# CUDA memory

## Coalescing

## Constant memory

## Texture memory

## Pinned memory

---
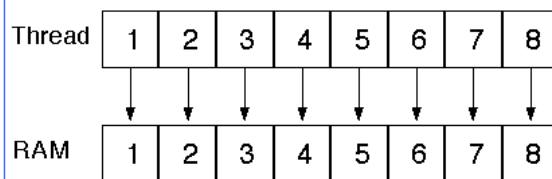
# CUDA memory

## We already know...

- Global memory is slow.

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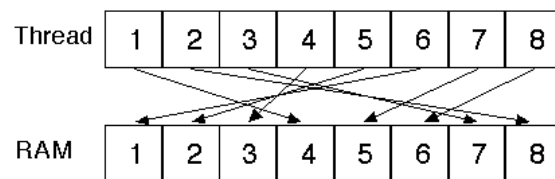
- There were some other kinds of memory...

# Coalescing

**Always access global memory "in order"**

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

| Thread | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|--------|---|---|---|---|---|---|---|---|

| RAM | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-----|---|---|---|---|---|---|---|---|

## Good!

| Thread | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|--------|---|---|---|---|---|---|---|---|

| RAM | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|-----|---|---|---|---|---|---|---|---|

## Bad!

---

# WTF?

**How can performance depend on what order I access my data??? Isn't it "random access"?**
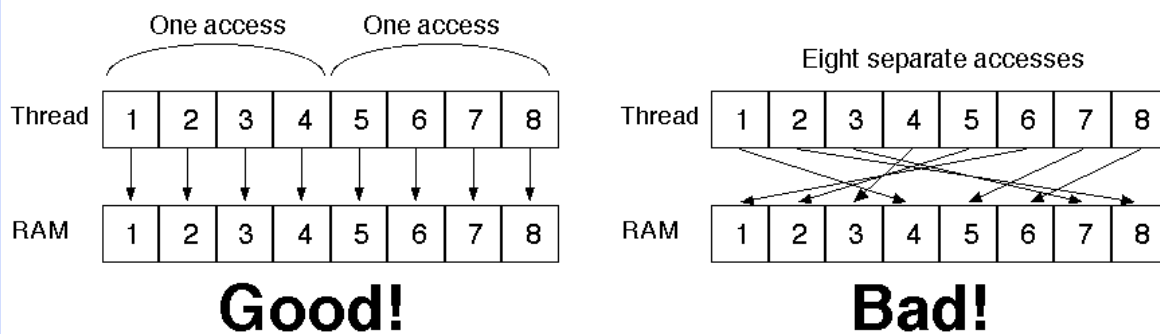
**Yes... You can access in any order you want, but ordered access *helps* the GPU to read more data in one access!**

