GPU TECHNOLOGY CONFERENCE

GPU-accelerated data expansion for the Marching Cubes algorithm

San Jose (CA) | September 23rd, 2010

Christopher Dyken, SINTEF Norway
Gernot Ziegler, NVIDIA UK

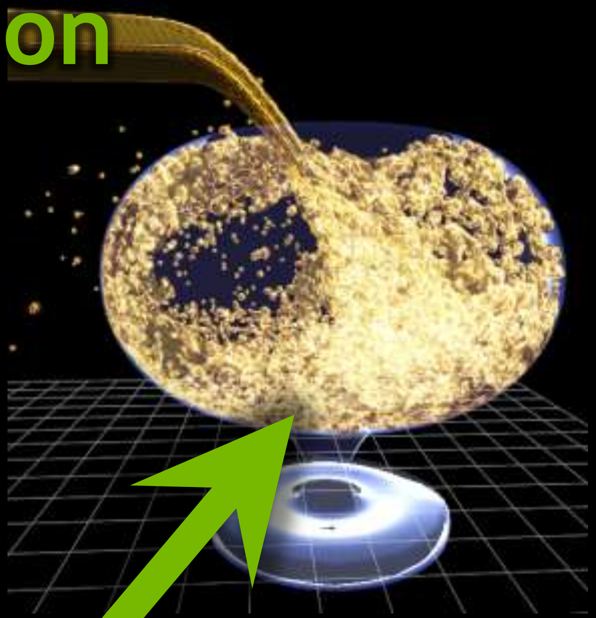
PRESENTED BY NVIDIA.

Agenda

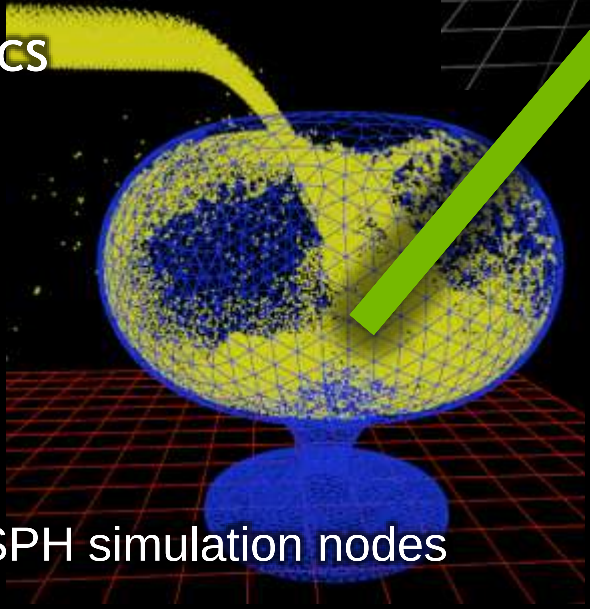
- Motivation & Background
- Data Compaction and Expansion
  - Histogram Pyramid algorithm and its variations
  - Optimizations and benchmark results
- Marching Cubes based on Histogram Pyramids
  - Mapping and performance considerations
  - Benchmark results
- Visualization of SPH simulation results
  - Videos

Motivation: Fast SPH visualization

- Smoothed-particle Hydrodynamics (SPH)
  - Meshless Lagrangian method:
    - Nodes (particles) are not connected
    - Node position varies with time
  - Models fluid and solid mechanics
  - Nodes form a density field
- High-quality visualization:
  1. Approximate density field
  2. Marching Cubes
  3. Render iso-surface



