3D Bilateral Filtering on the GPU

E. Wes Bethel

13 April 2010

Lawrence Berkeley National Laboratory

Outline

What is Bilateral Filtering?

CUDA Background

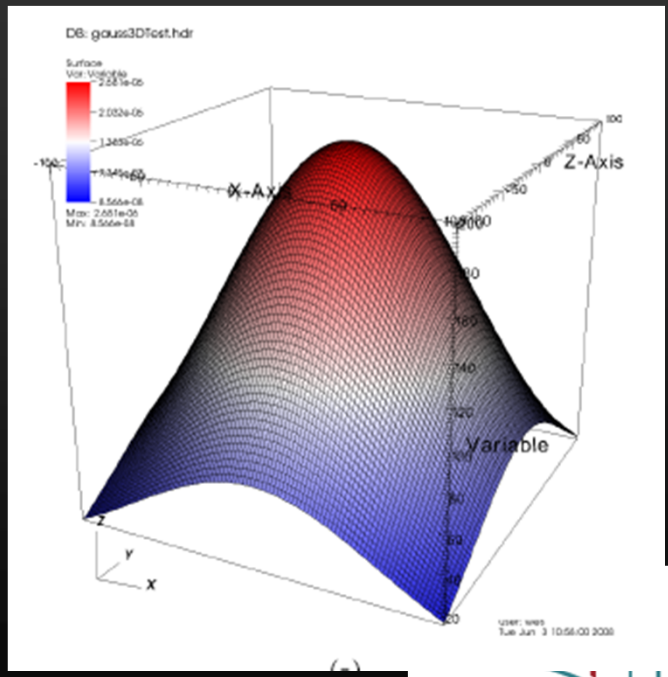
GPU implementation project objectives.

The implementation, performance evaluation, optimization: algorithmic design choices, tunable algorithm parameters.

Gaussian Smoothing

- Convolution kernel, a *stencil-based algorithm.*
- Weights are a 2D Gaussian (right).
- Idea: nearby pixels have more influence, distant pixels have less influence.

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$



BERKELEY LAB

Bilateral Filtering/Smoothing

- Dest pixel *i* is the sum of:
- Gaussian weight of nearby pixel *i*
- "Photometric difference" between pixel *i* and pixel *i*
- Normalization constant *k* – *c* weights are data dependent.

$$d(i) = \frac{1}{k(i)} \sum g(i, \bar{i}) c(i, \bar{i})$$

$$g(i, \bar{i}) = e^{-\frac{1}{2}\left(\frac{d(i,\bar{i})}{\sigma_d}\right)^2}$$

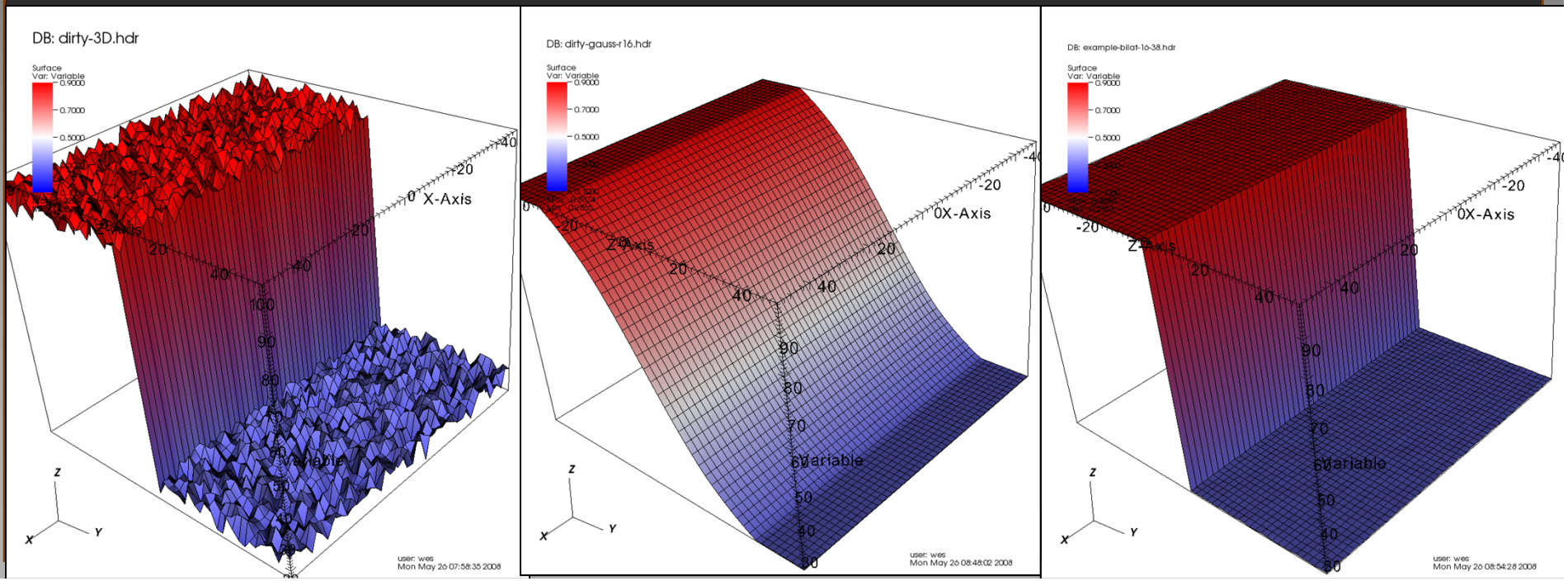
$$k(i) = \frac{1}{\sum g(i, \bar{i}) c(i, \bar{i})}$$

