



# **CUDA memory**

**Coalescing**

**Constant memory**

**Texture memory**

**Pinned memory**



## **We already know...**

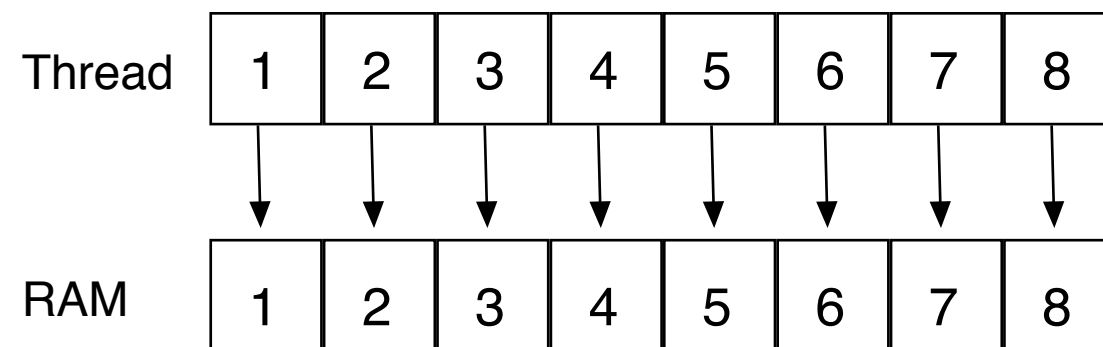
- **Global memory is slow.**
- **Shared memory is fast and can be used as "manual cache"**
- **There were some other kinds of memory...**



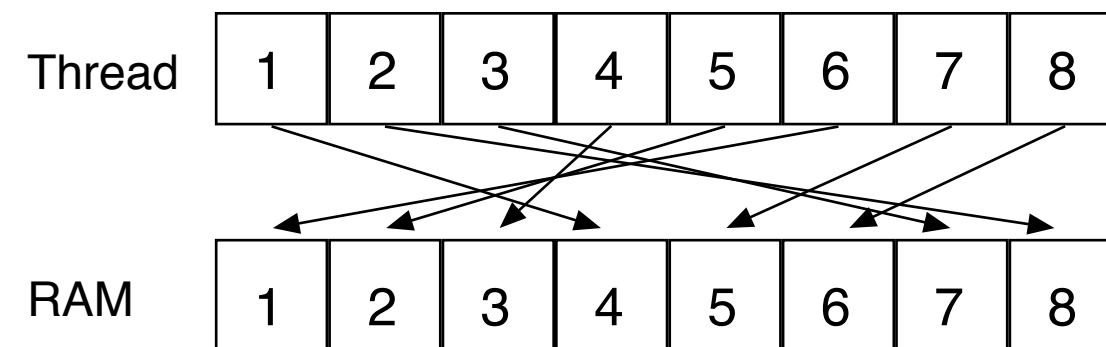
# Coalescing

**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"?**

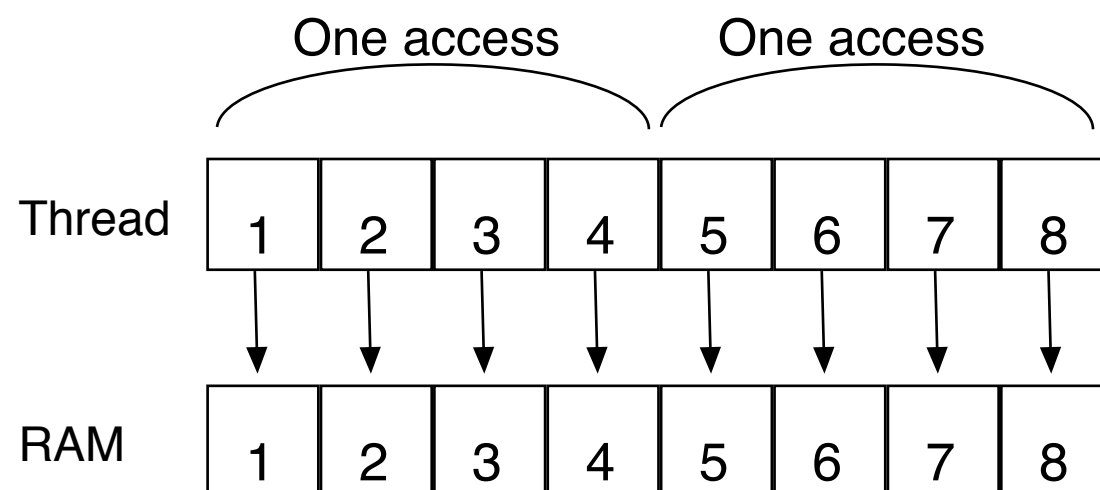
**Yes... You can access in any order you want, but ordered access *helps* the GPU to read more data in one access!**