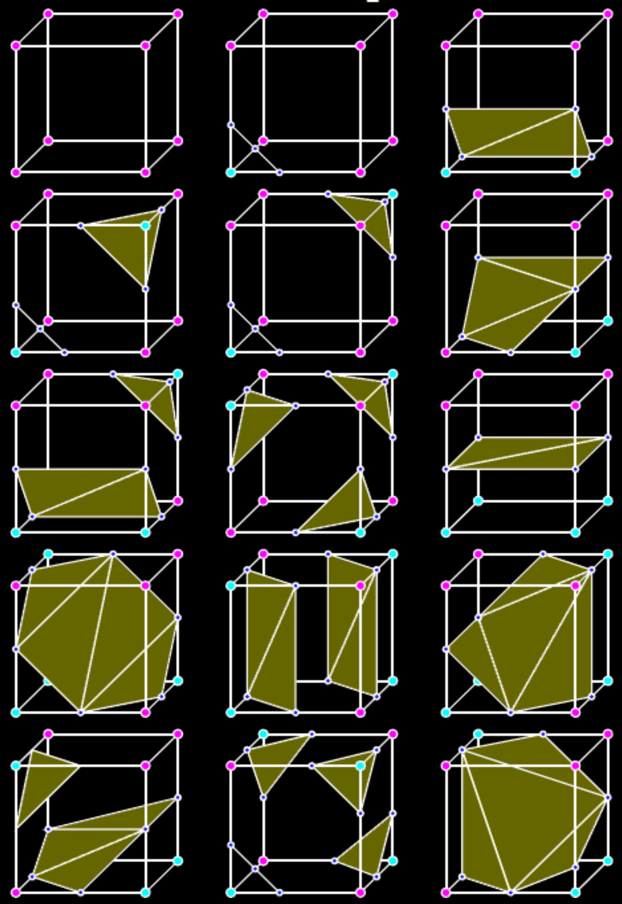
Surface Visualization



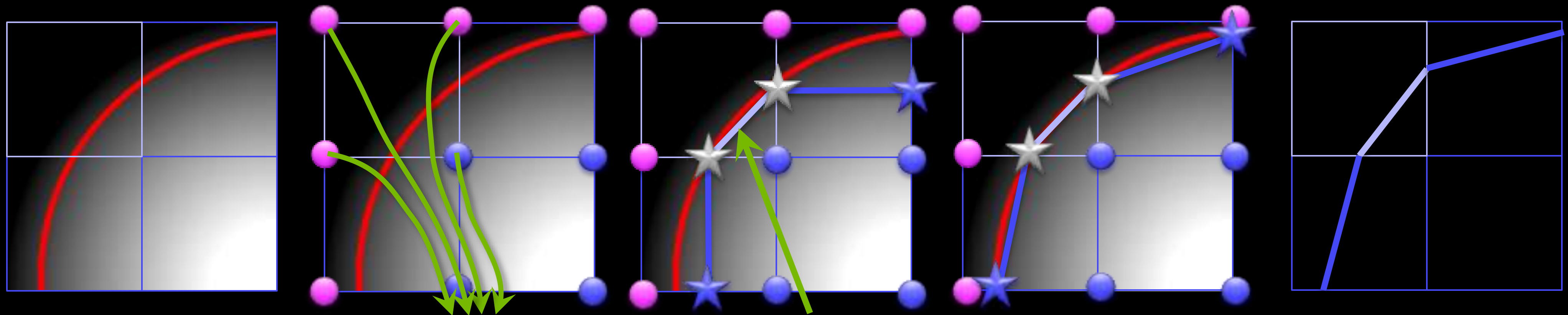
SPH simulation nodes

Extract iso-surface via Marching Cubes

- Scalar field is sampled over 3D grid
- Marching Cubes [Lorensen87]
  - Marches through a regular 3D grid of cells
    1. Each MC cell spans 8 samples
    2. Label corners as inside or outside iso-value
    3. Eight in/out labels give 256 possible cases
    4. Each case has a tessellation template
       - Devised such that tessellations of adjacent cells match
       - Vertices lie on lattice edges
         - positioned using linear interpolation
  - De-facto standard algorithm for this problem



Example: Marching Cubes in 2D



Input: A scalar field (gray=scalar field) (red=iso-surface)

Upper left MC cell, case = %0001 = 1 (pink=outside,blue=inside)

Upper left MC cell, produce template tessellation 1

Upper left MC cell, calculate vertex positions

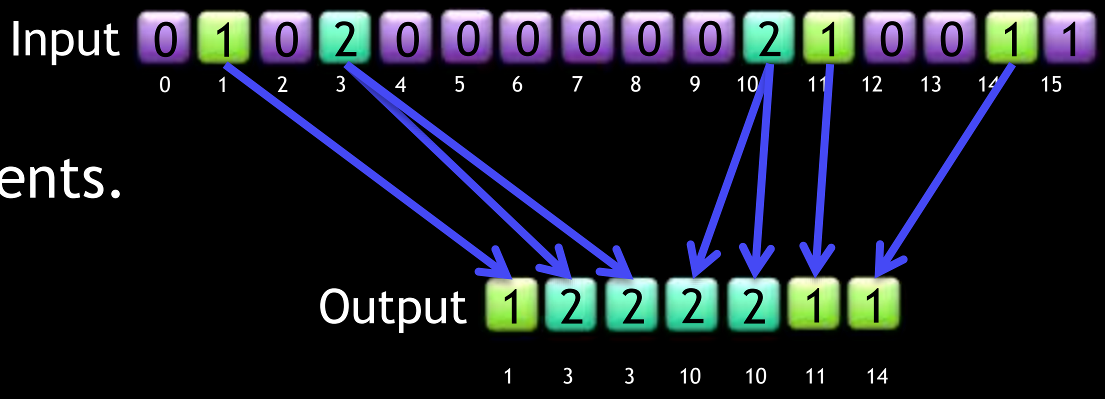
Upper left MC cell, Output: A line segment

1. For each cell:
   Determine MC case and # vertices of template — ✓ Data-parallel!
2. Determine total # vertices and
   output index of each MC cell's vertices — **Not trivially data-parallel!**
3. During vertex output: calculate actual positions — ✓ Data-parallel!

