CUDA for Graphics

Ville Timonen, 9.2.2010

Contents

- Motivation
- Use case: Height field shadowing
  - Design issues
  - Code preview

Why CUDA?

- Hardware resource exposition
- Less limitations; only those of hardware
- Means: (1) You can do (almost) whatever the hardware can, and (2) you can take shortcuts for epic performance

Why CUDA?

- For example:
  - Shared memory communication
  - Arbitrary memory access patterns

Why <u>not</u> CUDA?

- OpenGL does a lot for you, efficiently
  - Data alignment, coalescing (rasterization, vertex/pixel buffers)
  - Thread topology, optimal scheduling
- Only use CUDA when **you have to**
- Whatever OpenGL does, you probably cannot match with CUDA (perf. wise)

When do you have to?

- When your algorithm does not map to OpenGL shaders at all, and would otherwise have to do it in software
- When using OpenGL abstractions forces you to do things in an awkward (inefficient) way

Program for the architecture

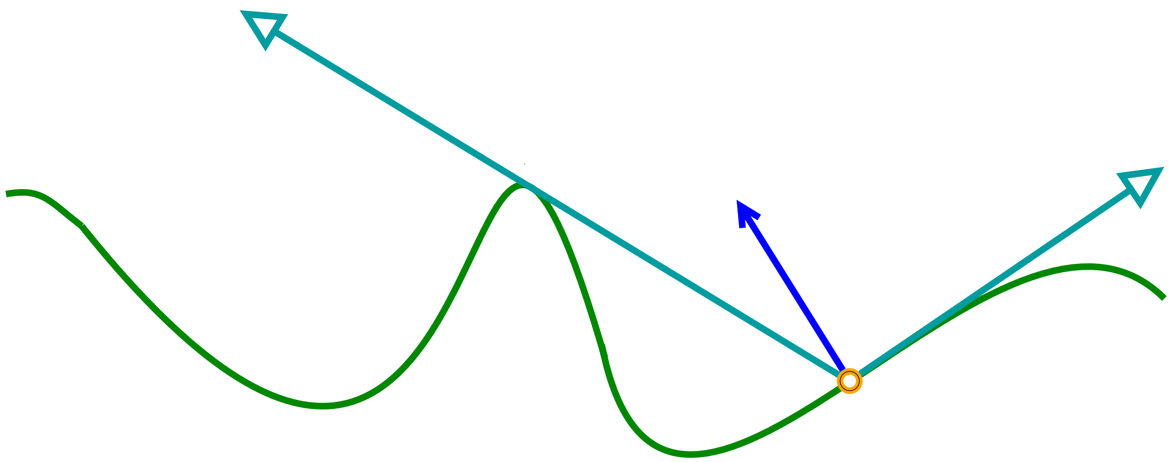
- With CPU code, you can *"program in C"*
- With GPUs, you *"program for G80"*
- Language (CUDA, OpenCL, ATI Stream (Brook+, Cal, ...)...) quite irrelevant

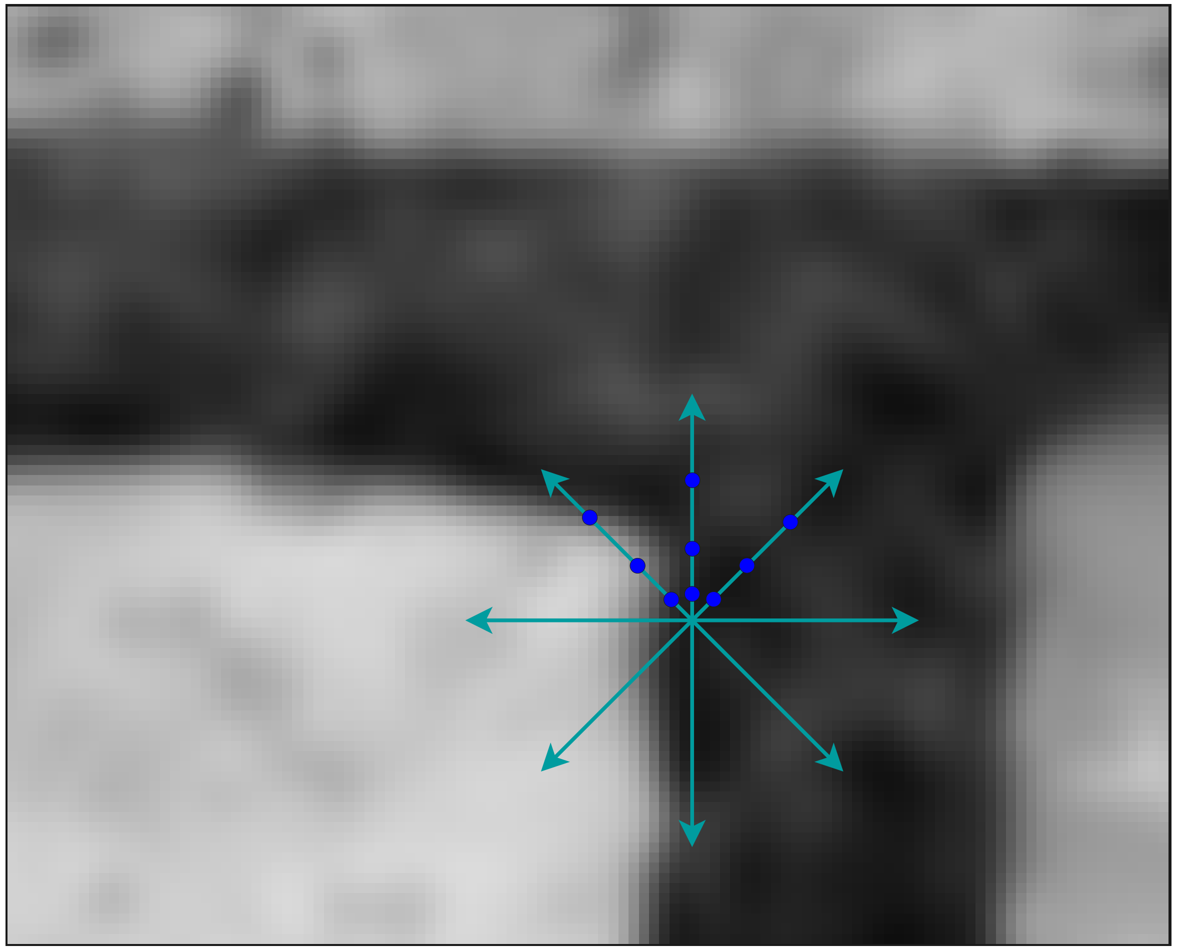
Design considerations

- Figure out input and output data
- In which memory to store data in CUDA
- Execution configuration
- Making the kernel efficient