Comparison of Bilateral and Gaussian Smoothing

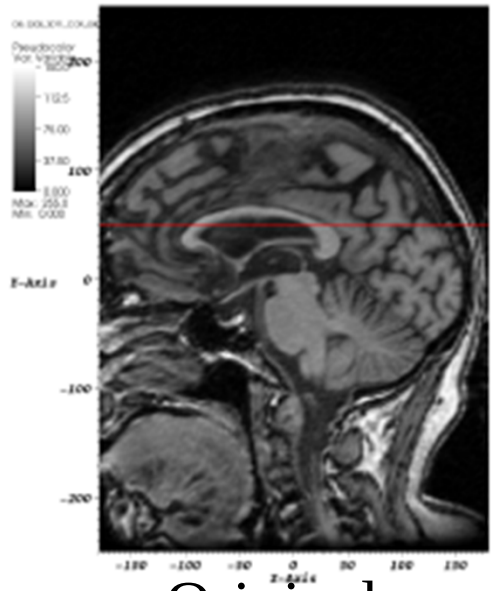
Synthetic data with gaussian noise

Gaussian smoothing

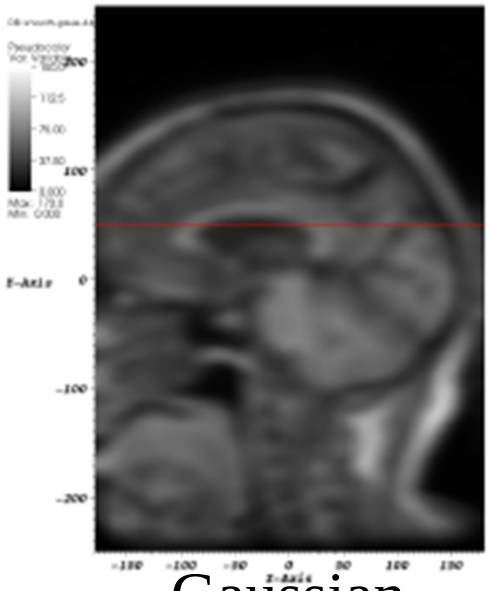
Bilateral smoothing



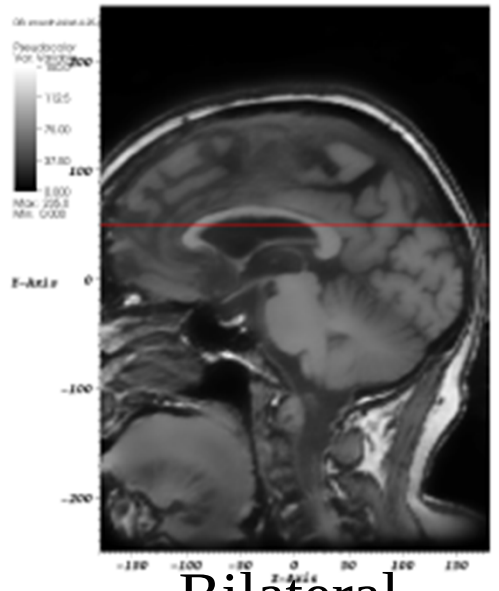
Comparison of Bilateral and Gaussian Smoothing



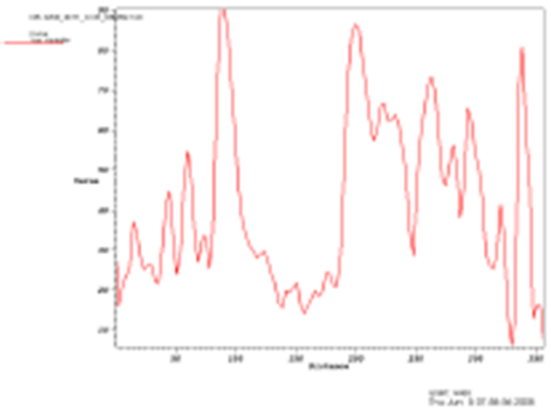
Original

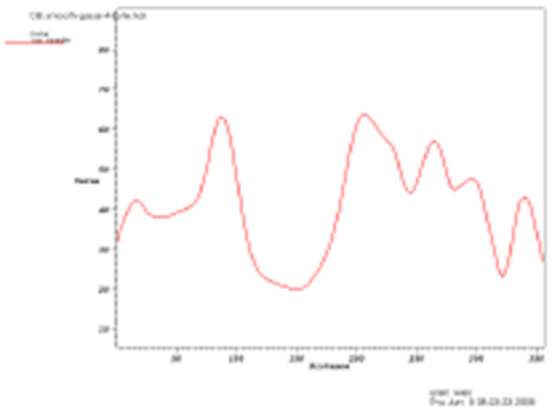


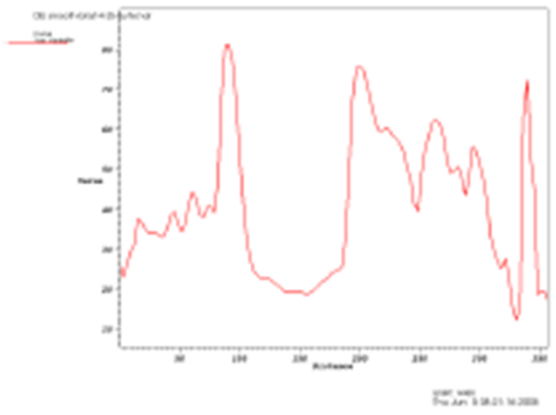
Gaussian



Bilateral







Why Bother with GPU Implementation?

- **This algorithm is compute-bound for large filter radii.**
- **Long run-times:**
  - R=8, ~8min, R=16, ~60min.
- **Data parallel algorithm, non-iterative.**

GPU Implementation Objectives

- **Gain experience developing in CUDA**
- **Performance optimization**
  - Algorithmic design choices: device memories and access patterns.
  - Tunable parameters: thread block size/shape

CUDA Background

- **Data parallel programming language:**
  - Eg., A[I] = B[I] + C[I]
  - Runs in parallel on all cores on the GPU.
    - GeForce GTX 280: 30 “multi-processors”, 8 cores/MP, 240 cores total.
- **Requires GPU code and host code (next slides)**