Step 2 is Data Compaction & Expansion

- We want to answer:
  - How many triangles to draw?
  - What is the mapping between input and output?
    - Classic: At which output position j shall MC cell i write vertex k?
    - Put differently: Which MC cell i and vertex k does output position j belong to?
- Data compaction & expansion provide answers:
  - Data compaction:
    - Extract all cells that produce geometry
  - Data expansion:
    - Each cell that produces geometry issues 3-15 vertices

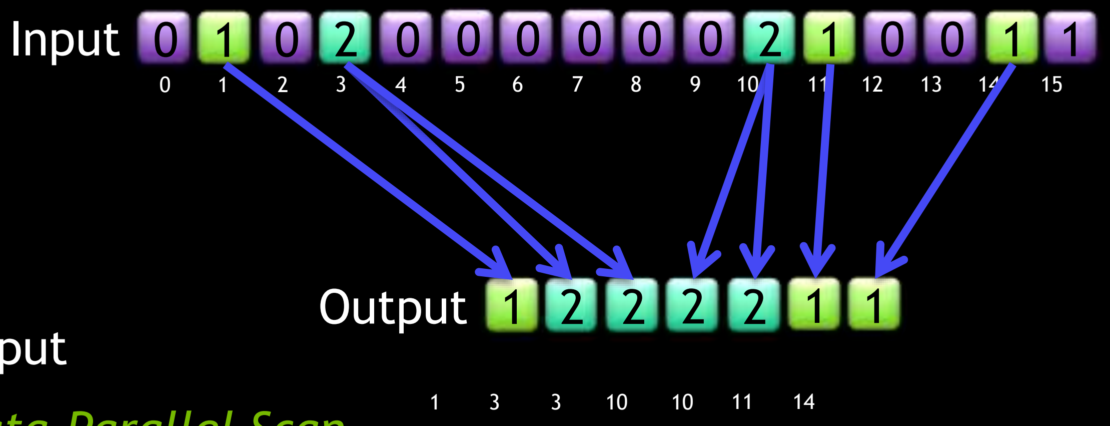
Data Compaction and Expansion



- Problem definition
  - We start with n input elements.
  - Input element j produces a_j output elements.
  - Discard all elements where $a_j = 0$.
- An important algorithmic pattern!
  - Trivial implementation in serial implementation (e.g. CPU).
  - Non-trivial on data-parallel architectures (e.g. GPU)!

Input or Output-centric solutions

- Input-centric solution:
  - For every input element
    - Compute output offsets
    - *Scatter* relevant input to output
    - Typical serial solution and *Data-Parallel Scan*
- Output-centric solution:
  - For every output element
    - *Determine* input element from output index
    - Histogram Pyramid (*HistoPyramid*): Reduction-based search structure



HistoPyramid: Stages of Algorithm

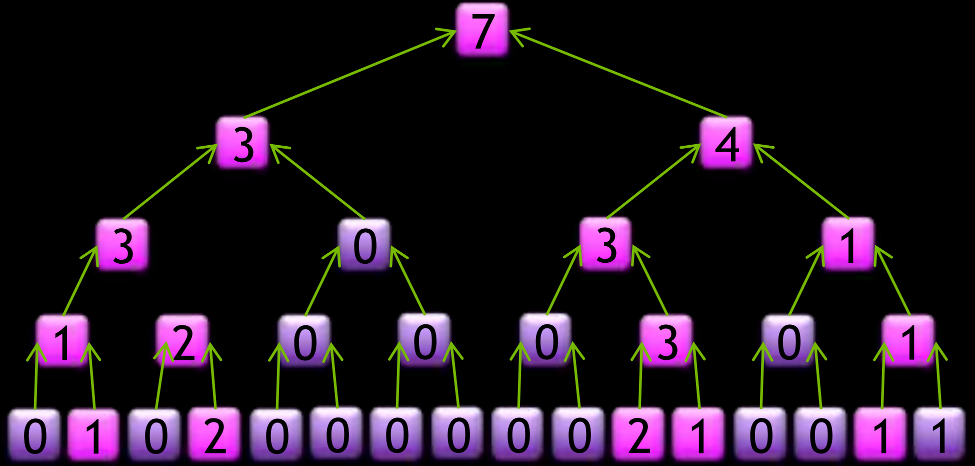
- *Input is Baselevel*
  - For each input element, init with number of output elements