Use case:
Height field shadowing

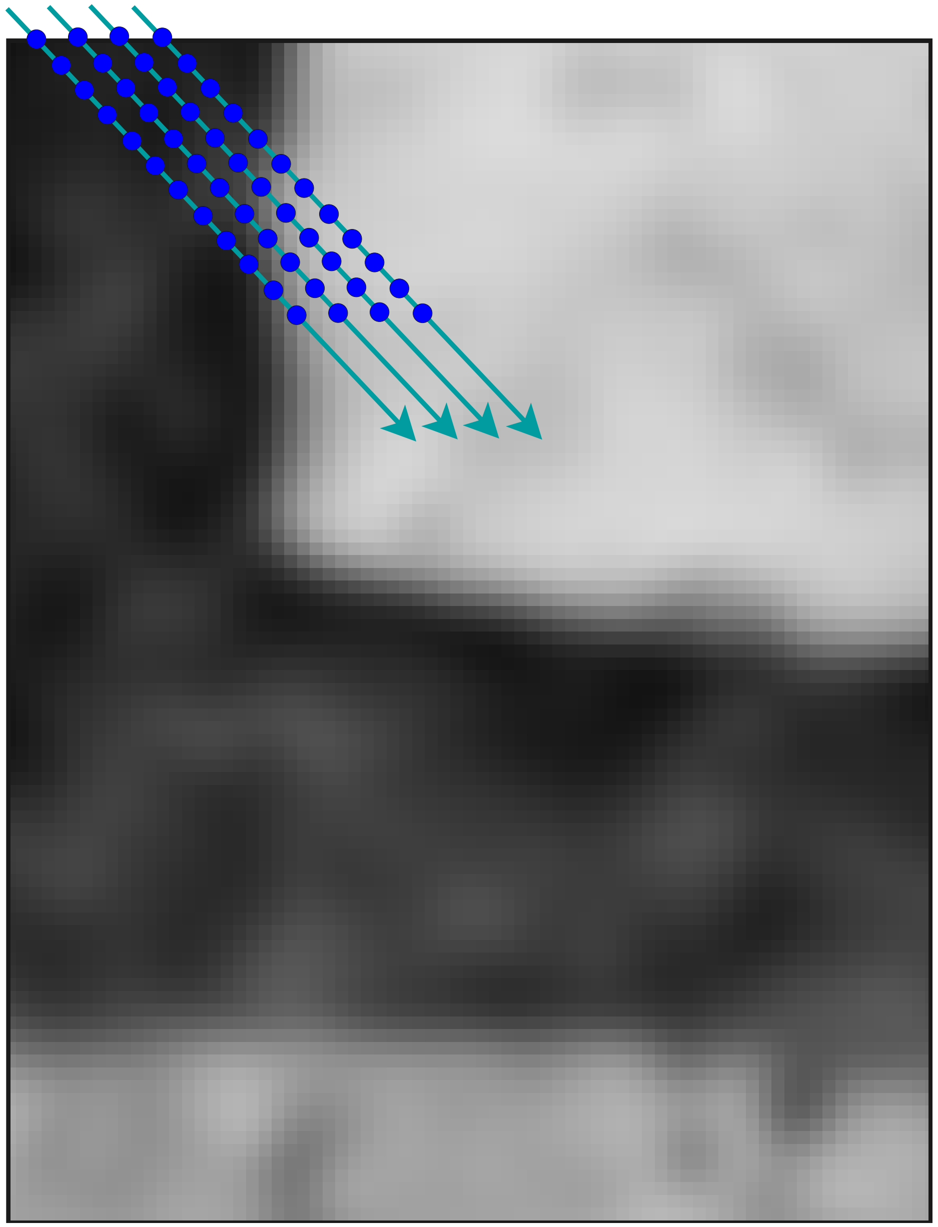
Self visibility

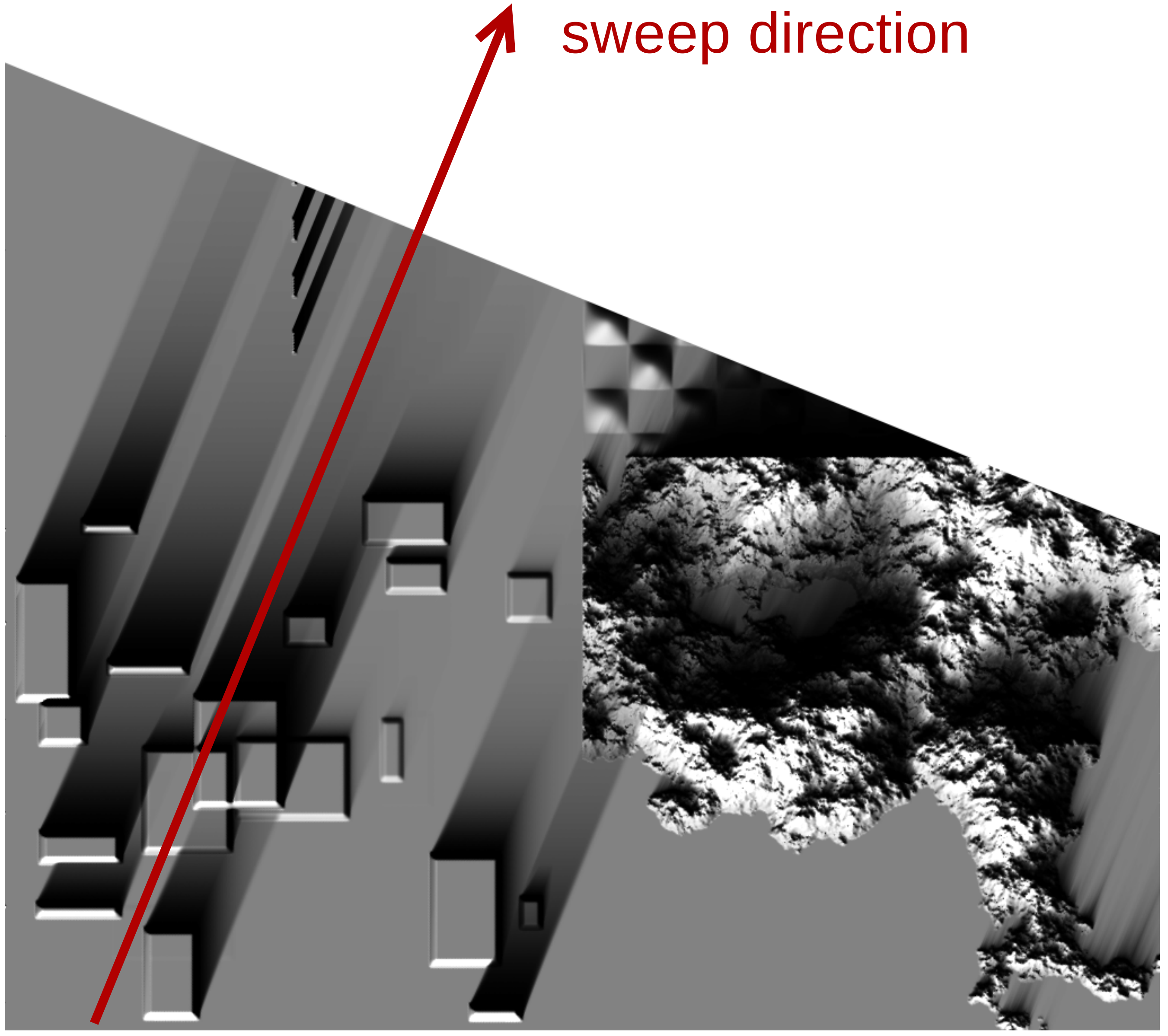




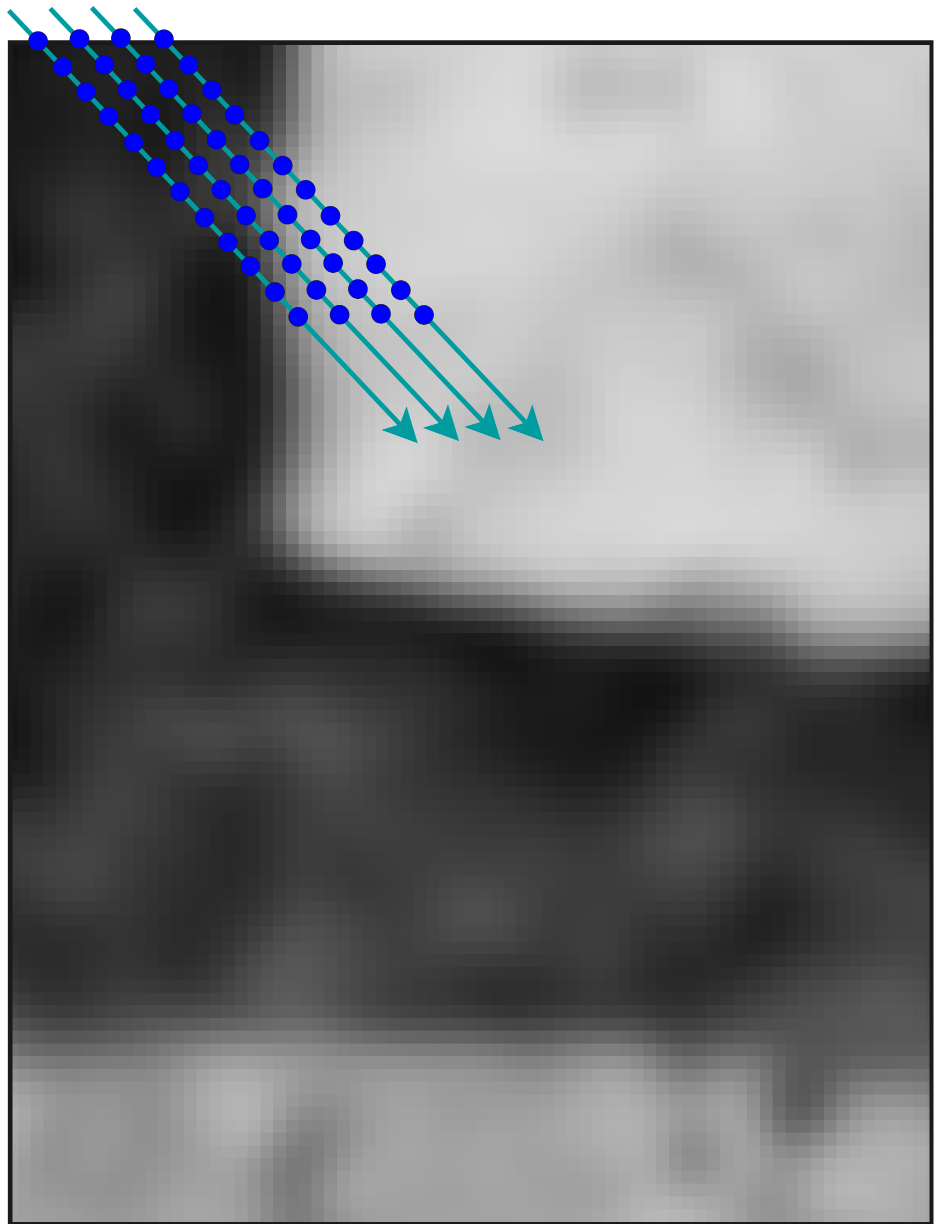
Screen space ambient occlusion (SSAO)