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

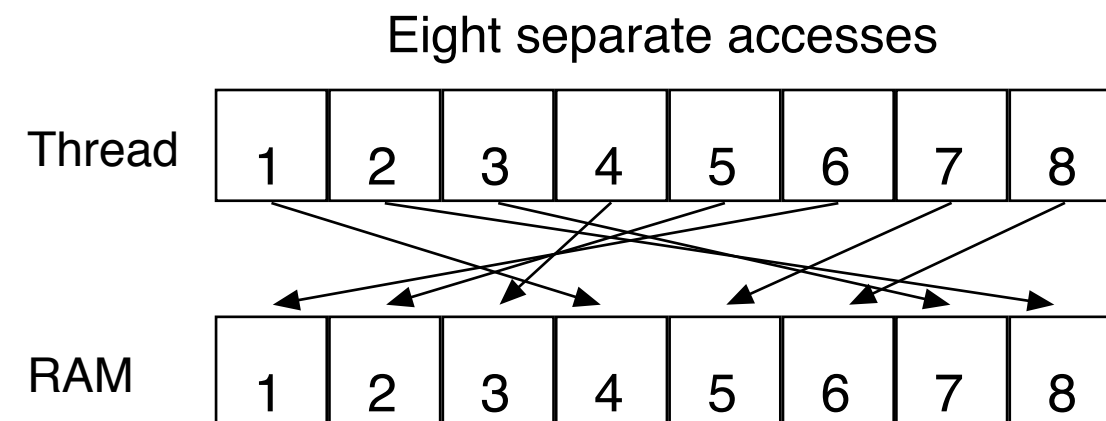


# Coalescing

**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.**

**"Perhaps the single most important performance consideration... is coalescing of global memory accesses." (CUDA C Best Practices Guide 2018)**