Input element index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base level:	0	1	0	2	0	0	0	0	0	0	2	1	0	0	1	1

- *Level Buildup*
  - Build further levels through reduction
- *HistoPyramid Traversal*
  - For each output index:
    Find corresponding input index (via HistoPyramid traversal)

HistoPyramid Buildup

- Build further levels from baselevel
  - Add two elements (reduction)
    - Number of elements halves each iteration
    - $\log_2 n$ iterations
      - Each iteration half the size of the previous iteration
  - Data-Parallel algorithm



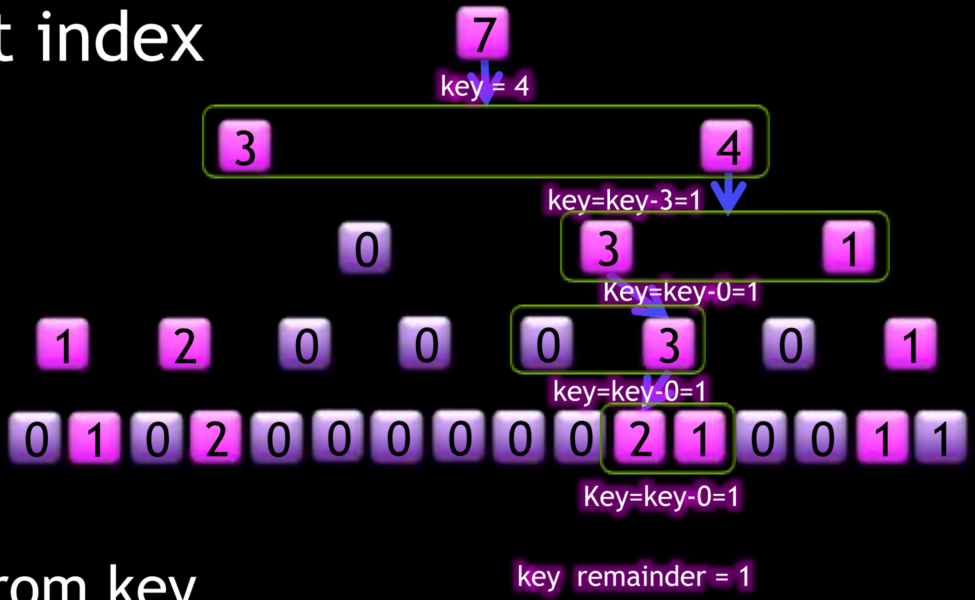
- Top element equals number of output elements (Step 2A)
- Data of all reduction levels: 2:1 HistoPyramid

Output Allocation

- Output size is known from top element of HP
- Allocate output
- Start one thread per *output element*
- Each thread knows its output index
- Now use HistoPyramid as
  search structure for finding corresponding input element