**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 & Kepler

**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];
    }
}
```
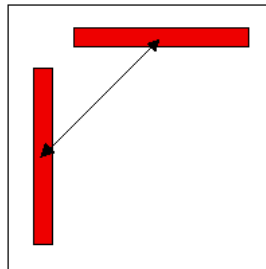
**How can this be bad?**

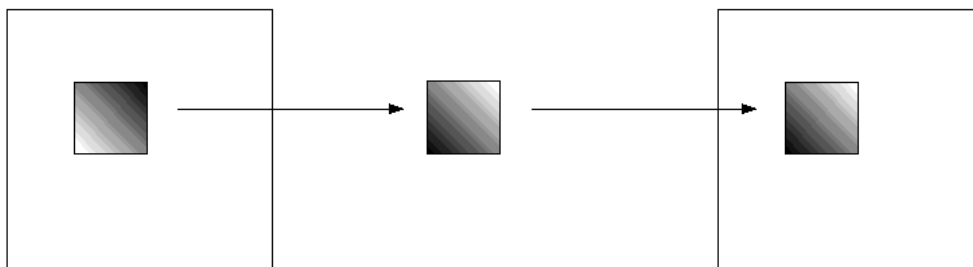# Matrix transpose

## 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
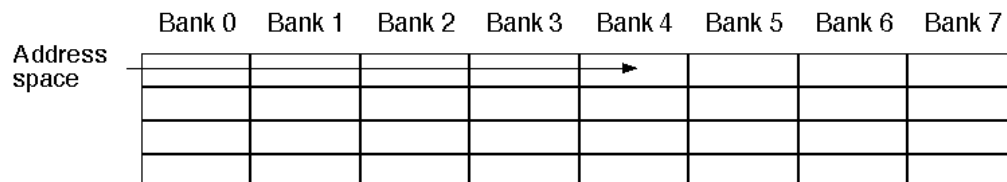
• Data should be in blocks of 4, 8 or 16 bytes

# Shared memory

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

**Interleaved, 32 bits chunks**

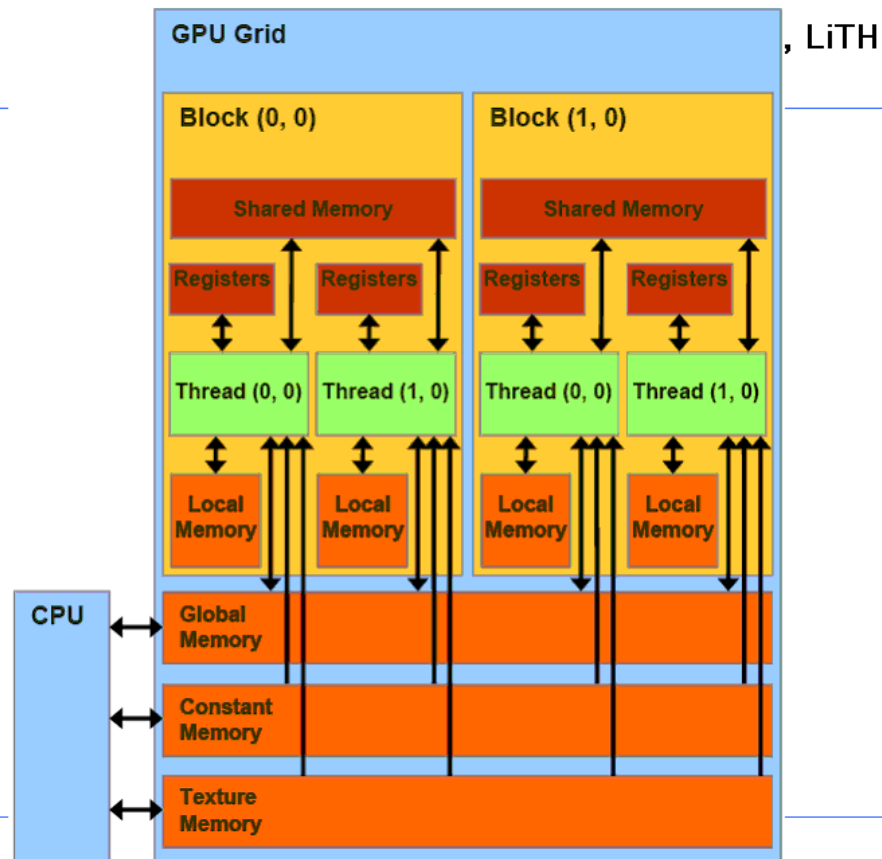**Thus: Address in 32-bit steps between threads for best performance**

| | Bank 0 | Bank 1 | Bank 2 | Bank 3 | Bank 4 | Bank 5 | Bank 6 | Bank 7 |
|---|---|---|---|---|---|---|---|---|
| Address space | | | | | → | | | |
| | | | | | | | | |
| | | | | | | | | |
| | | | | | | | | |

---

# Constant memory

**Sounds boring... but has its uses.**

**Read-only (for kernels)**

**__constant__ modifier**

**Use for input data, obviously**

Information Coding / Computer Graphics, ISY, LiTH