## **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];
    }
}
```

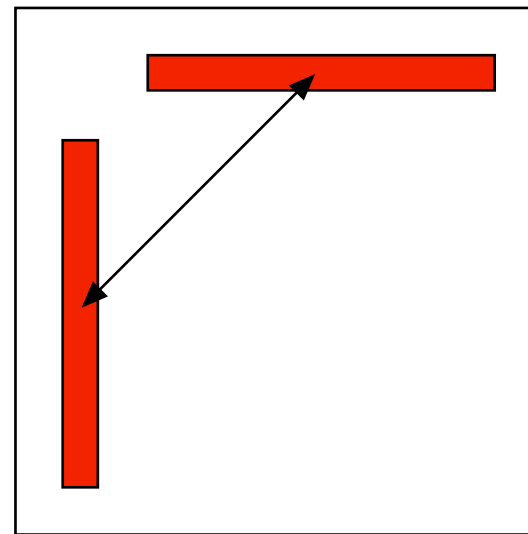
**How can this be bad?**





## Matrix transpose

### Coalescing problems

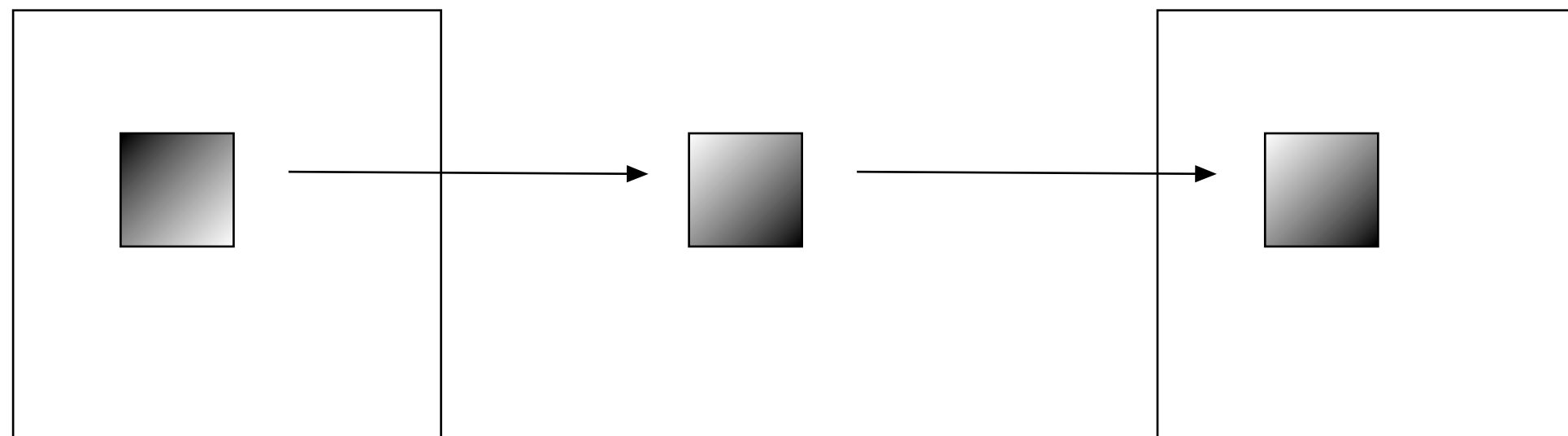


**Row-by-row and column-by-column.  
Column accesses non-coalesced!**



## 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 global 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**

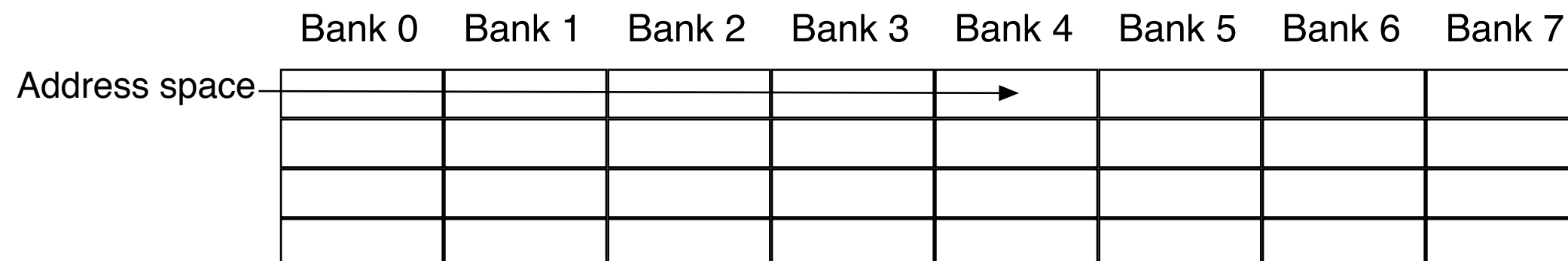