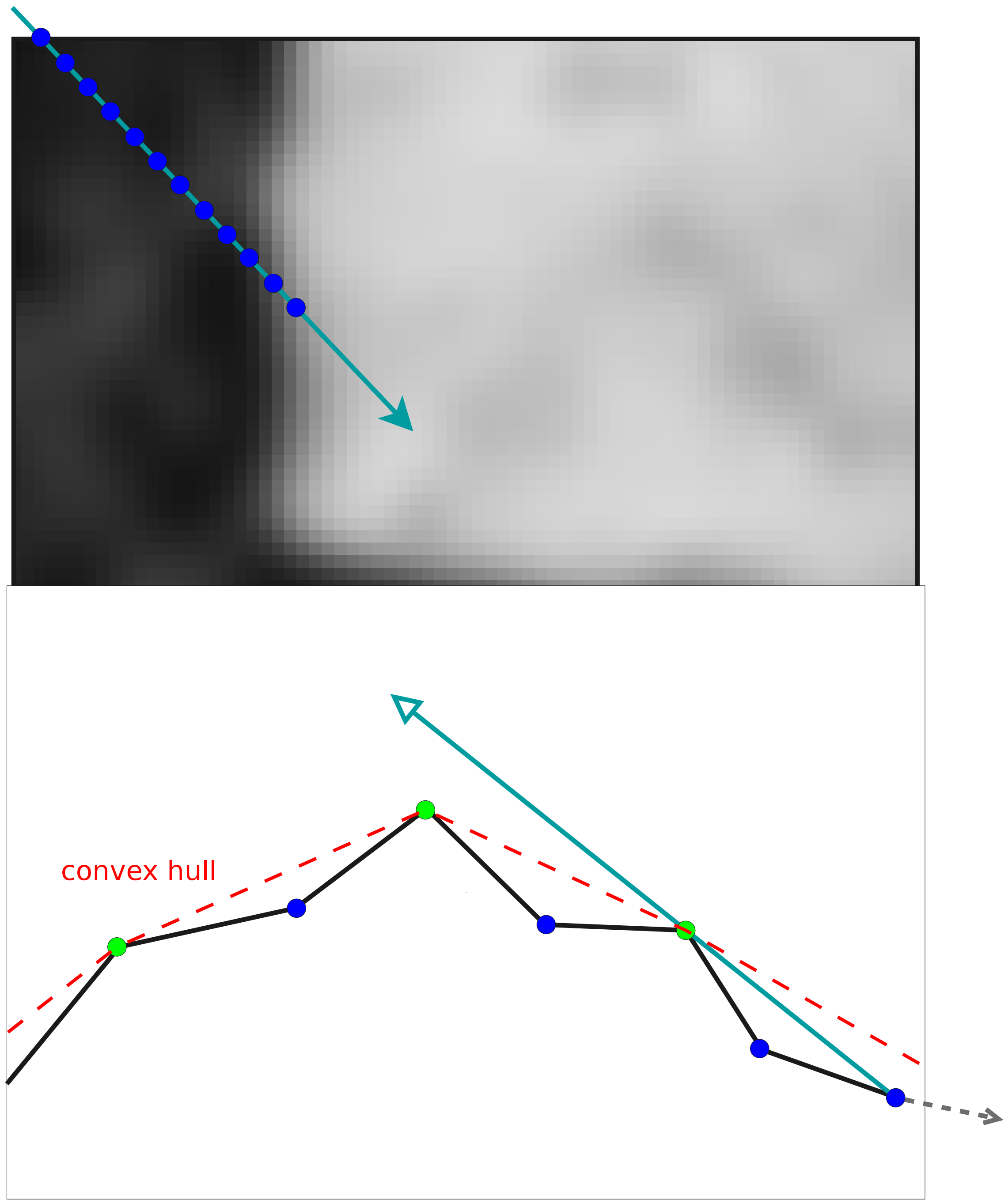
Sweeps





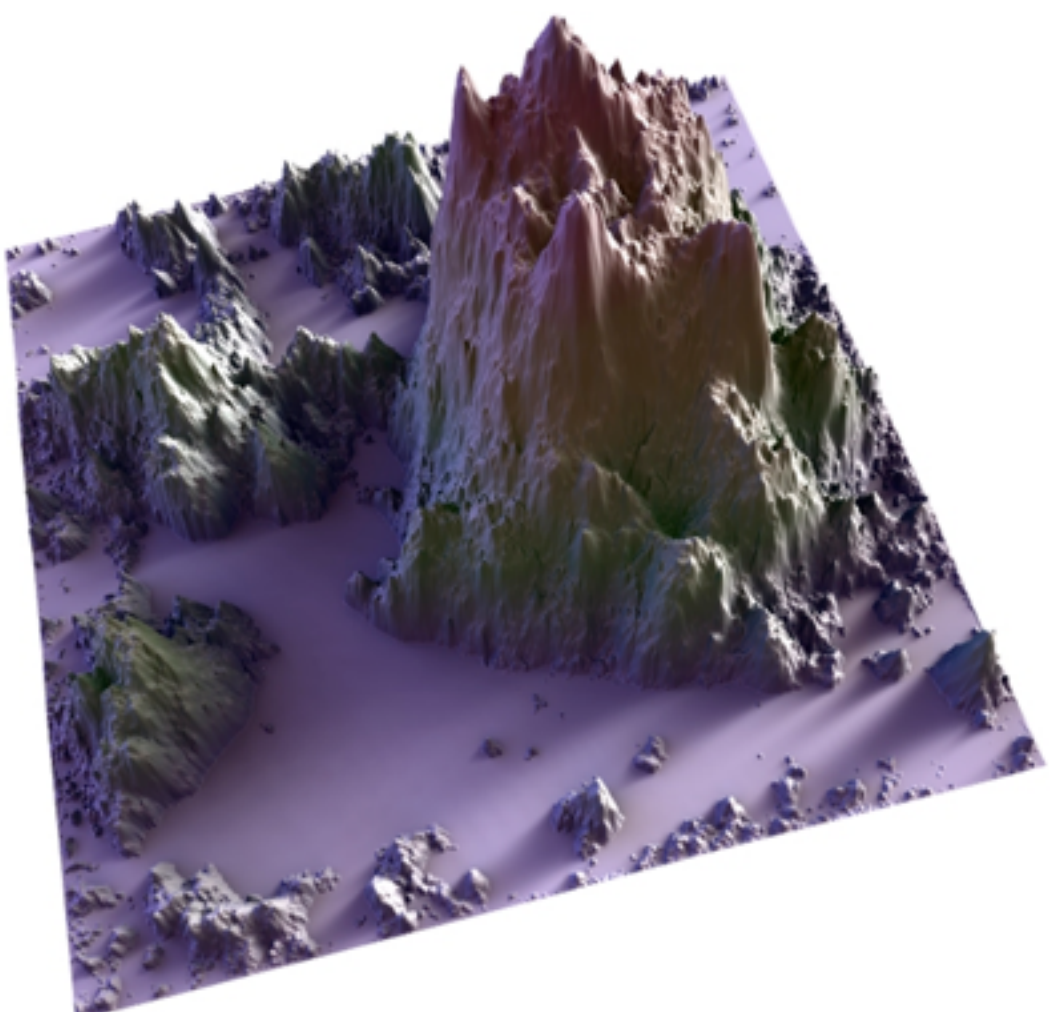
A thread for each line



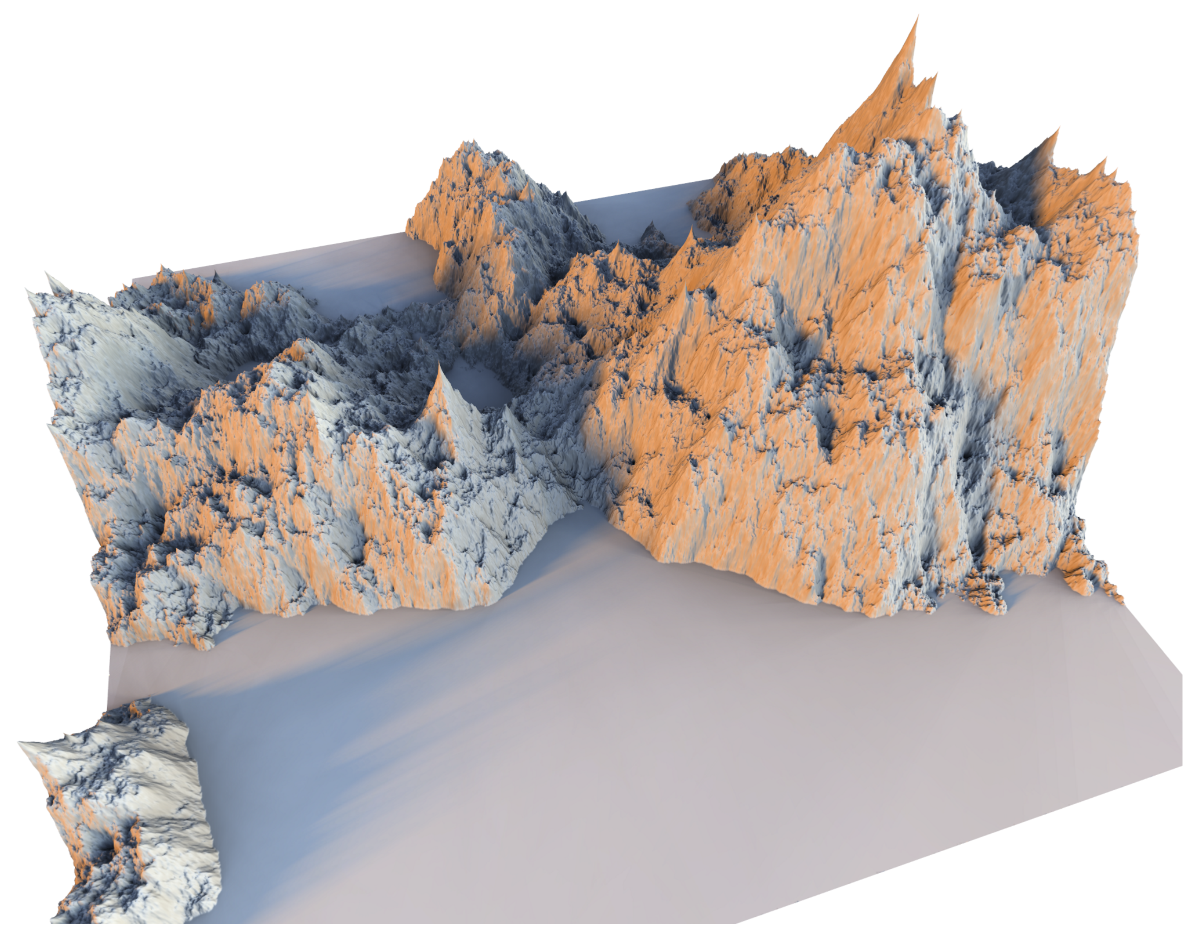


Teaser: Results

Time complexity drops from $O(n^3)$ to $O(n^2)$

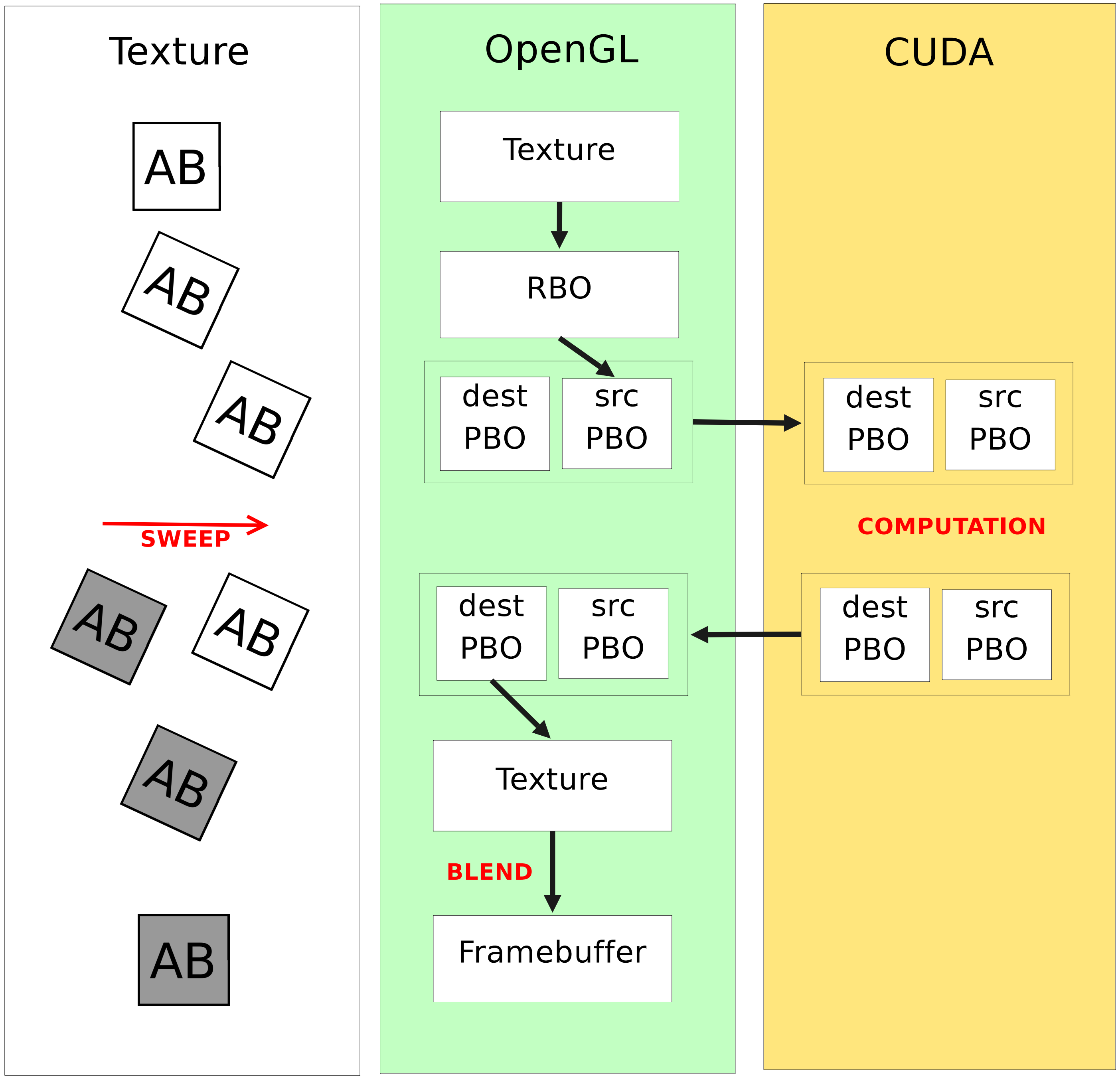