Example: Vector Addition Kernel

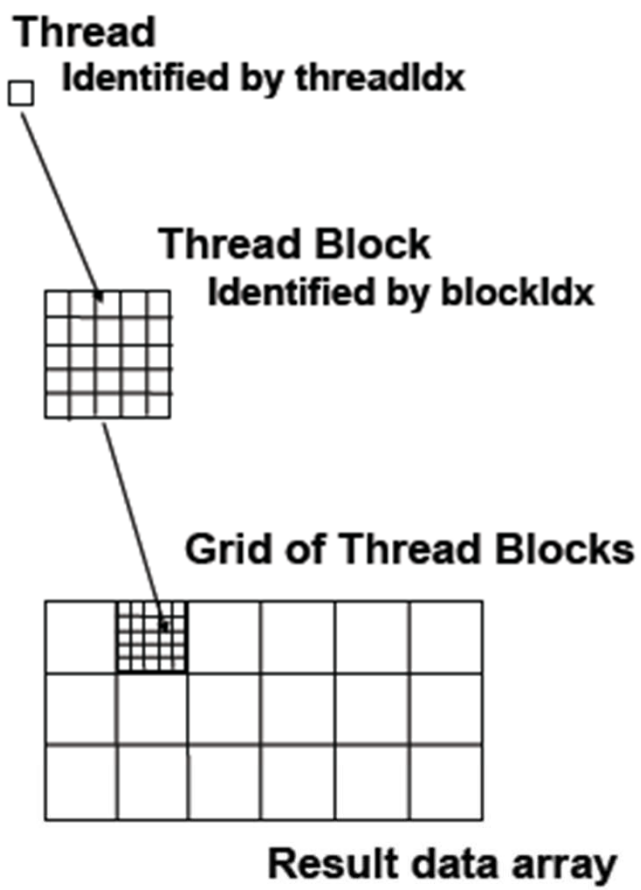

```
// Pair-wise addition of vector elements
// One thread per addition

__global__ void
vectorAdd(float* iA, float* iB, float* oC)
{
    int idx = threadIdx.x
        + blockDim.x * blockId.x;
    oC[idx] = iA[idx] + iB[idx];
}
```

Example: Vector Addition Host Code


```
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
// … initalize h_A and h_B

// allocate device memory
float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float)));
cudaMalloc( (void**) &d_B, N * sizeof(float)));
cudaMalloc( (void**) &d_C, N * sizeof(float)));

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
    cudaMemcpyHostToDevice) );
cudaMemcpy( d_B, h_B, N * sizeof(float),
    cudaMemcpyHostToDevice) );

// execute the kernel on N/256 blocks of 256 threads each
vectorAdd<<< N/256, 256>>>( d_A, d_B, d_C);
```


<<<nblocks, nthreads>>>

Execution Model

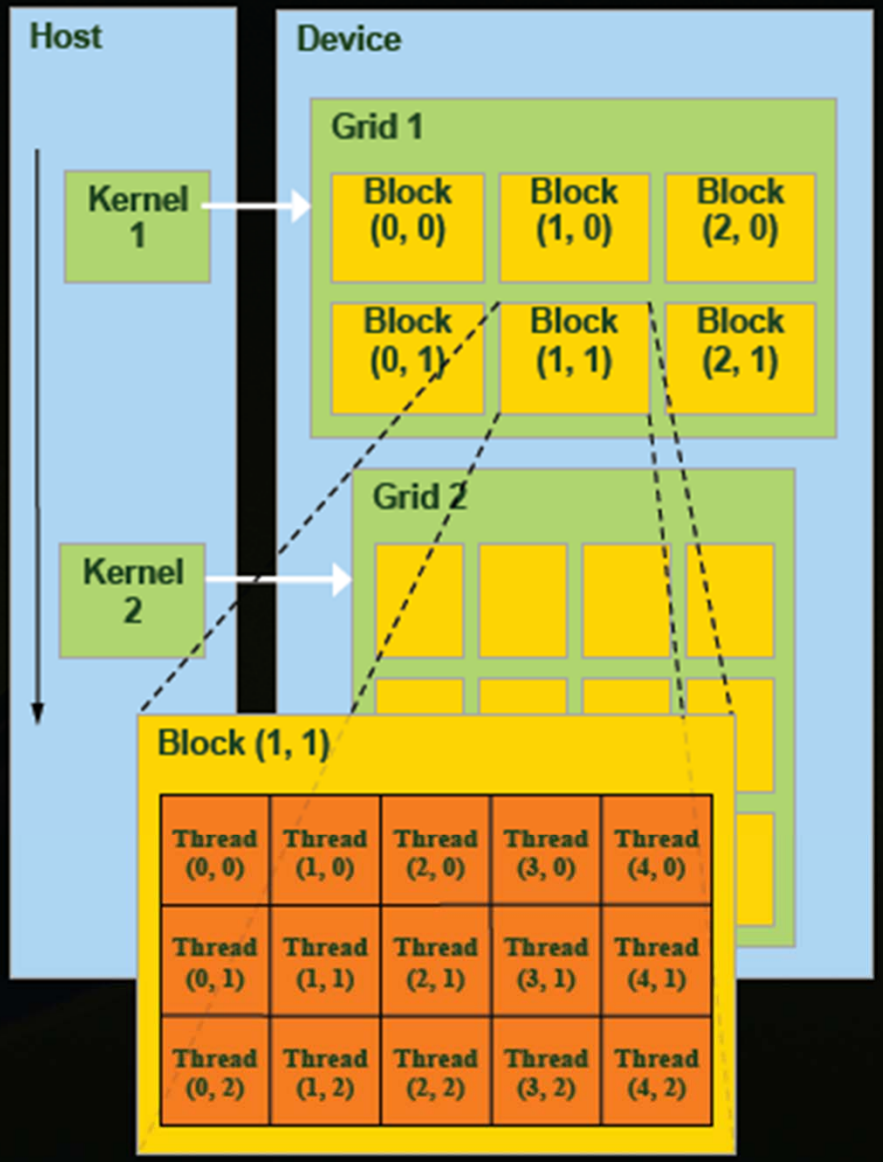


Multiple levels of parallelism

- Thread block
  - Up to 512 threads per block
  - Communicate through shared memory
  - Threads guaranteed to be resident
  - `threadIdx, blockIdx`
  - `__syncthreads()`
- Grid of thread blocks
  - `f<<<nblocks, nthreads>>>(a,b,c)`

Programming Model (SPMD + SIMD): Thread Batching

- A kernel is executed as a grid of thread blocks
- A thread block is a batch of threads that can cooperate with each other by:
  - Efficiently sharing data through shared memory
  - Synchronizing their execution
    - For hazard-free shared memory accesses
- Two threads from two different blocks cannot cooperate



3D Bilateral Filtering on the GPU

- **Algorithm design choices**
  - How do threads access memory?
  - Choices about use of high-speed local caches.
    - Global memory (shared), constant memory, shared memory, texture memory, etc.
- **Tunable algorithm parameters**
  - Thread block size, number of threads per block.

