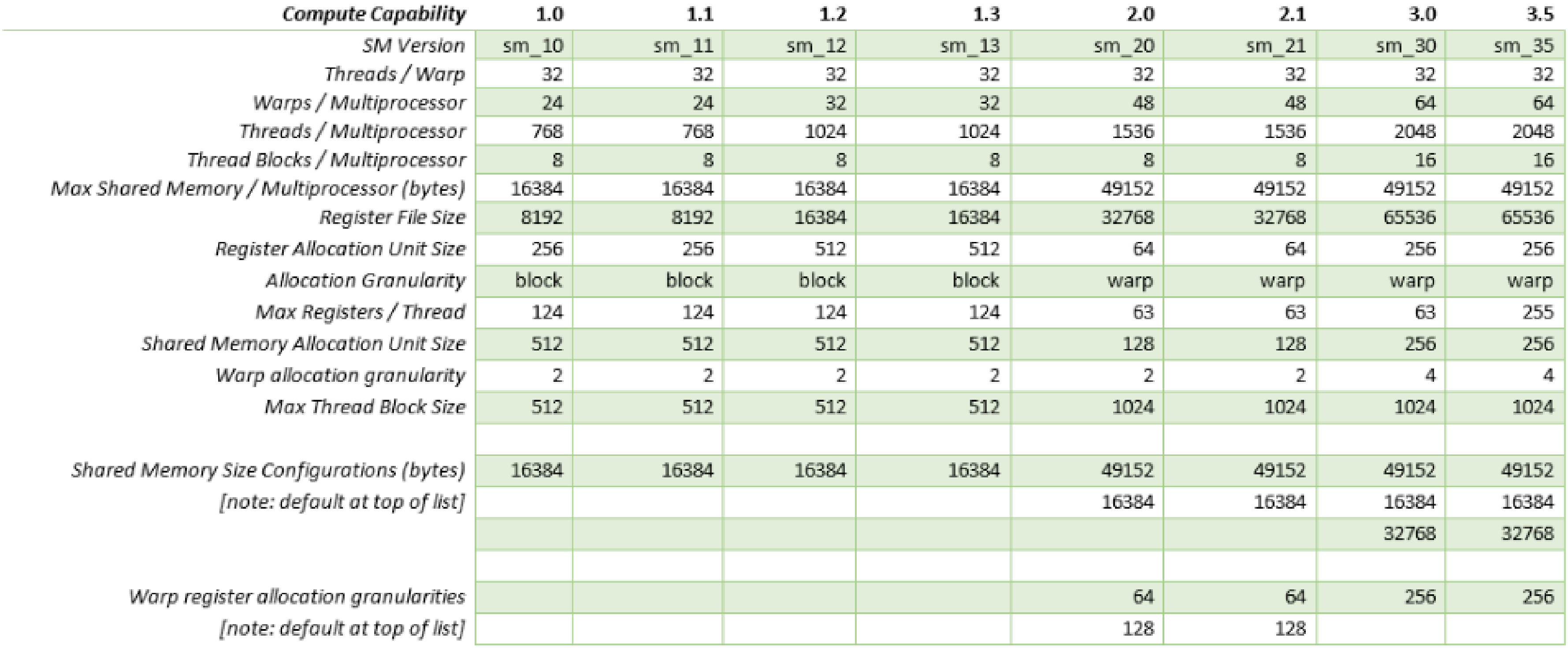


<table>
<tr><th>Feature Support</th><th colspan="6">Compute Capability</th></tr>
<tr><th>(Unlisted features are supported for all compute capabilities)</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x, 3.0</th><th>3.5</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in 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>



	FERMI GF100	FERMI GF104	KEPLER GK104	KEPLER GK110
Compute Capability	2.0	2.1	3.0	3.5
Threads / Warp	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16
32-bit Registers / Multiprocessor	32768	32768	65536	65536
Max Registers / Thread	63	63	63	255
Max Threads / Thread Block	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		

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

CUDA memory

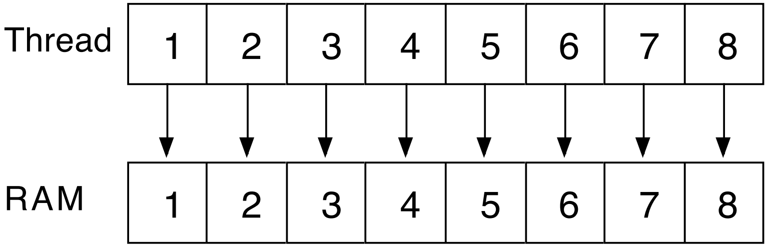
We already know...