HistoPyramid Traversal

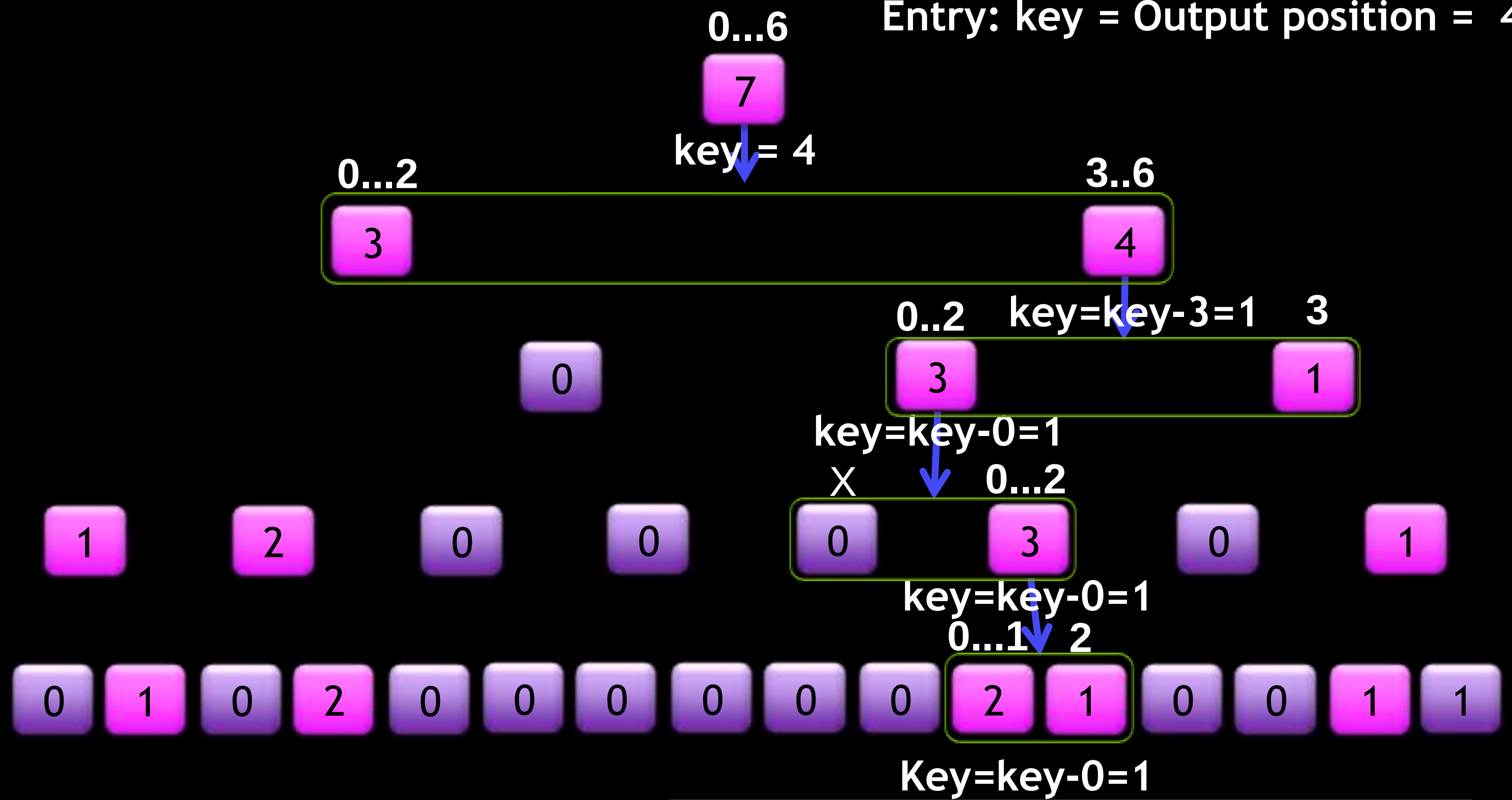
- Each thread handles one output element
- *key* : variable, initially output index
- Binary Search through HP, from top-level to base-level
  - Reduction inputs x and y form *key ranges [0, x) and [x, x+y)*
  - Choose fitting range for key
  - Subtract chosen range's start from key



- Note: For $a_j > 1$, several output threads will end up at same input element: key remainder is *index within this set*