2,5 fps



38 fps

OpenGL <> CUDA

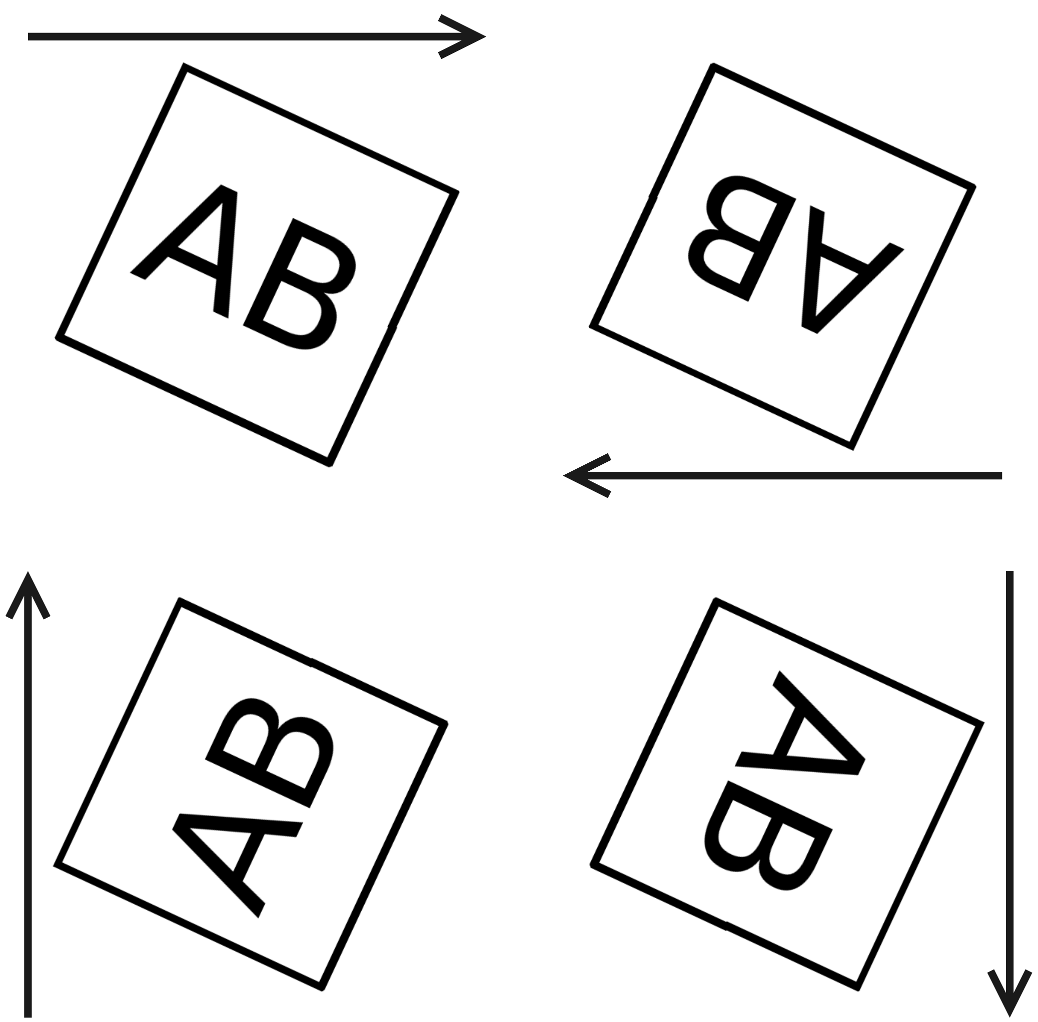


Texture
AB
AB
AB
SWEEP
AB
AB
AB
AB
OpenGL
Texture
RBO
dest PBO
src PBO
dest PBO
src PBO
Texture
BLEND
Framebuffer
CUDA
dest PBO
src PBO
COMPUTATION
dest PBO
src PBO

Global (off-chip) memory

- When passing a buffer object to CUDA, it gets mapped as *linear memory*
- There's no caching whatsoever with linear memory, you have to be careful with it
- Use *CUDA arrays* when in doubt - most of the on-board caching is for texture sampling (in current architectures)