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

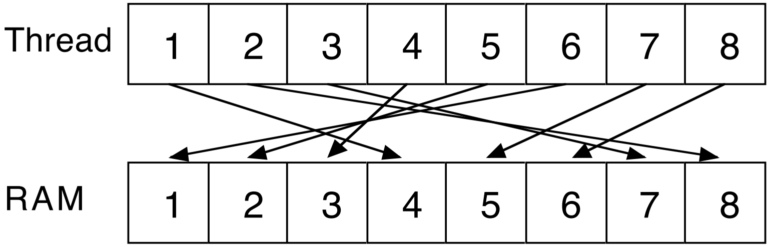
Coalescing

Always access global memory ”in order”

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



Good!



Bad!

WTF?

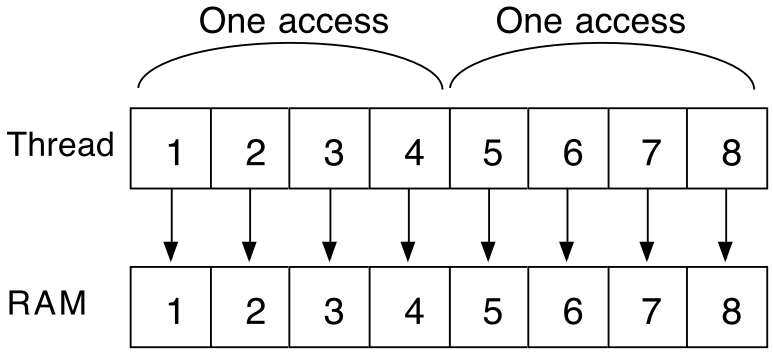
How can performance depend on what order I access my data??? Isn’t it ”random access”?

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

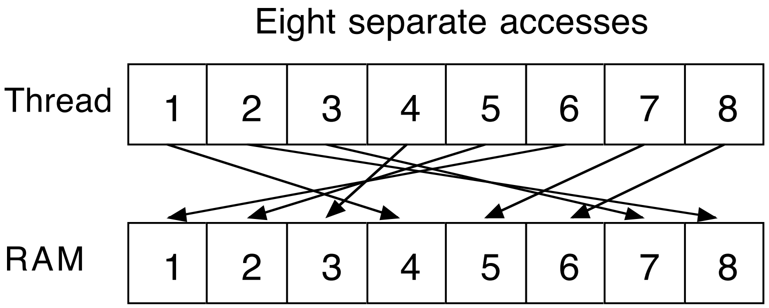
Why? Because the GPU bus is wider than your data!

Coalescing

Example: Assume that the data below is 1/4 of the bus width.



Good!



Bad!

Coalescing on Fermi & later

Effect reduced by caches - but not removed.

Coalescing is still needed for maximum performance.

Accelerating by coalescing

Pure memory transfers can be 10x faster by taking advantage of memory coalescing!

Example: Matrix transpose

No computations!

Only memory accesses.