# Shared memory

**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**





## 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



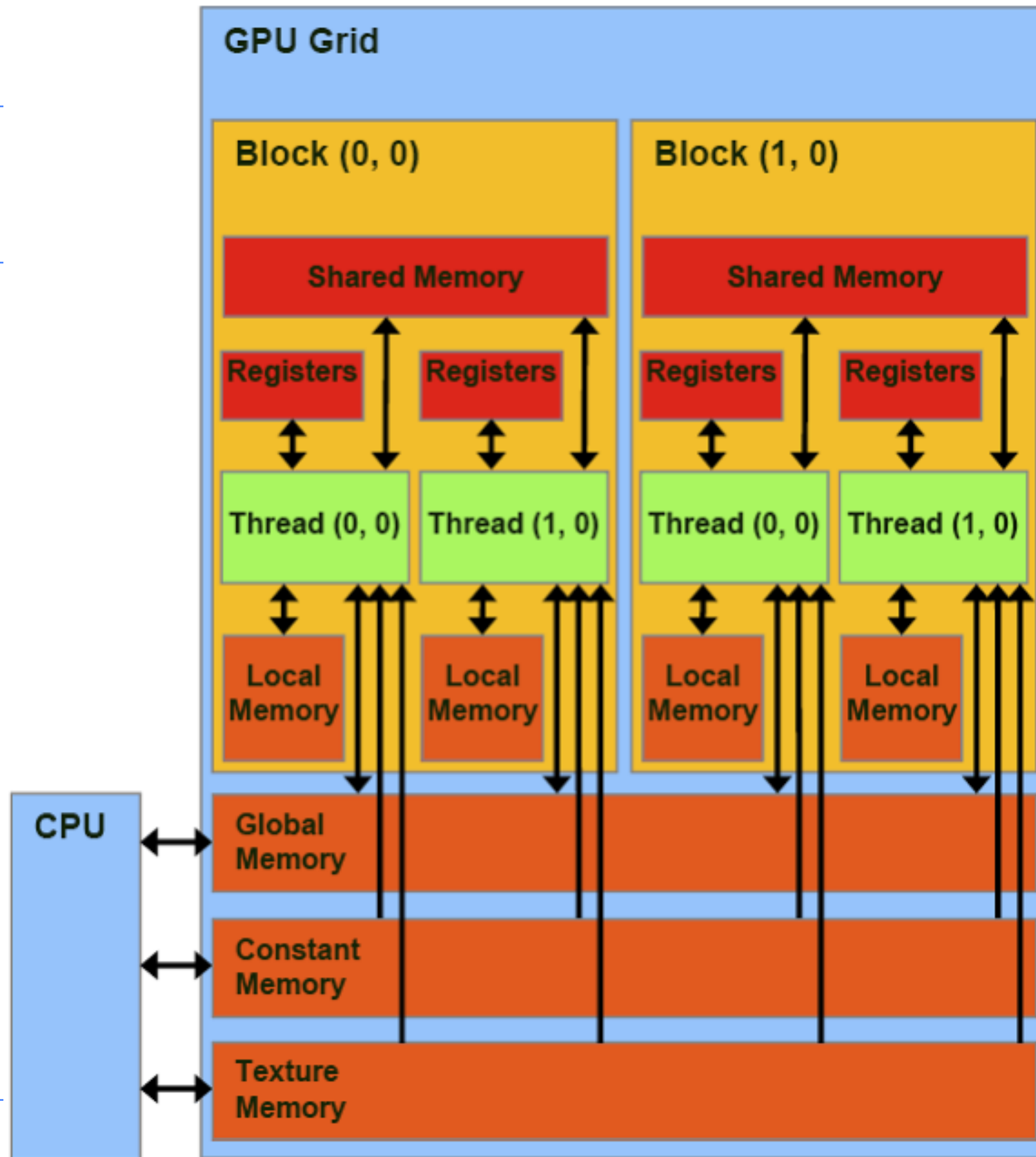

## **Constant memory**

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

**Read-only (for kernels)**

**`__constant__` modifier**

**Use for input data, obviously**