Other Speed Bumps Influencing Design

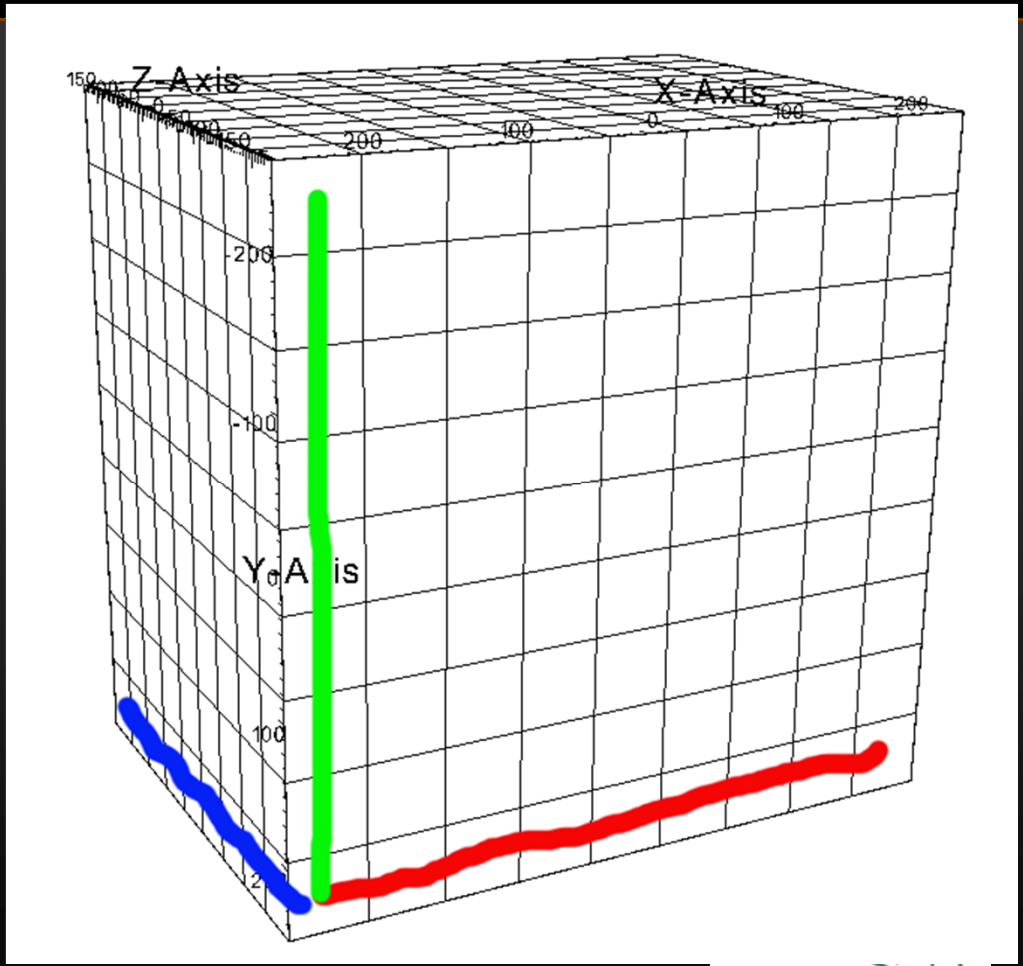
- **Limit on number of thread bocks.**
  - 1D and 2D grids of thread blocks.
  - No 3D grid of thread blocks.
  - Max dim size = 64K.
- **Limit on number of threads per thread block.**
  - Max of 512 threads per block.
  - Max dims (512,512,64) threads/block.

Design Constraints

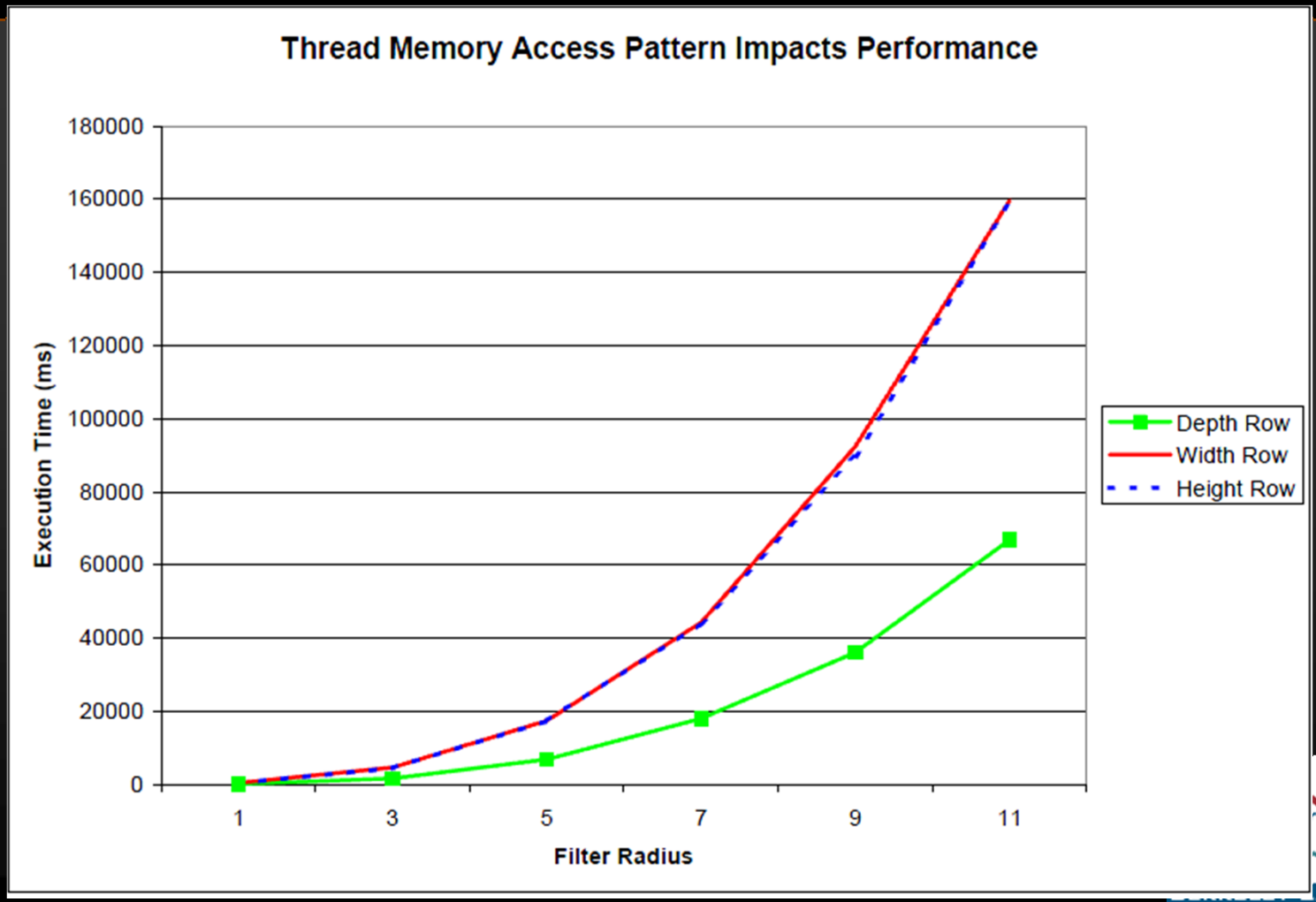
- **No 3D grid of thread blocks:**
  - Our thread kernel must process a row of voxels in width, height or depth.
    - Which works best?
  - Thread block array is 2D of some number of threads.
    - Which size/shape works best?