Matrix transpose

Naive implementation

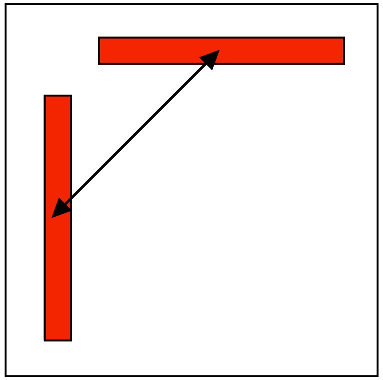
```
__global__ void transpose_naive(float *odata, float* idata, int width, int height)
{
   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
   unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

   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

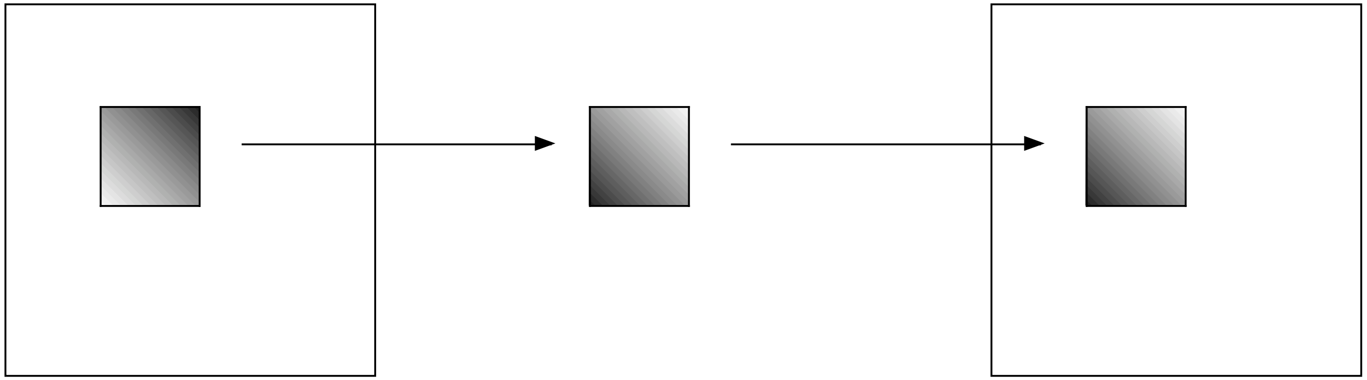
Coalescing problems



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 tglobal memory

Coalescing rules of thumb

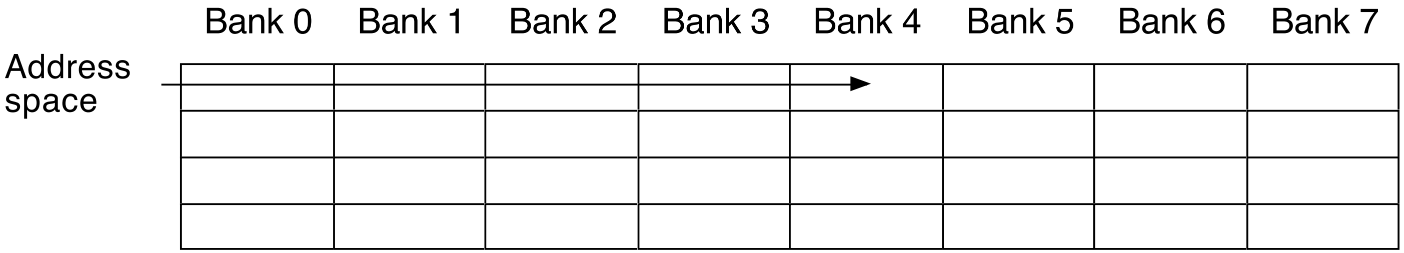
- The data block should start on a multiple of 64
- It should be accessed in order (by thread number)
- It is allowed to have threads skipping their item
- Data should be in blocks of 4, 8 or 16 bytes

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