## **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)**

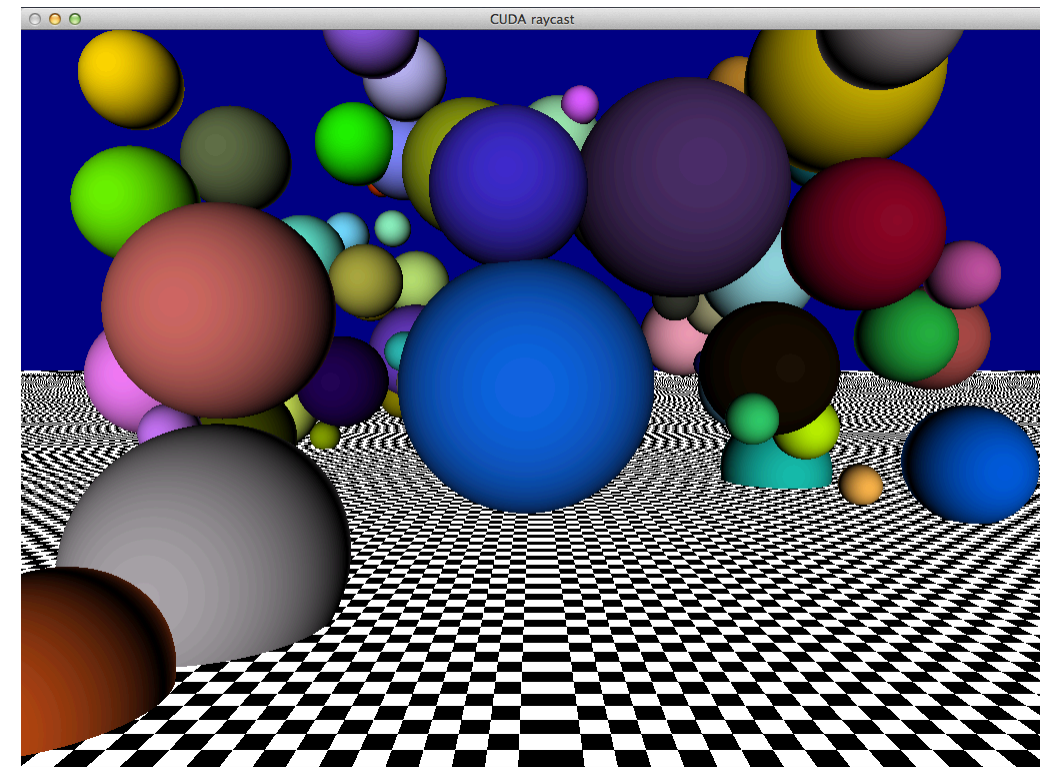
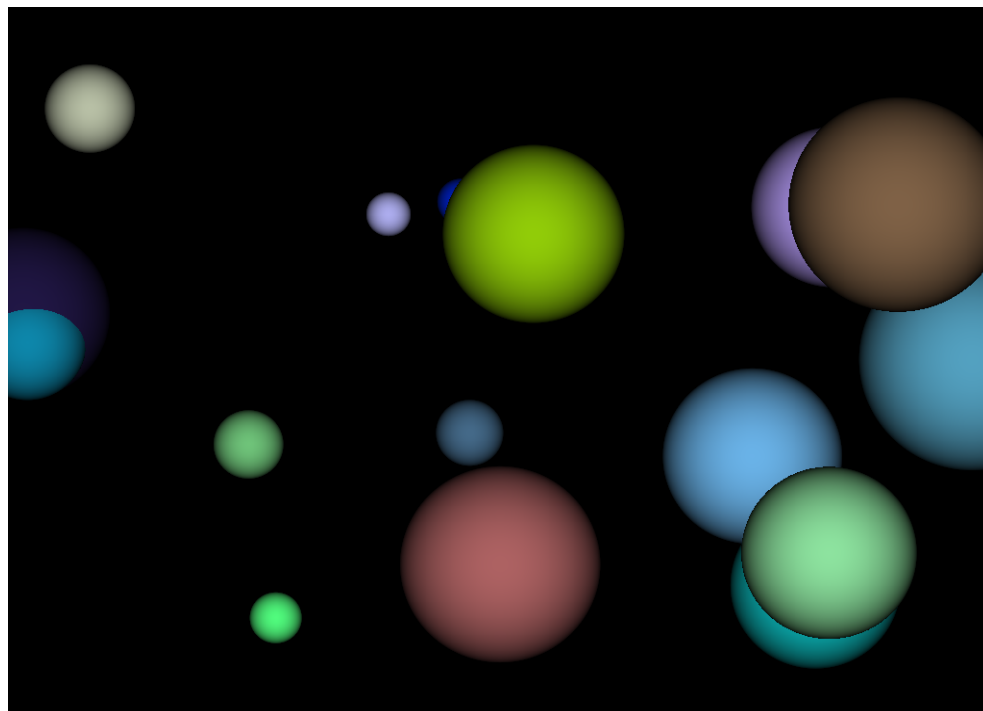
**But no help if threads are reading different data!**



## 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: 13.9 ms**

**With const: 10.6 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!**

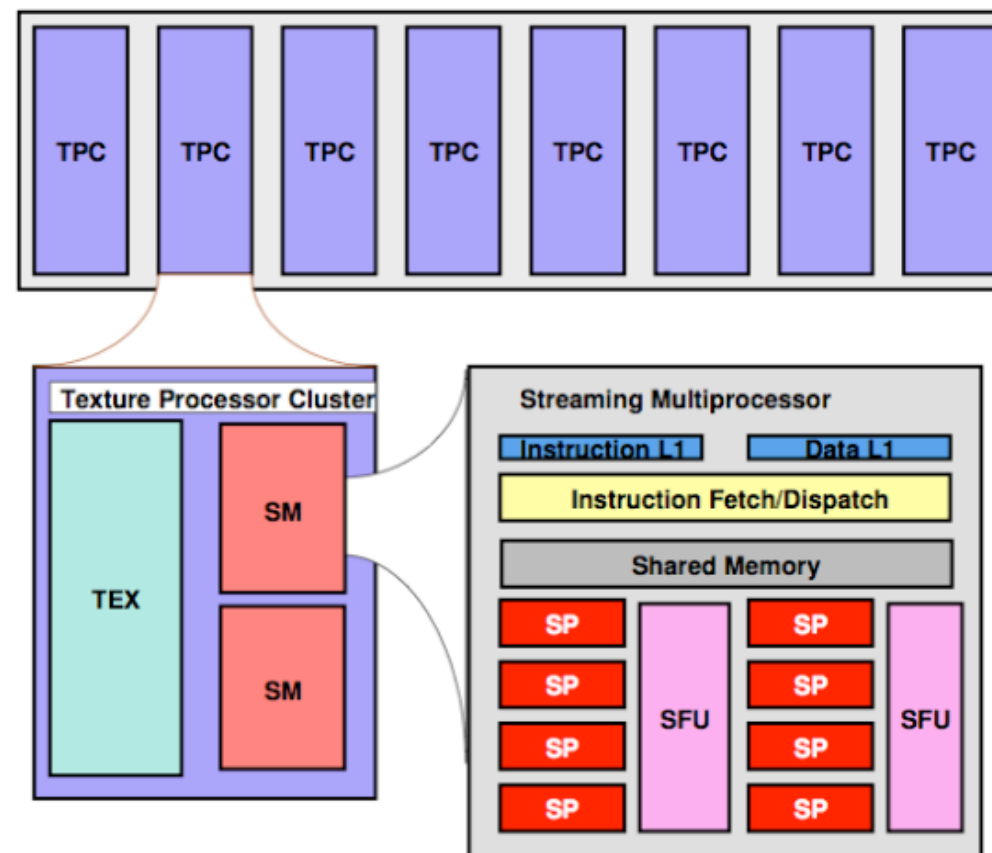