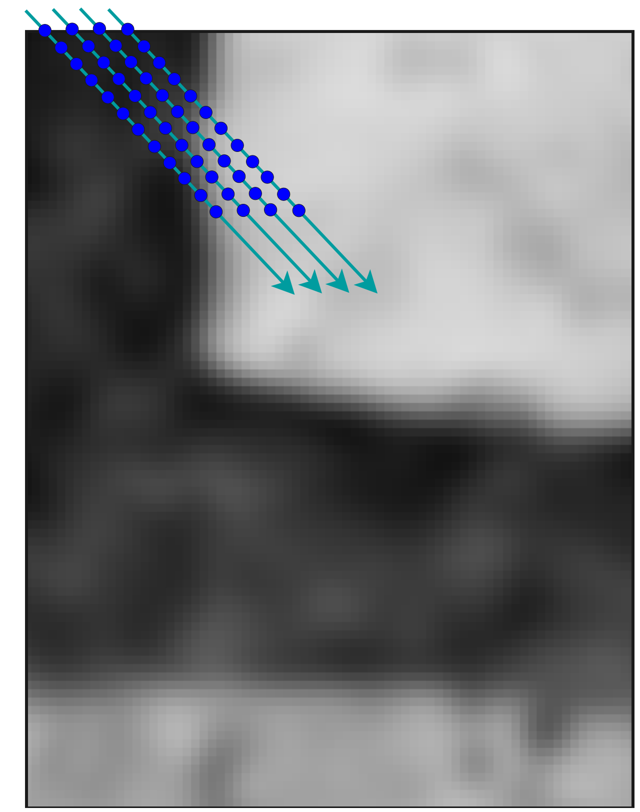
Using SM for efficient texture transposing

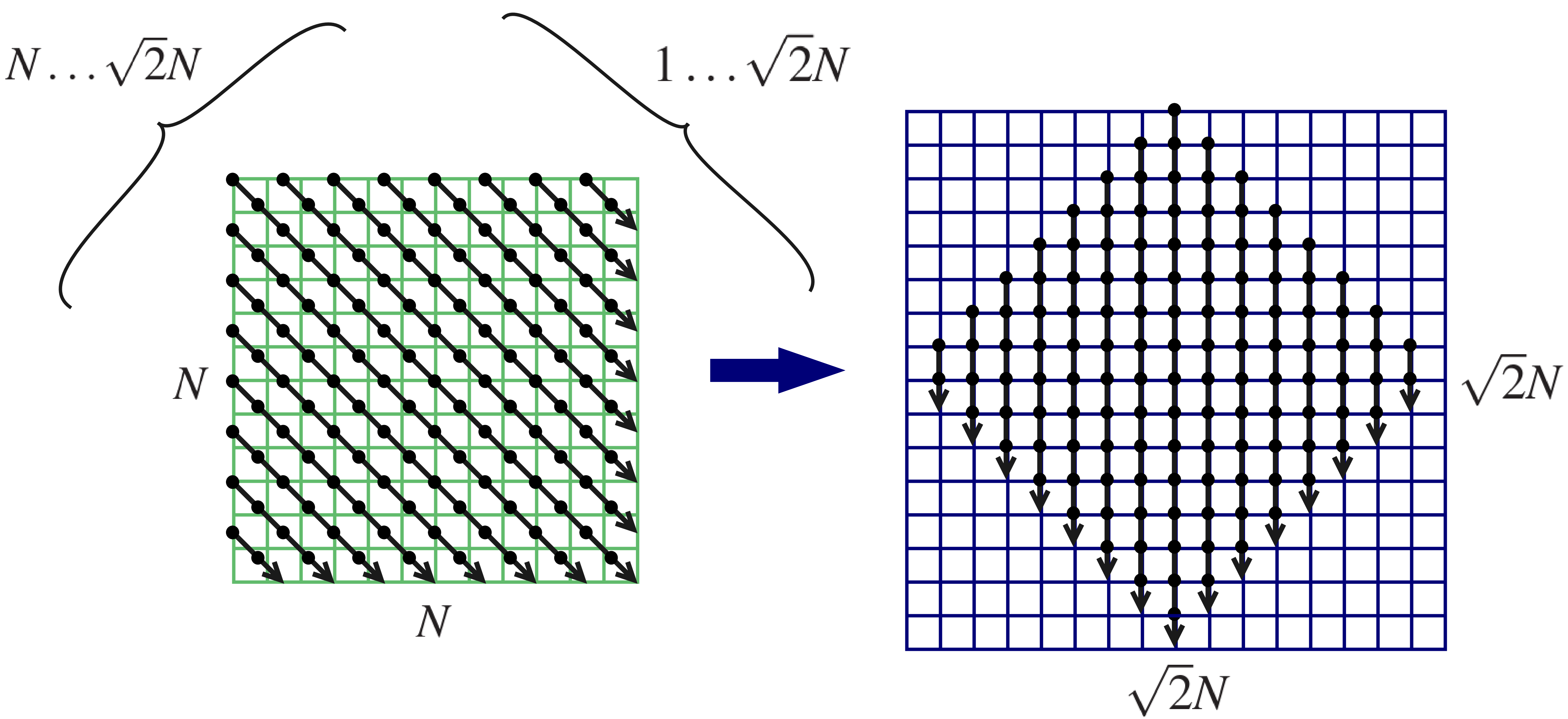


Shared memory

- The most important means for thread communication
- Such communication cannot be carried out in OpenGL shaders
- Shared memory is also very fast, and can be used for acceleration