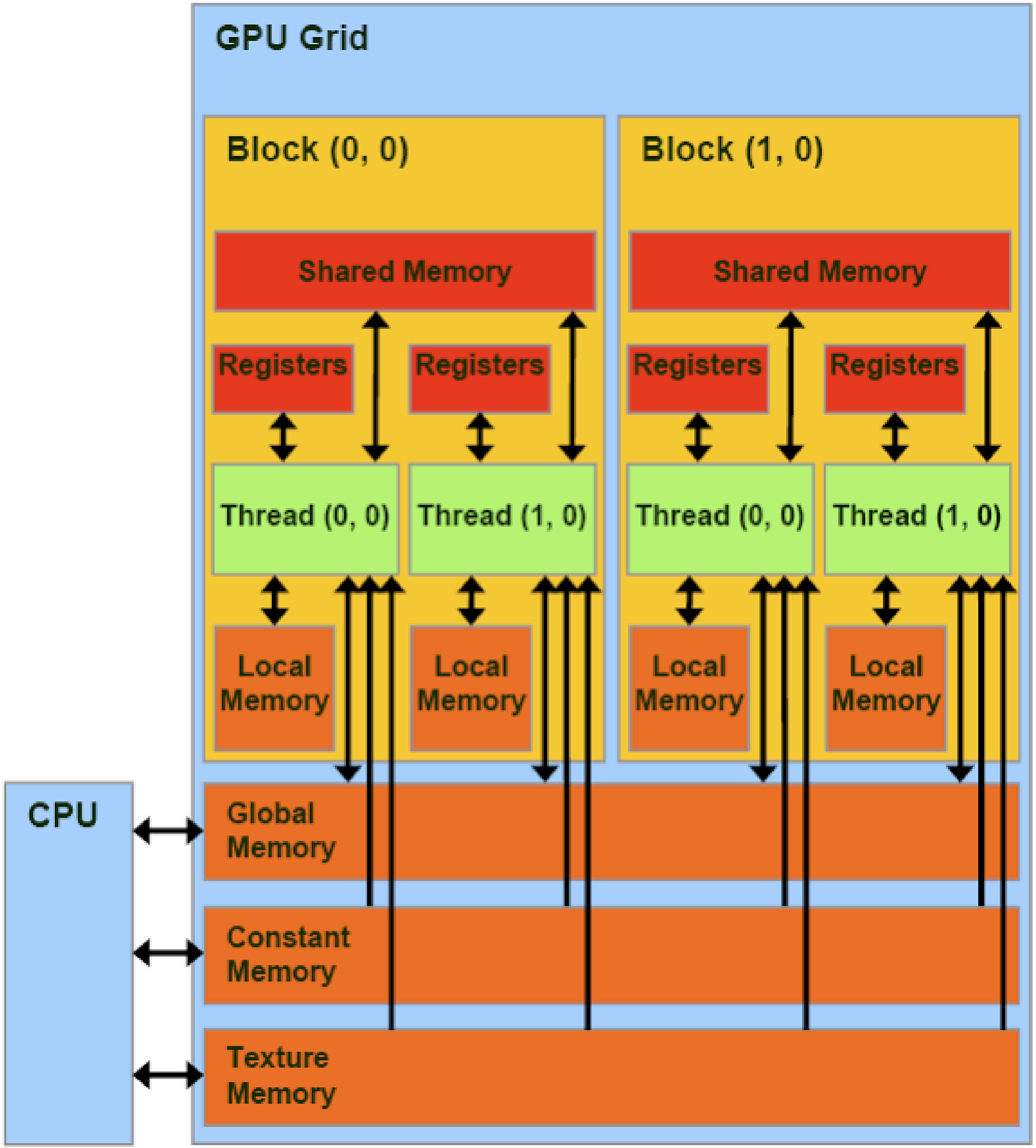
Constant memory

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 threads reading the same data.

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

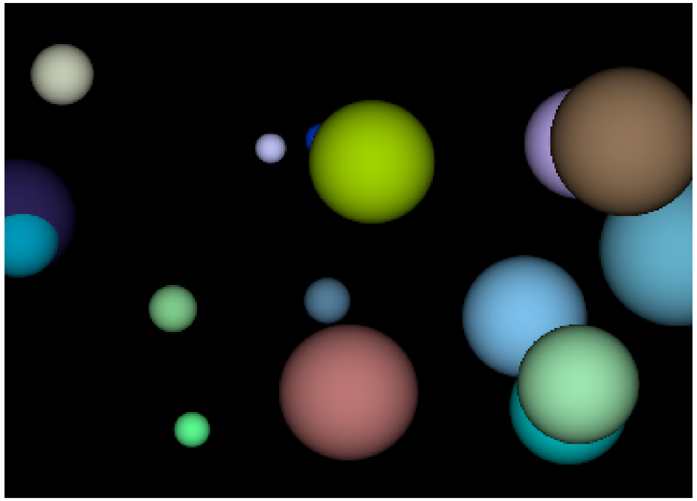
Nearby? All threads in same ”half-warp” (16 threads in most pre-Fermi architectures)

But no help if threads are reading different data!

Example of using constant memory: Ray-caster

Demo from ”CUDA by example”

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: 70.2 ms

With const: 41.9 ms

Significant difference - for something that simplified the code!

Constant memory conclusions

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

Some advantage for code complexity.

NOT something we use for everything.