**Some advantage for code complexity.**

**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**

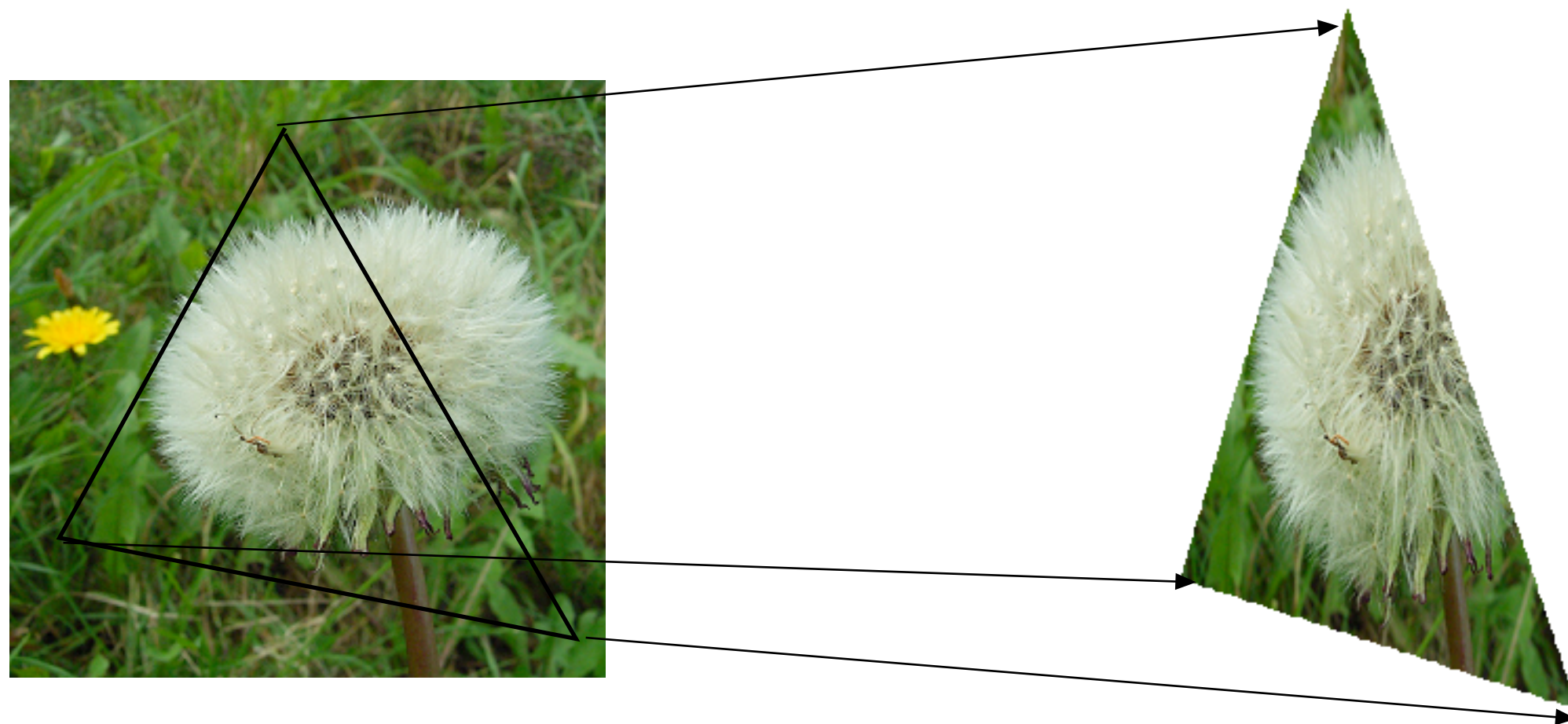
**Texture data mostly for rendering textures**

**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**

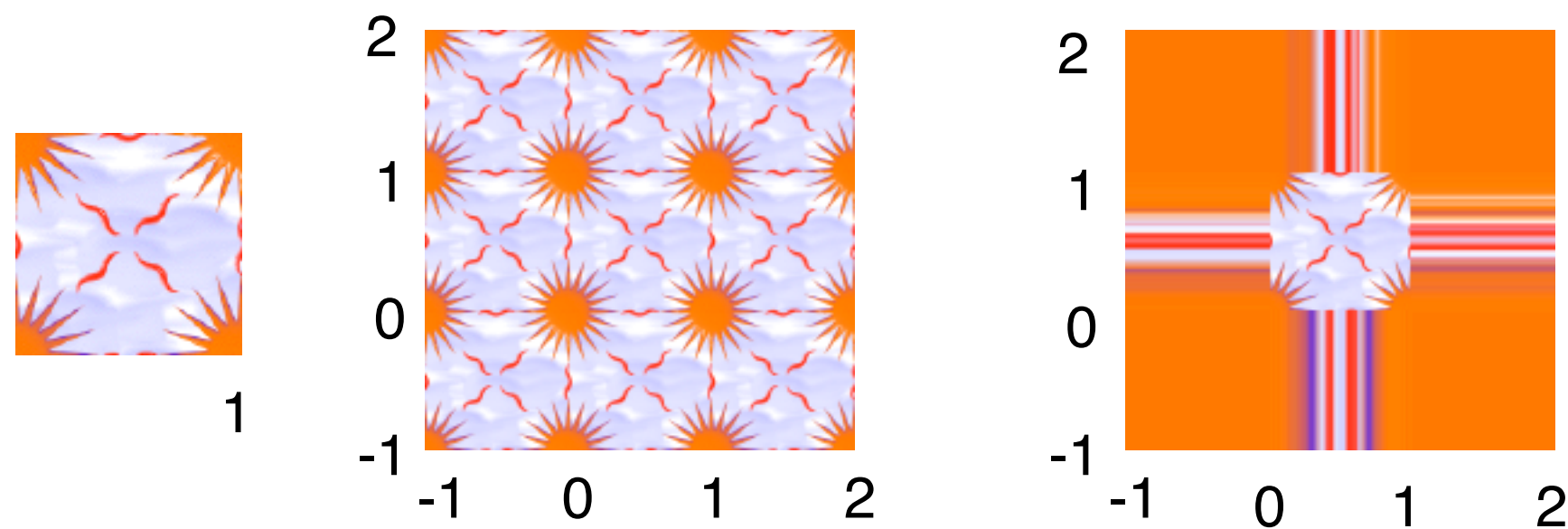
**Bind to texture unit using cudaBindTexture2D()**

**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

## Computation tricks when optimizing