HistoPyramid Traversal

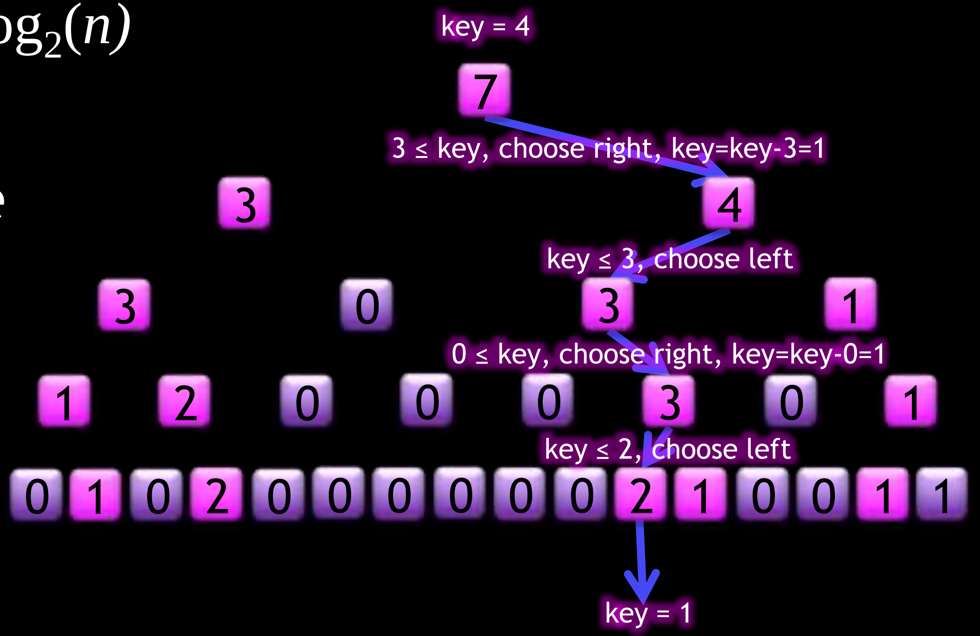
Entry: key = Output position = 4



Input pos=10, key remainder = 1

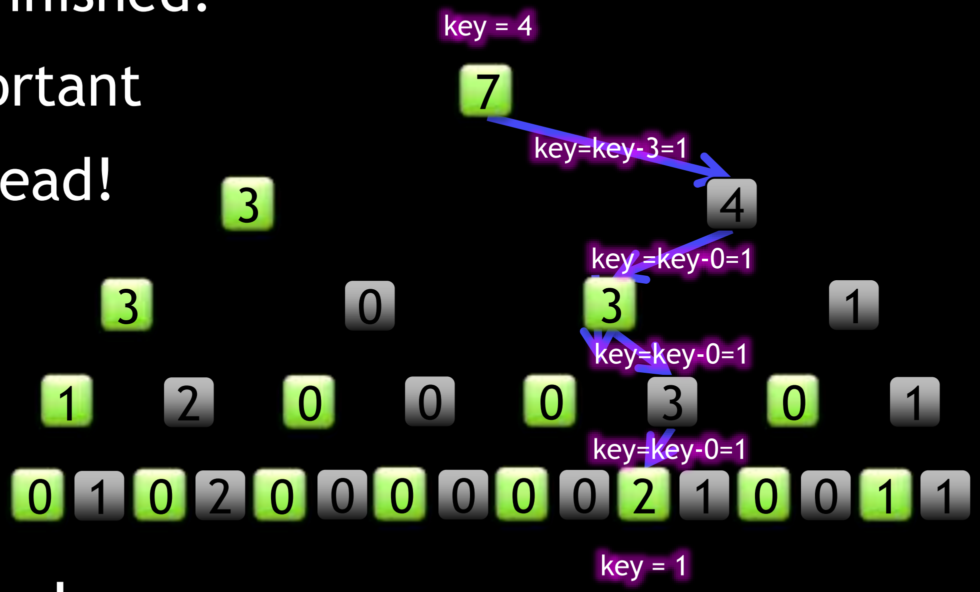
More observations on HP traversal

- Fully data-parallel algorithm (HP is read-only in traversal)
- Traversal steps/Data dependency: $\log_2(n)$
  - Note: A pyramid has less latency
- Traversal path follows roughly a line
  - Adjacent output elements have very similar traversal paths
    - Good cache coherence
  - Large chunks of output elements have identical paths from top
    - Good for many-thread broadcast
- Some elements are never visited



Optimization 1: Discard some partial sums

- Observation:
  - In traversal, after build-up has finished:
    - Only the left nodes are important
    - The right nodes needn't be read!
- We can discard all the right nodes
  - Note: Number of all left nodes equals number of input elements
  - Similarities to the Haar-transform!



Optimization 2: k-to-1 reductions

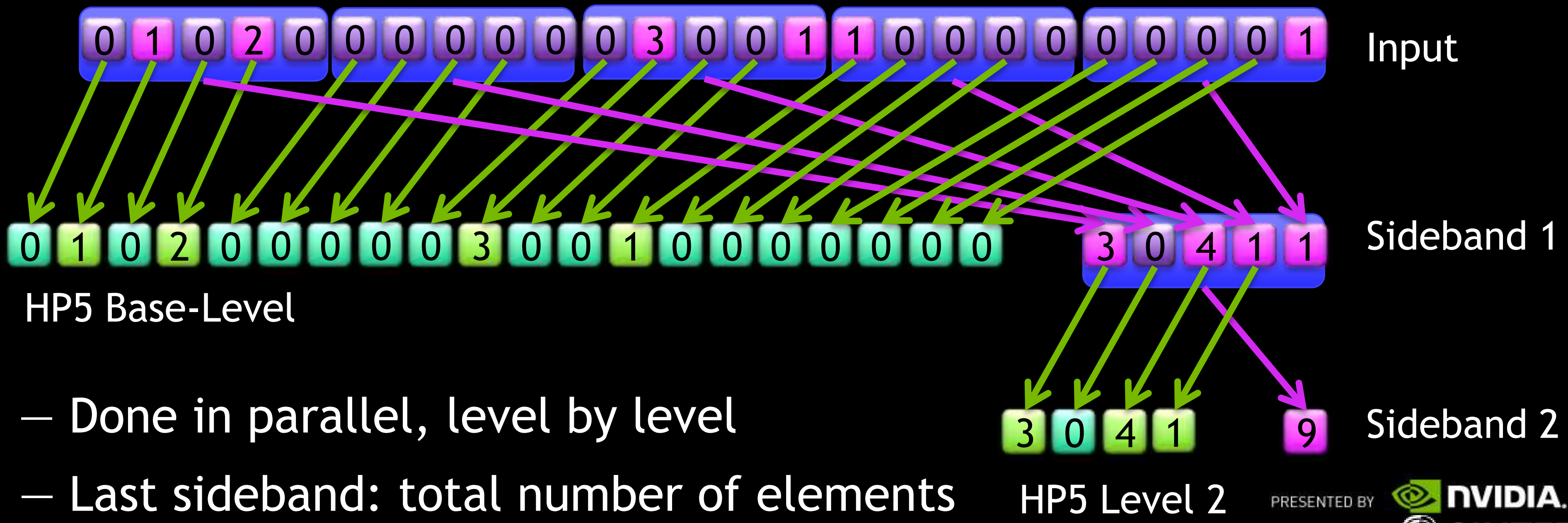
- Reduction does not have to be 2-to-1
- Example: 4-to-1 reduction is also possible
  - Fewer levels of reductions -> fewer levels of traversal : log4(n)
  - Better for hardware (can fetch up to 4 values at once, reduce overall latency with fewer traversal steps)
- HPMC from 2007 uses 4-to-1 reductions in 2D (texture mipmap-like)
  - Output extraction for consecutive elements follows space-filling curve in base level
  - Traversal: Adjacent HP levels accessed in mipmap-like fashion
  - **Excellent texture cache behaviour**