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 though the texturing units. Lets CUDA take advantage of the strong points with texturing units.

Texture memory

Read-only.

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

Texture filtering, linear interpolation.

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

cudaBindTextureToArray() binds data to a texture unit.

Texture memory for graphics

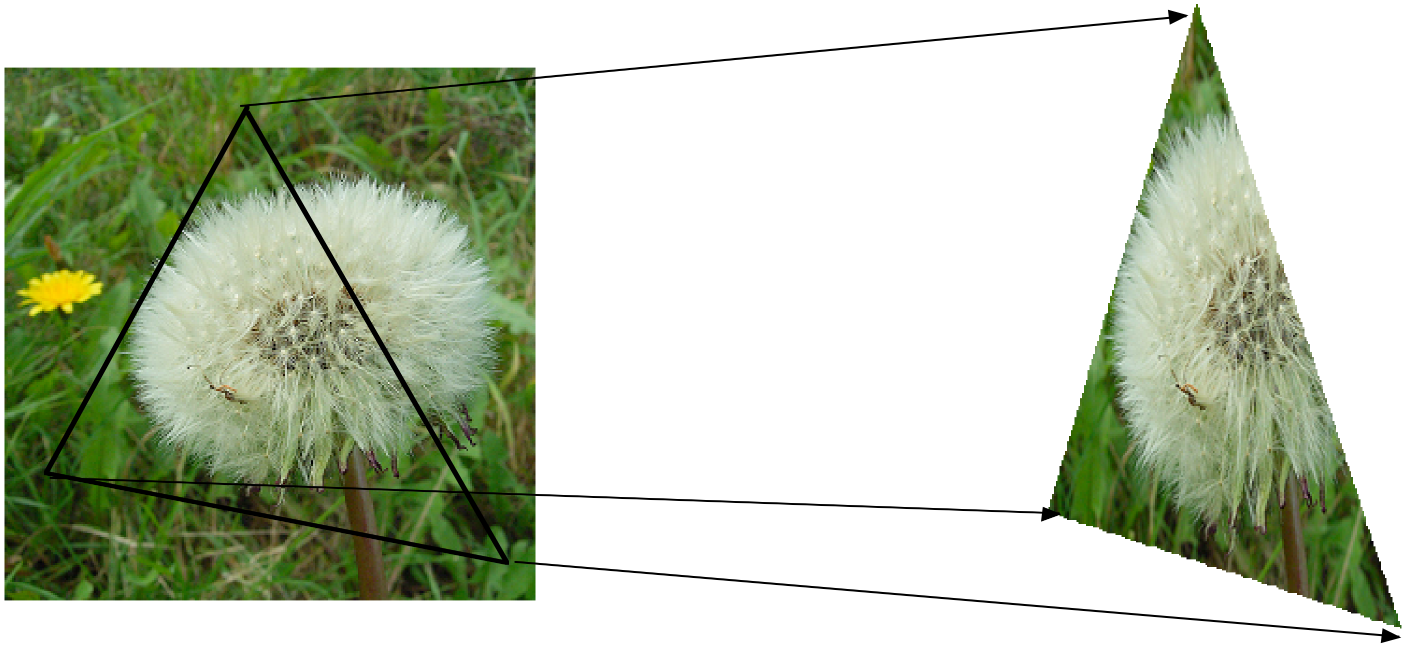
Texture data mostly for rendering textures

One texel used by 4 neigbor pixels

One pixel usually 4 bytes - more than one pixel can be read on one read.

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.

Interpolation

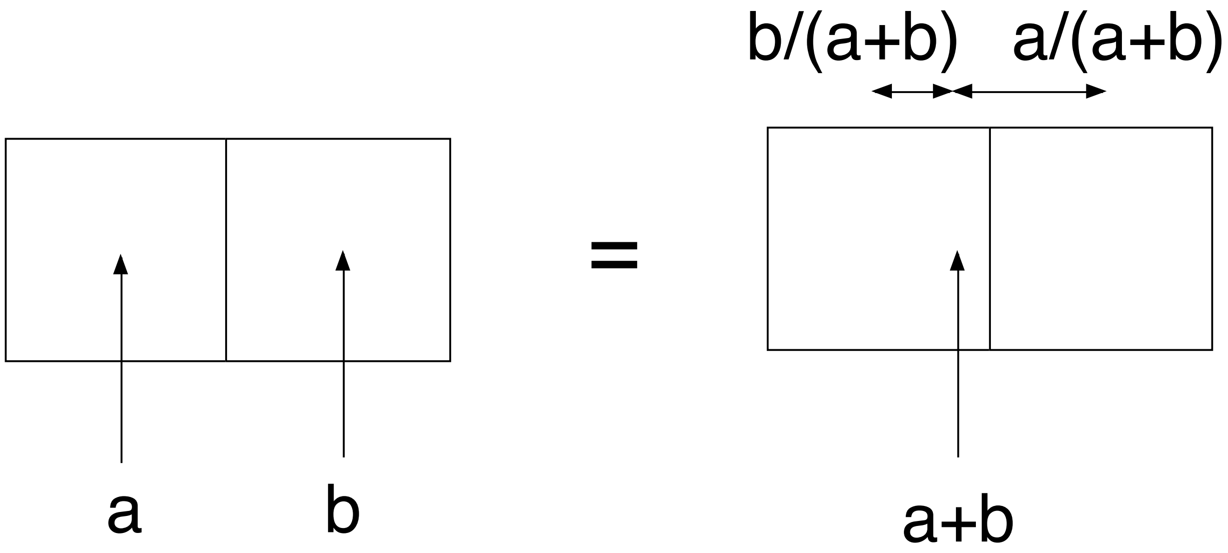
Computation tricks when optimizing

Texture access provides hardware accelerated linear interpolation!