Memory Access Patterns

- Depth-row (blue)
- Height-row (green)
- Width-row (red)
- Question: which access pattern results in best performance?



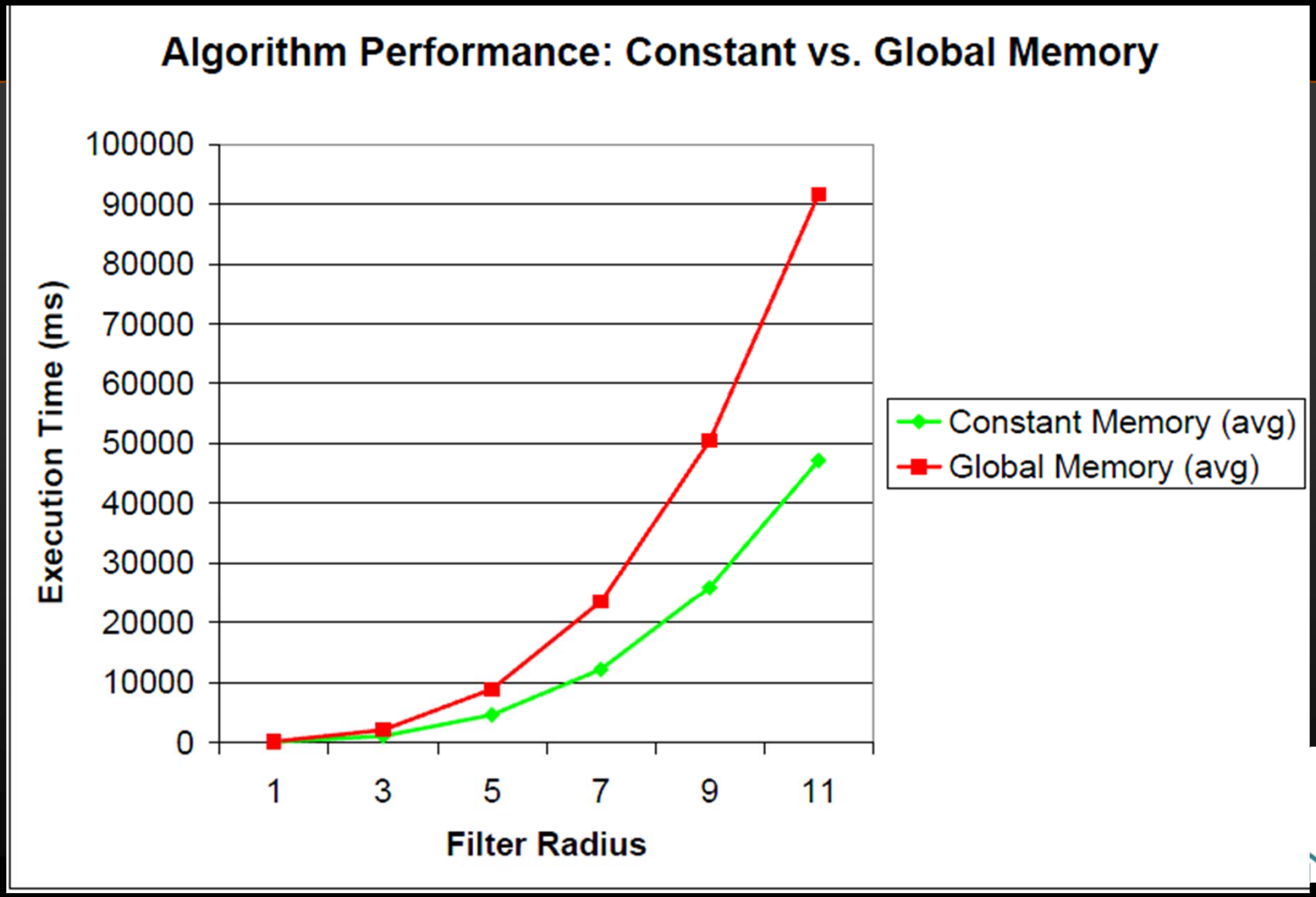
Memory Access Pattern Test Results



Device Memories

- **Global – large, high latency, low bandwidth**
- **Constant – small, low-latency, high bandwidth.**
  - 64KB not large enough for src, dst volumes
  - 64KB large enough for 1D&3D filter weights up to r=12.
- **Shared memory – small, 16KB, split into banks across multiprocessors (too small for this project).**
- **Question: how is performance affected if we use global vs. constant memory for the filter weights?**