HP5 (5-to-1 HistoPyramid)

- Combines two previous optimizations:
  - Buildup: Every reduction adds five elements into one output, BUT:
    - Only four of the reduction elements are stored!
    - Fifth reduction element goes to computational sideband
      - only acts as temporary data during reduction
- Traversal requires only first four elements
  - Fifth element is directly deducted during top-down path.
- Advantage of HP5:
  - Less data storage
  - more efficient traversal

The HP5 reduction

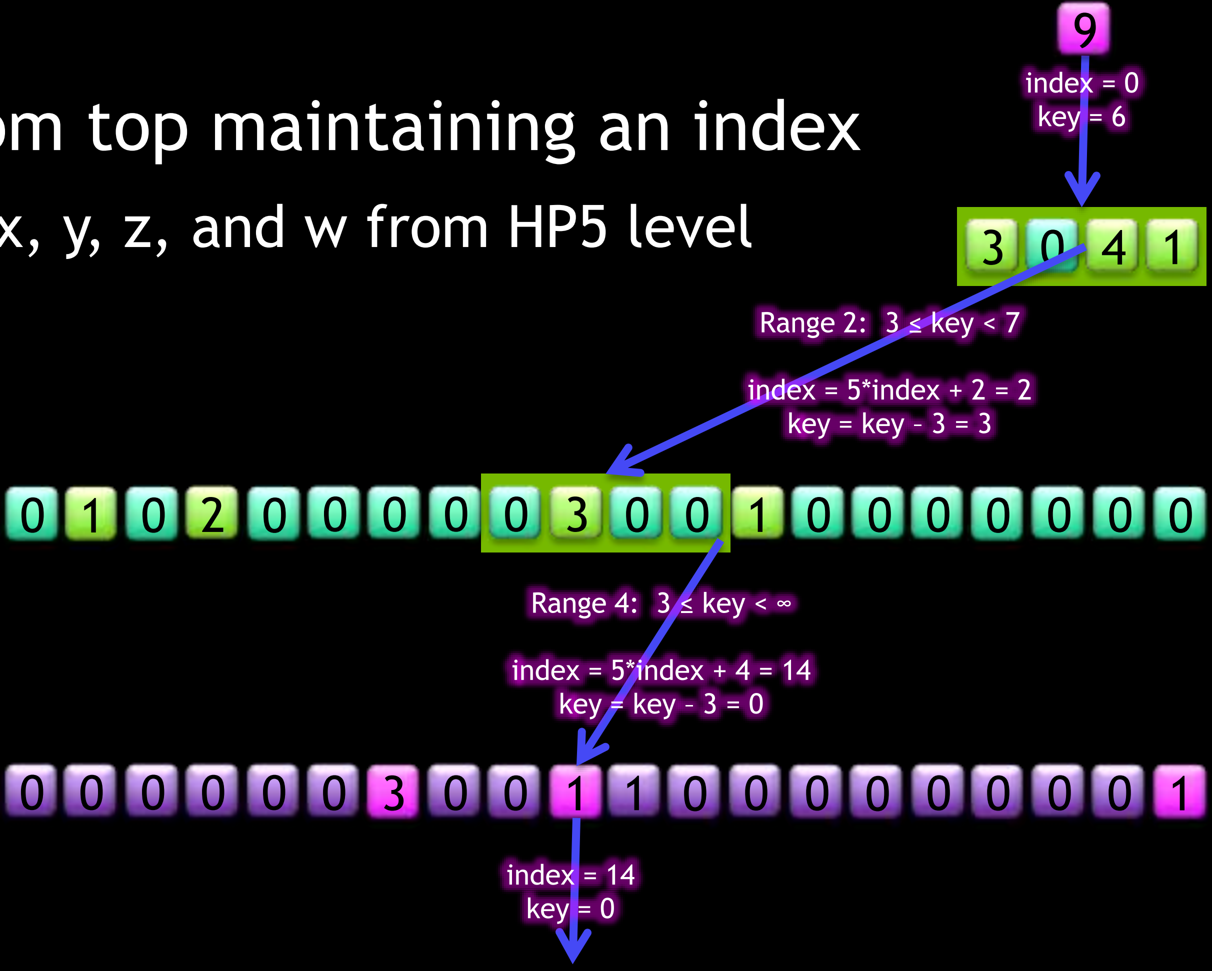
- For each group of 5 elements in input stream or sideband:
  - First 4 elements into HP5 level
  - The sum of the 5 elements into sideband