# 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.**

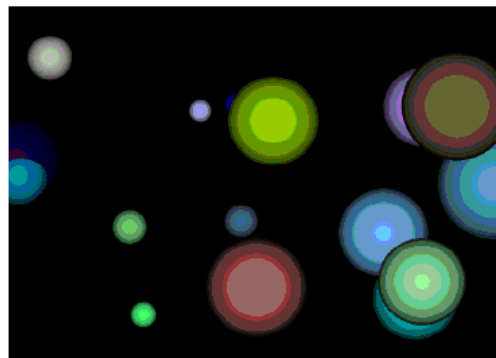**Nearby? All threads in same "half-warp" (16 threads in most pre-Fermi architectures)**

**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
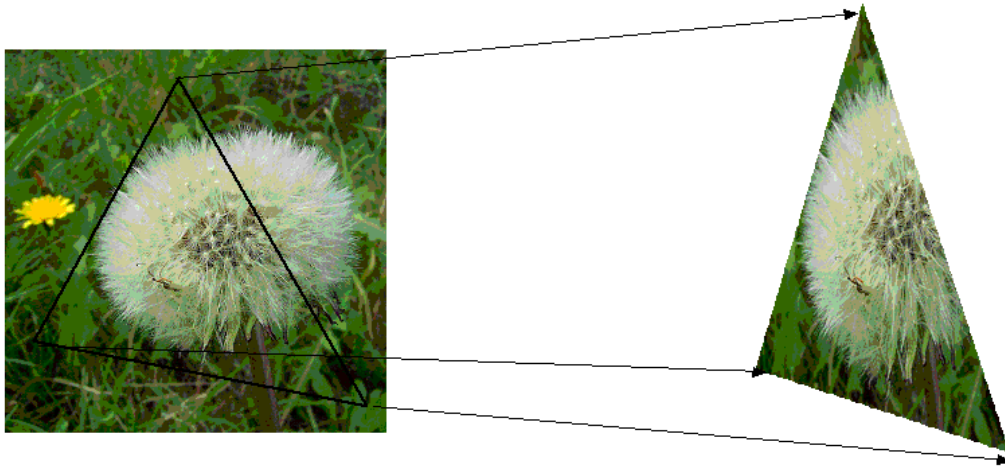
Texture data mostly for rendering textures

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.**

---
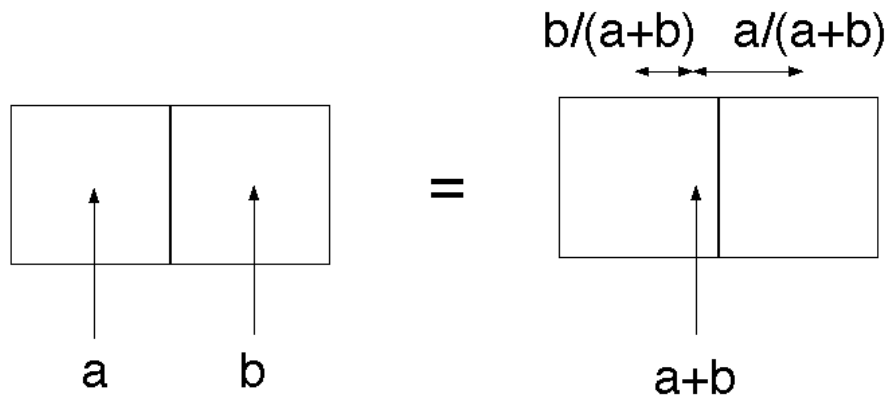
# Interpolation

## 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

b/(a+b)   a/(a+b)

a        b

=

a+b

## 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.
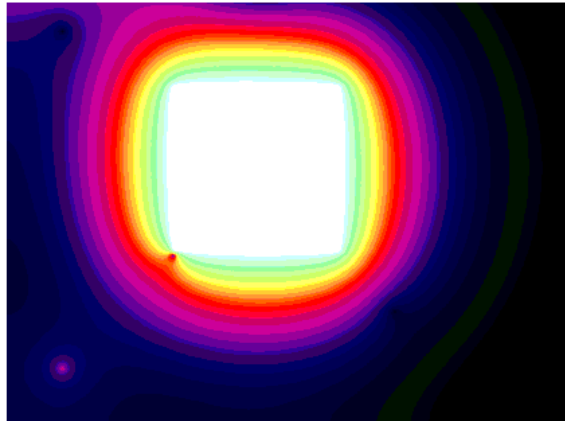
Not as fun then...

# Demo using texture memory

## Heat transfer demo

# Demo using texture memory

## Heat transfer demo

## Makes local operations modelling heat dissapation
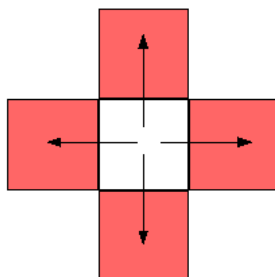


Seriously... pretty slow. I could beat this with pure OpenGL any time. Why?

# That's all folks!

Next: Computing with shaders.