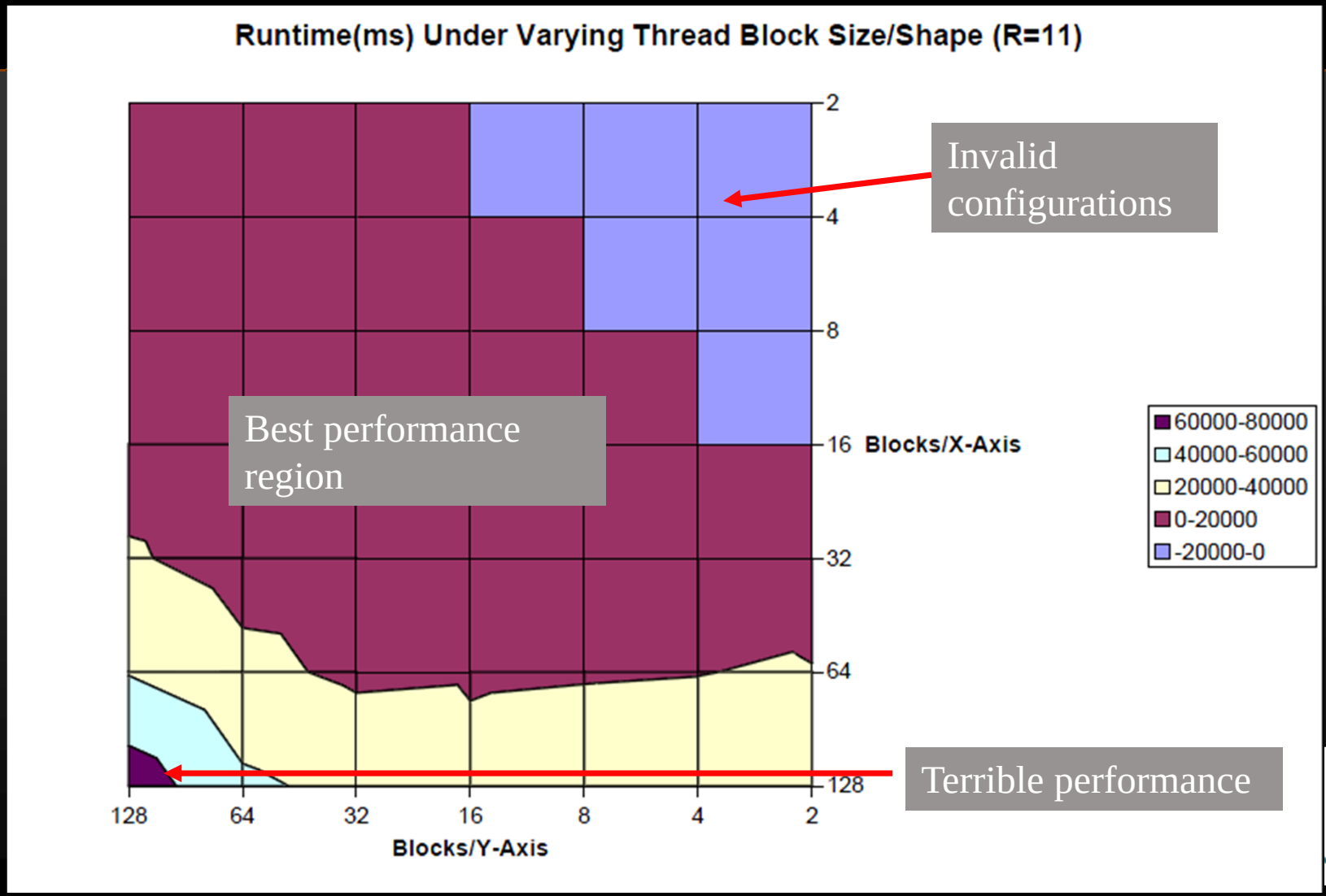
Device Memories Test Results



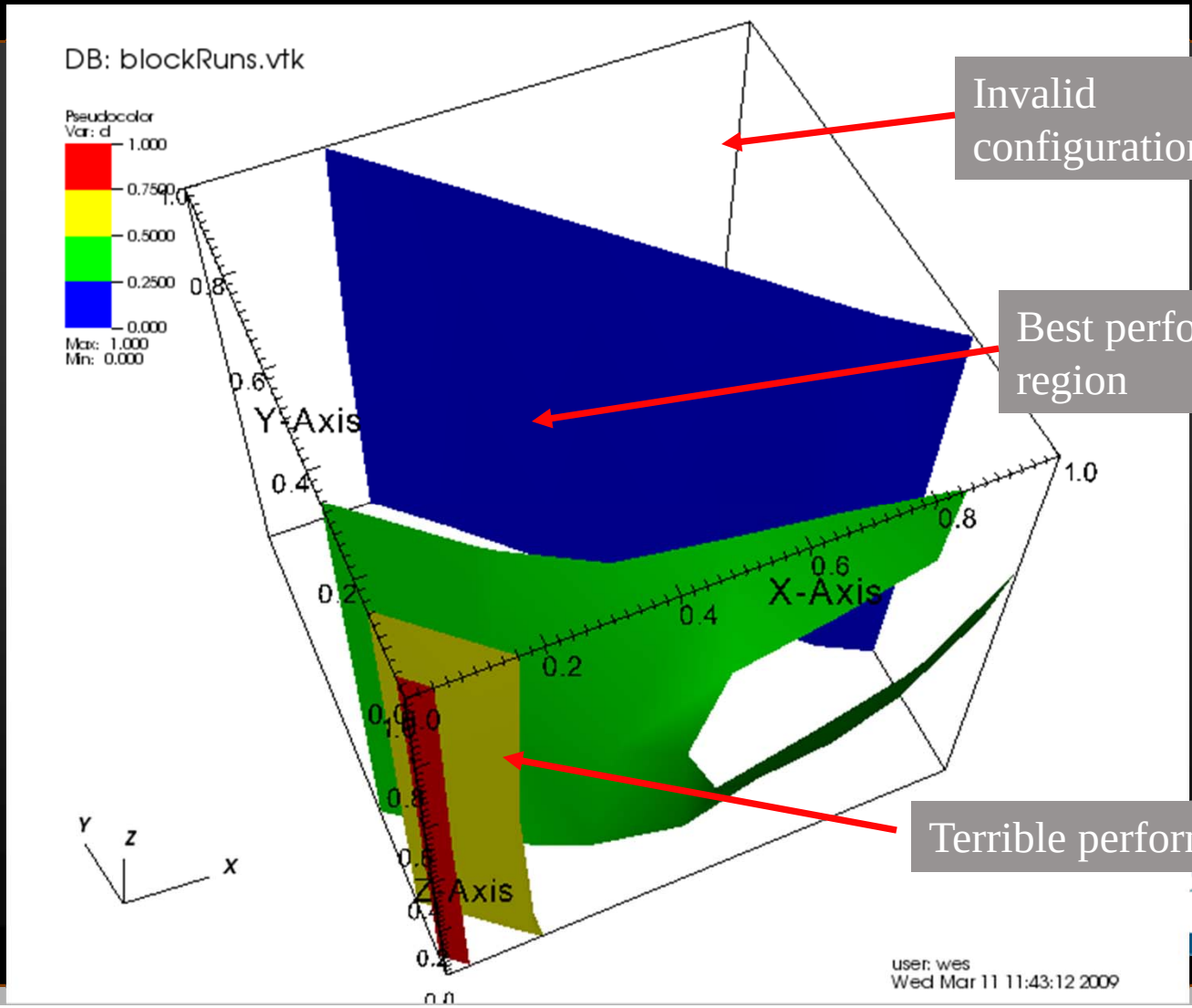
Tunable Parameters: Thread Block Size and Shape

- **Basic ideas:**
  - More vs. fewer thread blocks.
    - Fewer thread blocks means more threads per block.
  - Shape of thread blocks.
    - Square-shaped vs. oblong.
- **Question: which combination of thread block size and shape results in best performance?**
  - Note: this is the domain of autotuning.