Texture access provides hardware accelerated linear interpolation!

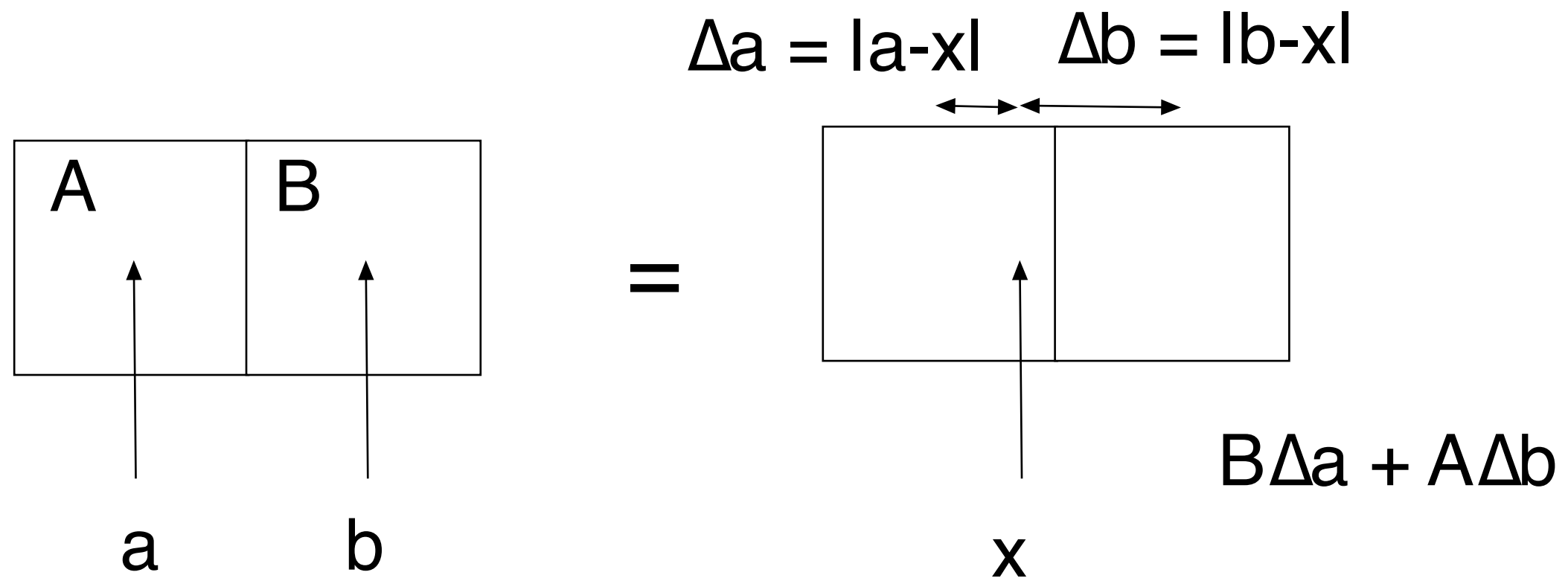
Access texture data on non-integer coordinates and the texture hardware will do linear interpolation automatically!

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





# Interpolation



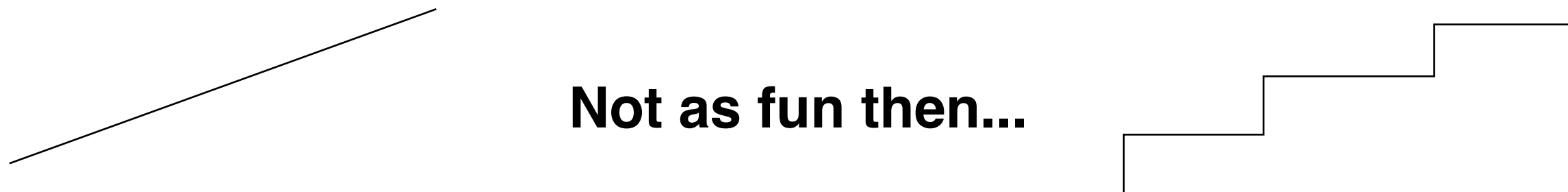
**Texture accesses and calculations hardware accelerated!