Sweeps and threads





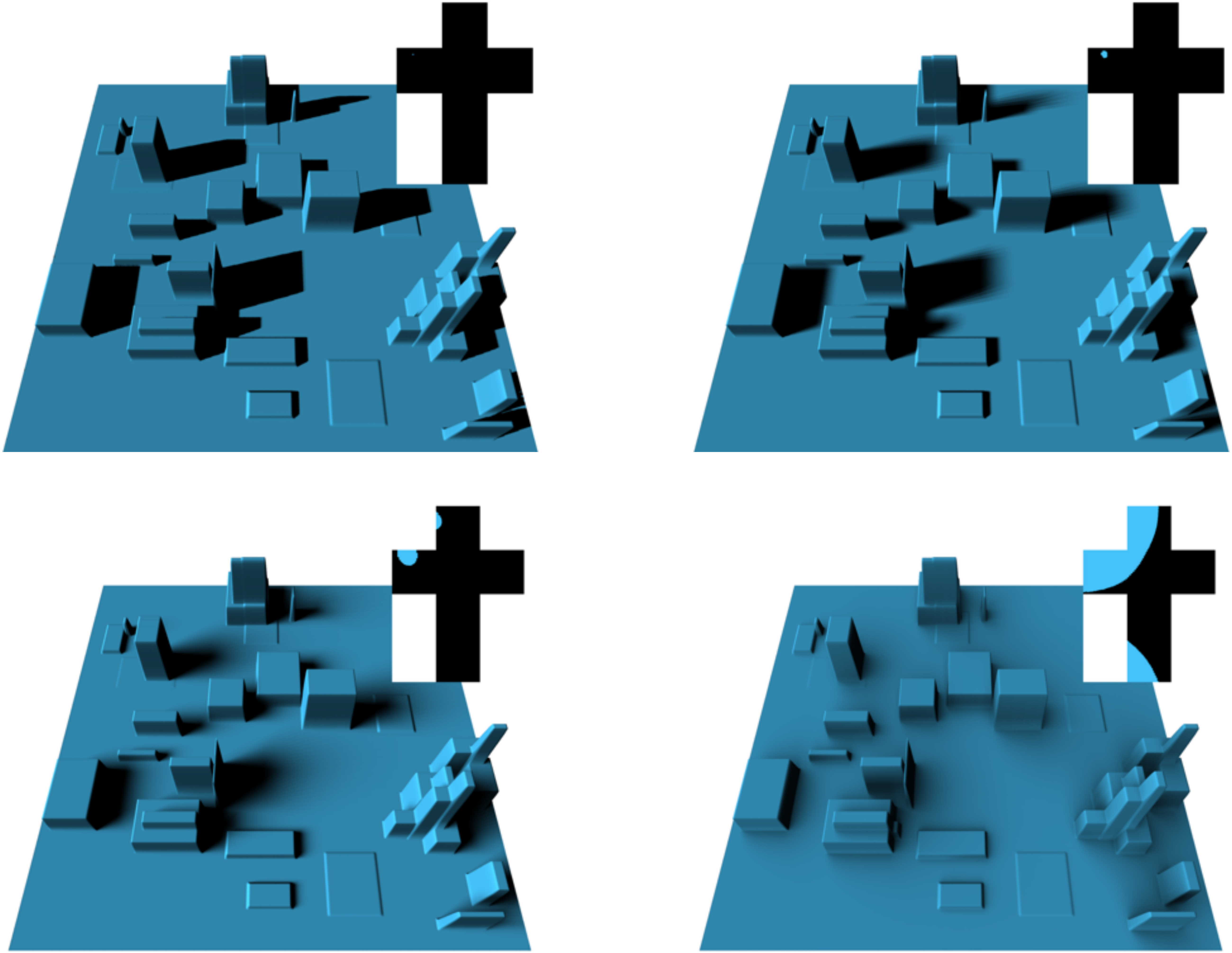
Thread block size

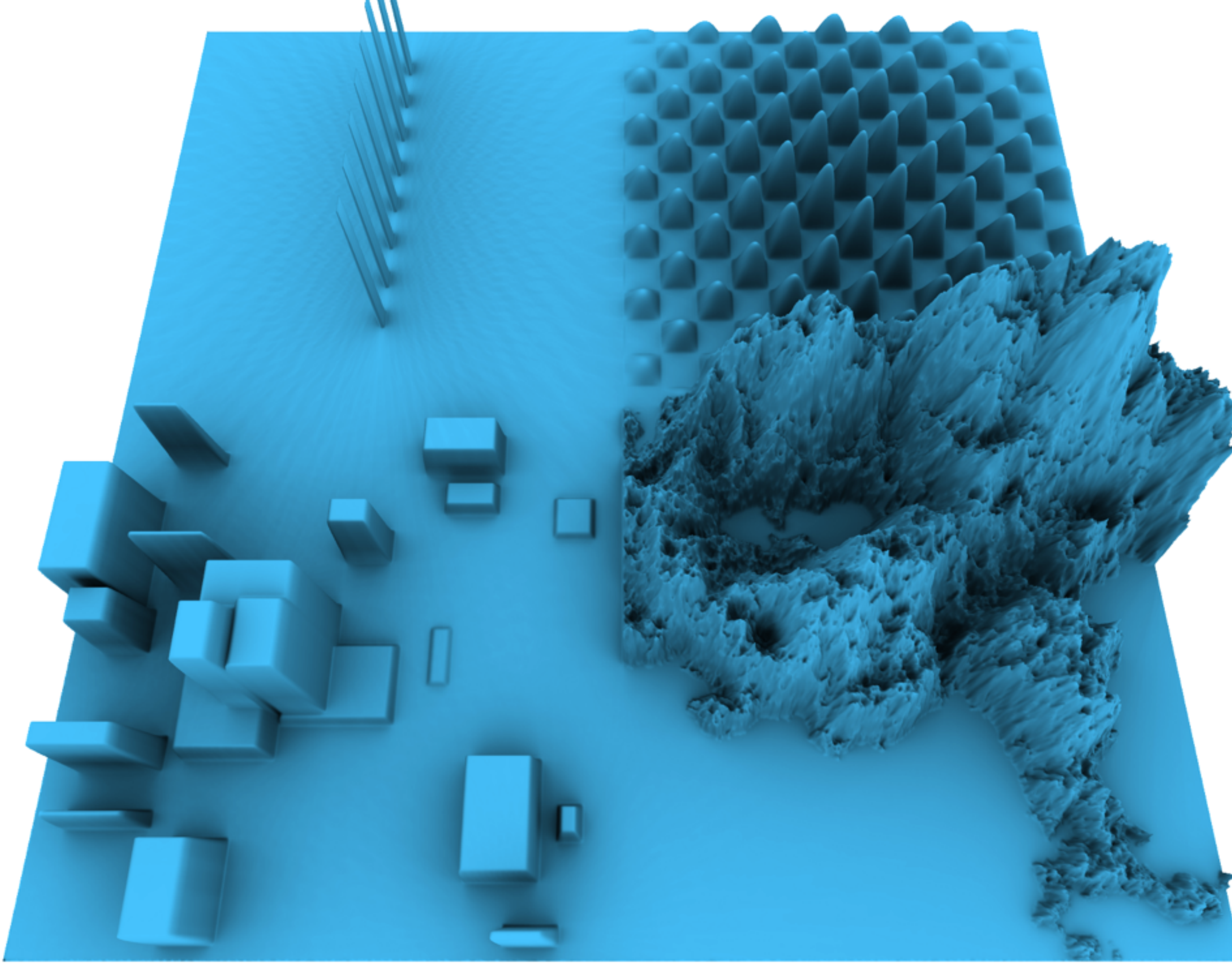
- At least a multiple of 32 (NVidia recommends 64) for max. utilization
- Not too big though, leave room for the scheduler to do its magic
- If threads execute different lengths, prefer small blocks for finer granularity
- Keep an eye on core resources (e.g. SM)
- Make your kernel flexible and experiment!