Thread Size/Shape Test Results (1/3)

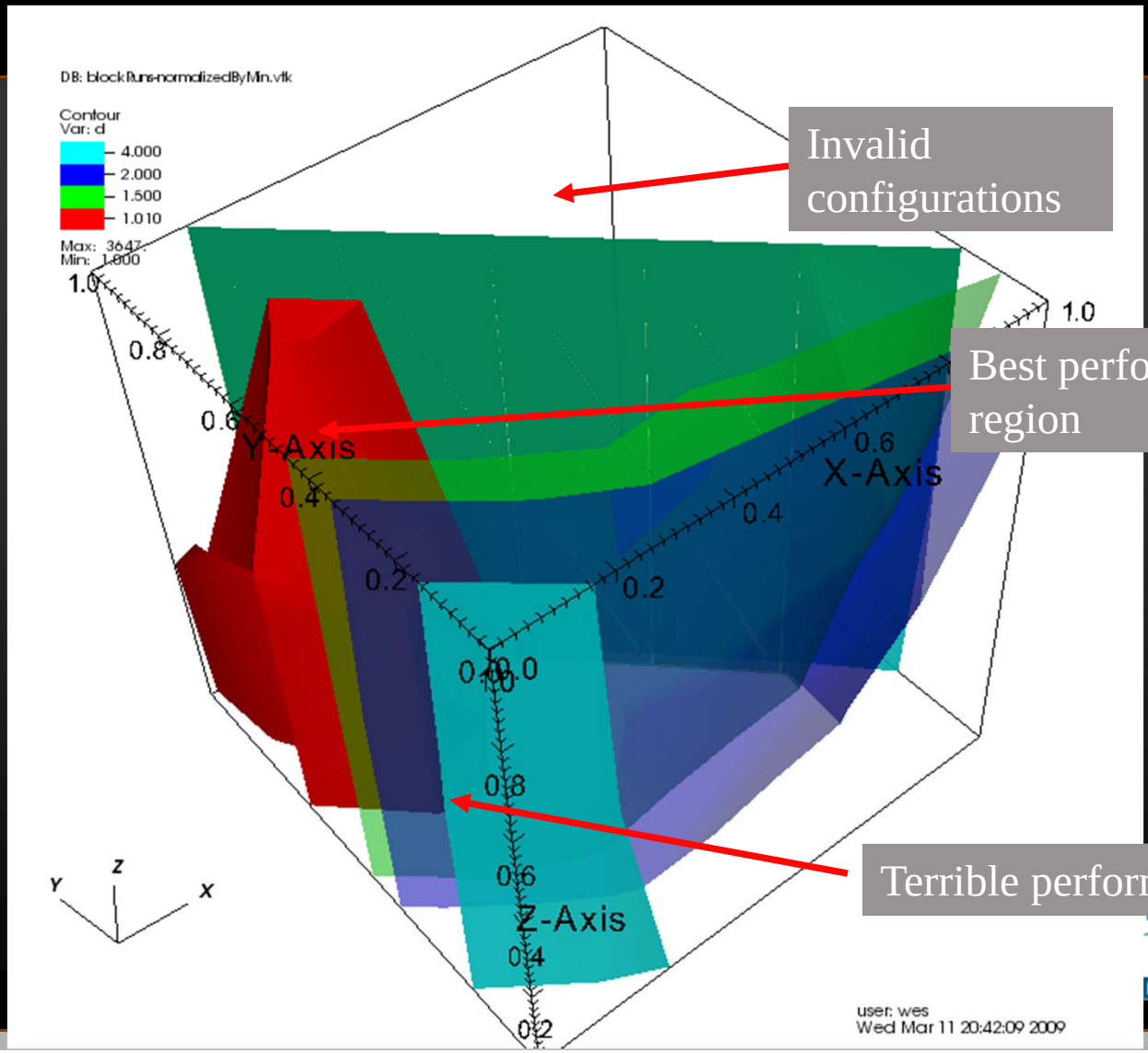


Thread Size/Shape Test Results (2/3)