Access texture data on non-integer coordinates and the texture hardware will do linear interpolation automatically!

Can be used for many calculations, e.g. filters.

Interpolation



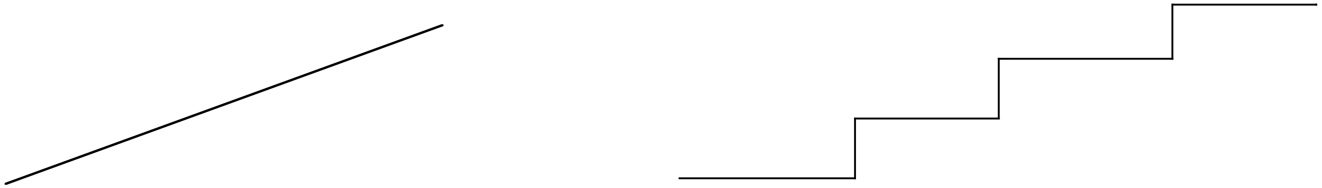
Texture accesses and calculations hardware accelerated!

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.

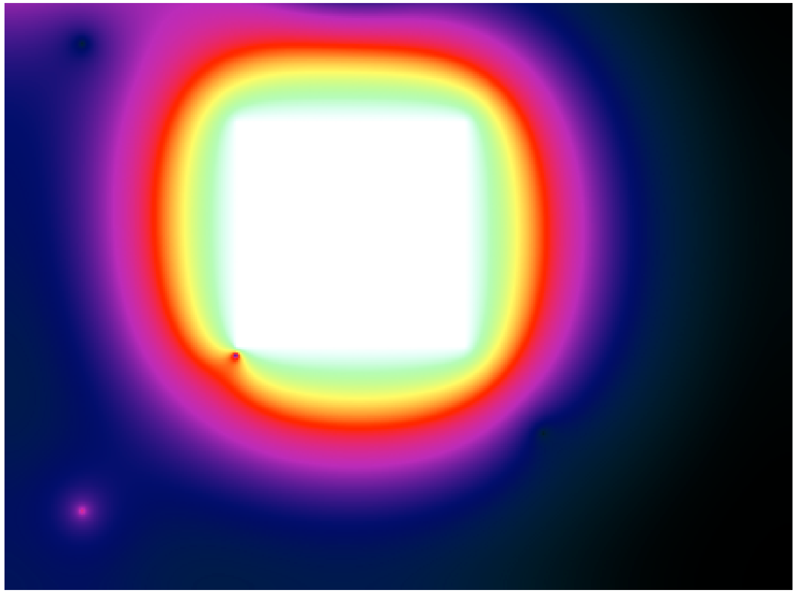
Why? It is ment for interpolating between texels, visually. Small errors is not a problem then! May have low precision, like 10 steps.

Not as fun then...



Demo using texture memory

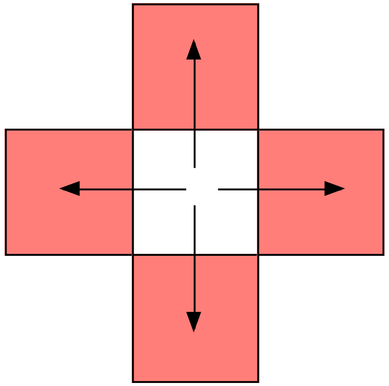
Heat transfer demo



Demo using texture memory

Heat transfer demo

Makes local operations modelling heat dissipation



Seriously... pretty slow. I could beat this with pure OpenGL any time. Why?

That’s all folks!

Next: Sorting on the GPU.