**



## **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.**

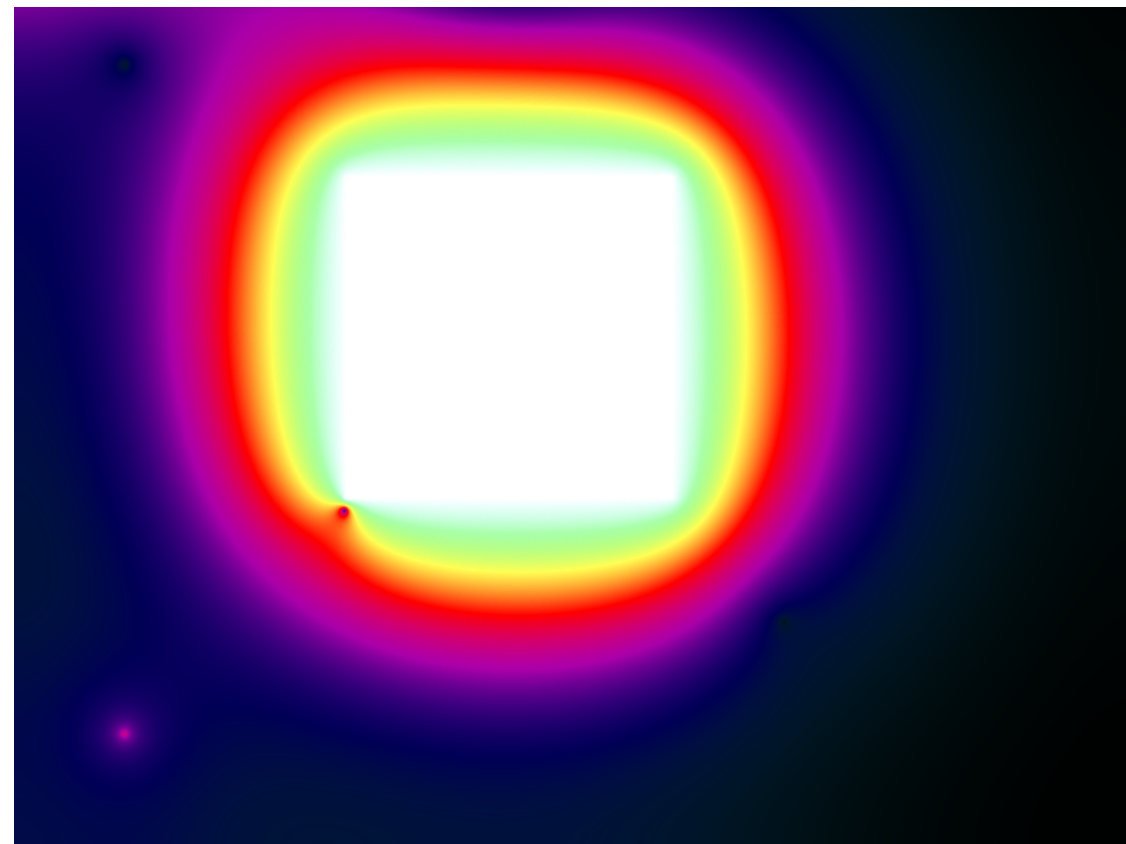
**Why? It is meant for interpolating between texels, visually. Small errors is not a problem then! May have low precision, like 10 steps.**





## Demo using texture memory

### Heat transfer demo





## Demo using texture memory

### Heat transfer demo

**Makes local operations modelling heat dissipation**