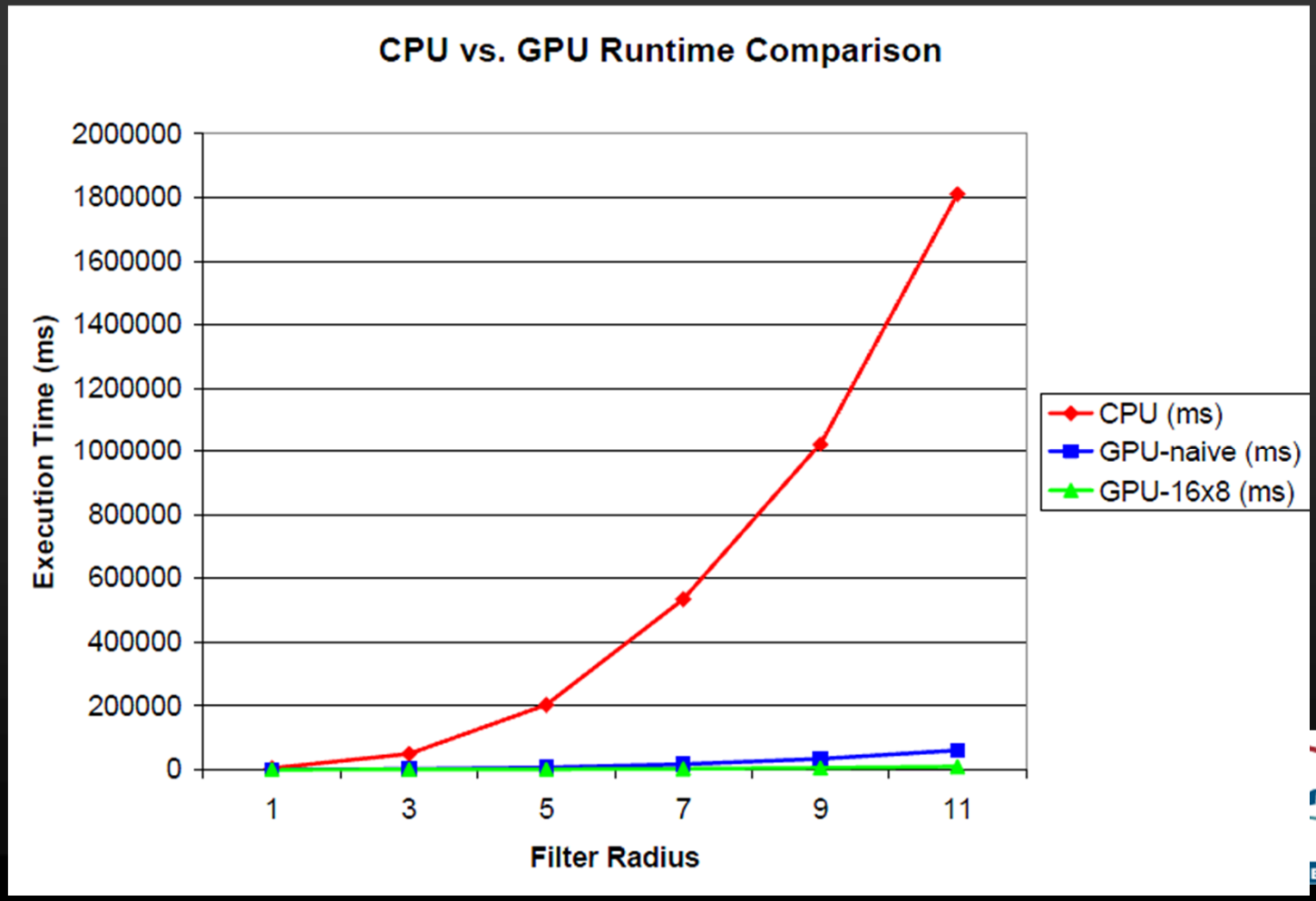
BERKELEY LAB

Thread Size/Shape Test Results (3/3)

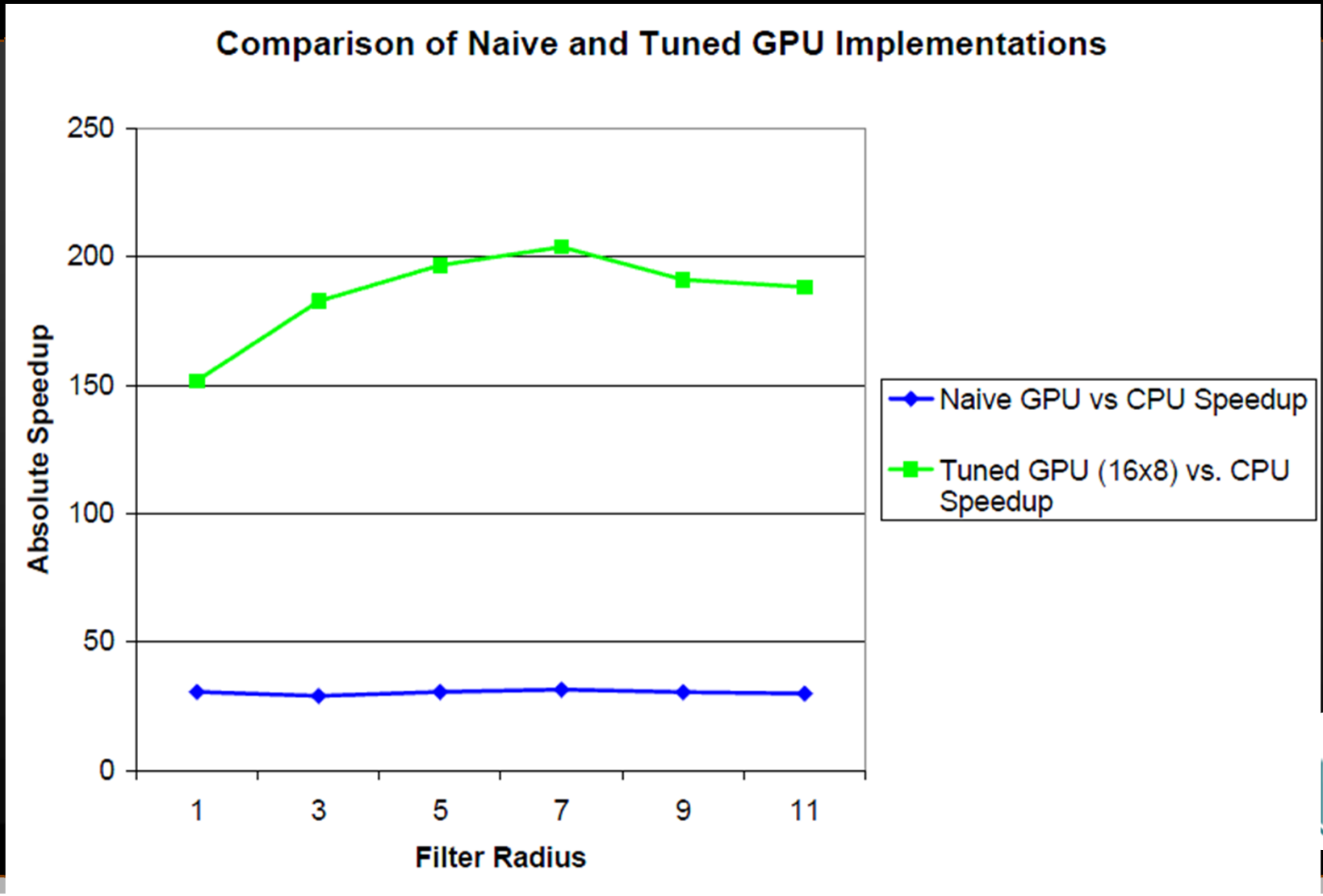


BERKELEY LAB

CPU vs. GPU Performance Comparison (1/2)



CPU vs. GPU Performance Comparison (2/2)



Conclusions/Discussion

- **GPU configurations with best performance:**
  - Threads access voxels along depth: *coalesced memory access*!
  - Use Constant memory rather than global memory to hold filter weights
  - Thread block size/shape: 16x8
- **GPU version outperforms CPU implementation**
  - 30x for naïve implementation.
  - 150x-200x for tuned implementation.
  - Why? Memory bandwidth (142GB/s vs. ~10GB/s) and keeping the memory pipeline full.