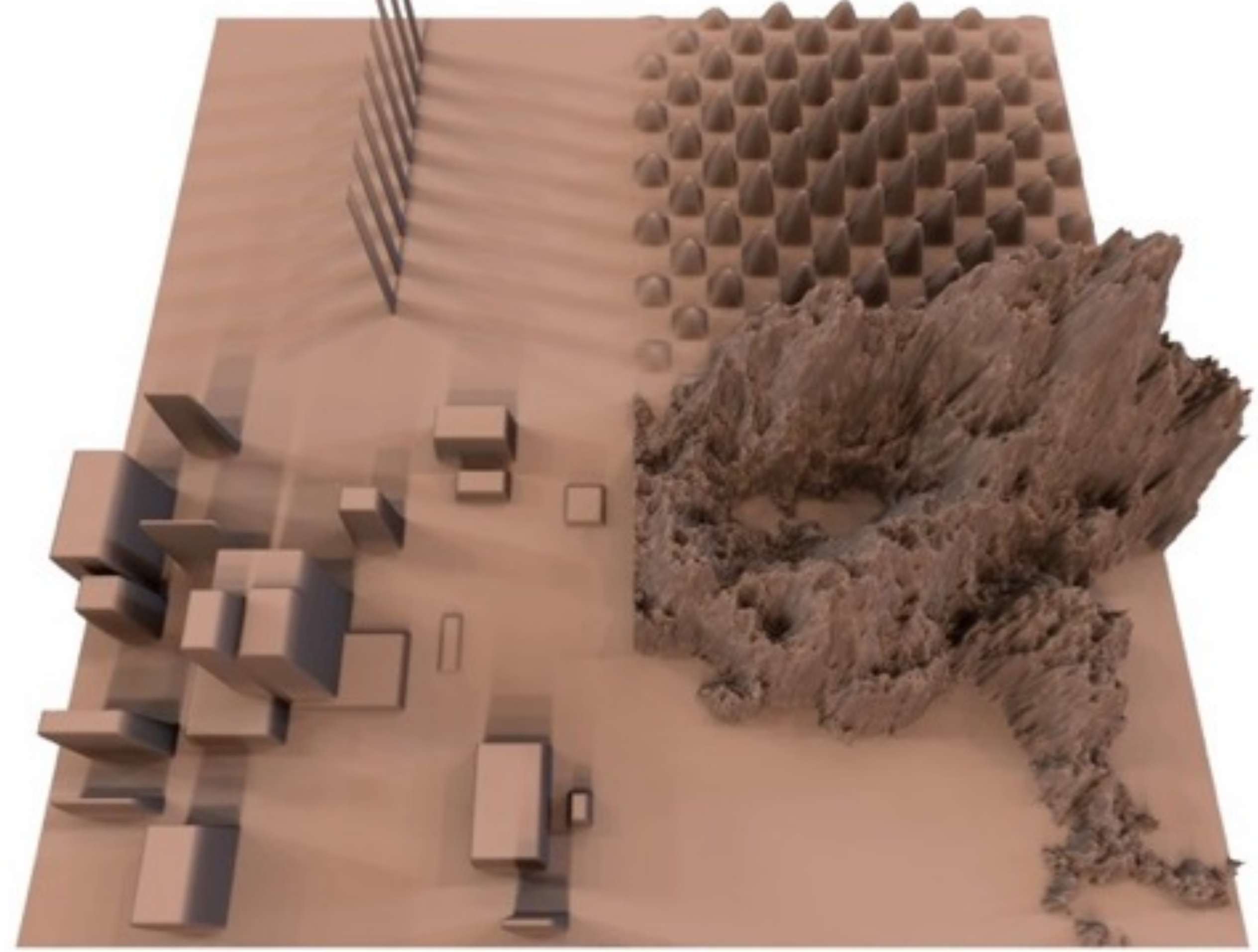
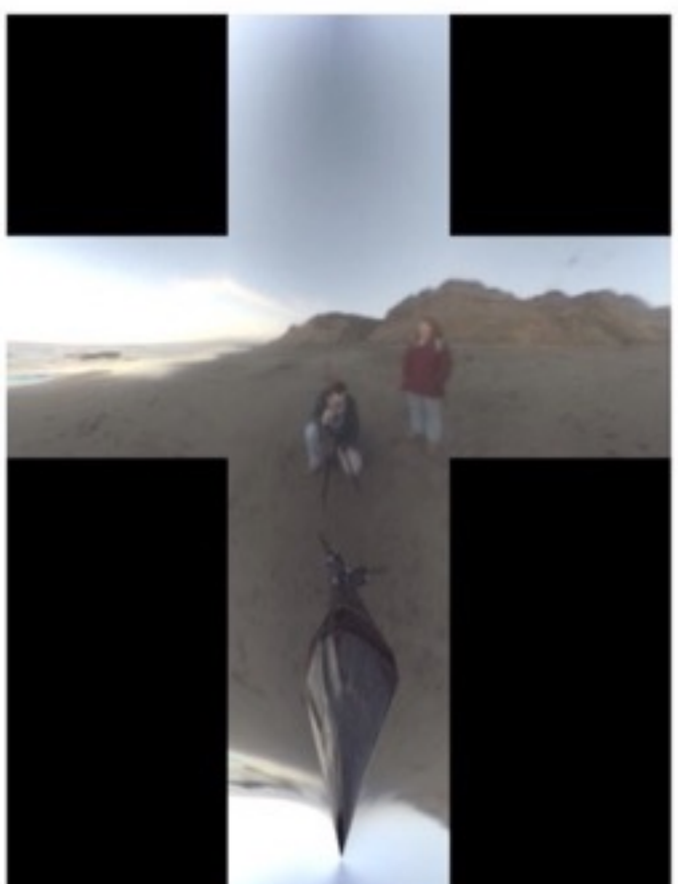
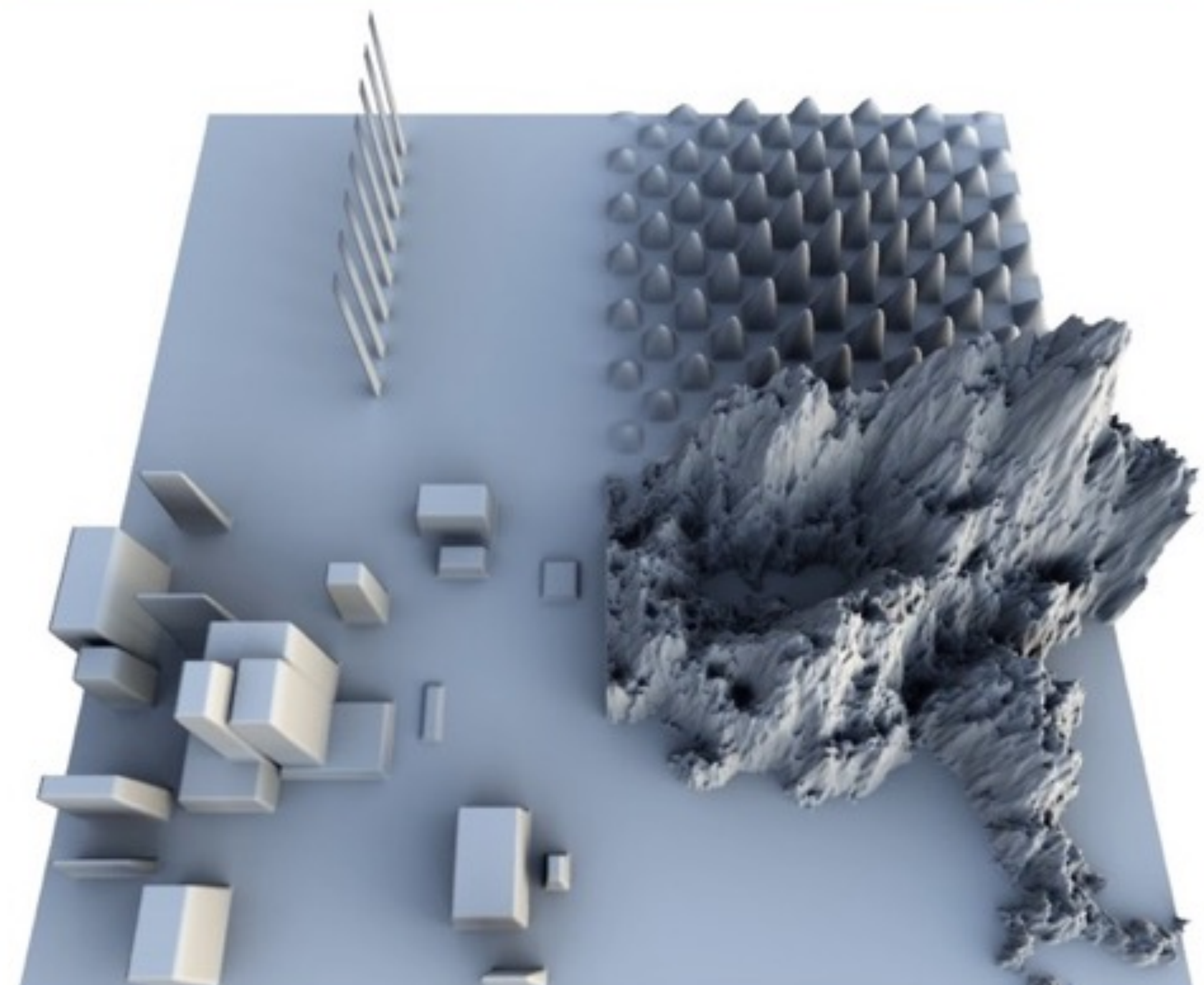
Grid size

- The bigger the better
- If you fix the thread block size, you rarely can affect this
- Take into account the number of cores
- Graphics hardware relies on lightweight scheduling; make sure each core has lots of threads to choose the work from and it will fly

More screencaps





teh end.

- I'm on the fourth floor if you have something to ask