  - Done in parallel, level by level
  - Last sideband: total number of elements

The HP5 traversal

- Given a key, traverse from top maintaining an index
  - Fetch 4 adjacent values x, y, z, and w from HP5 level
  - Build key ranges
    - [0,x)
    - [x,x+y)
    - [x+y,x+y+z)
    - [x+y+z,x+y+z+w)
    - [x+y+z+w, ∞)
  - Check range, adjust key and index.



HistoPyramid performance

- Data compaction: CUDA 3.2 SDK, Tesla C2050

2 million input elements, whereof N% retained	Scan	Atomic Ops	HP 4-to-1	HP 5-to-1
1% retained	0.70 ms	0.37 ms	0.34 ms	0.28 ms (2.5x)
10% retained	0.80 ms	3.04 ms	0.47 ms	0.38 ms (2.1x)
25% retained	0.81 ms	7.47 ms	0.63 ms	0.53 ms (1.53x)
50% retained	0.83 ms	14.89 ms	0.93 ms	0.81 ms (1.02x)
90% retained	0.85 ms	26.75 ms	1.40 ms	1.25 ms (0.60x)

HistoPyramid performance

- Data compaction: CUDA 3.2 SDK, Tesla C2050

2 million input elements, whereof N% retained	Scan	Atomic Ops	HP 4-to-1	HP 5-to-1
1% retained	0.70 ms	0.37 ms	0.34 ms	0.28 ms (2.5x)
10% retained	0.80 ms	3.04 ms	0.47 ms	0.38 ms (2.1x)
25% retained	0.81 ms	7.47 ms	0.63 ms	0.53 ms (1.53x)
50% retained	0.83 ms	14.89 ms	0.93 ms	0.81 ms (1.02x)
90% retained	0.85 ms	26.75 ms	1.40 ms	1.25 ms (0.60x)

Explanation: HistoPyramids vs. Scan

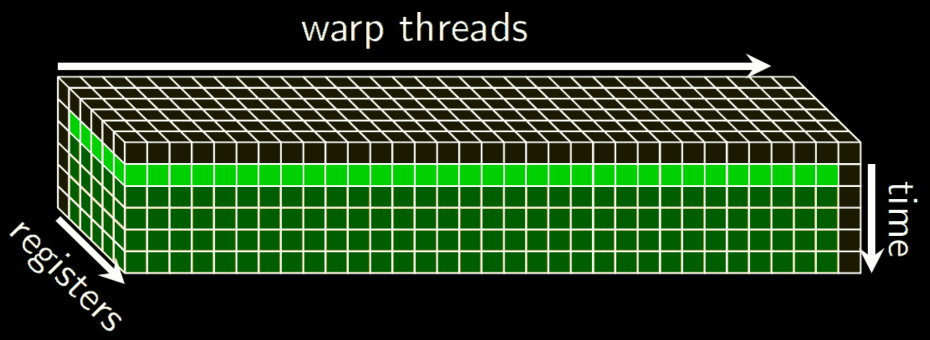
- Scan is input-centric
  - Efficiently computes output offset for all input elements
  - Uses one thread per input elements to write output (scatter)
  - For few relevant input elements:
    - Redundantly computes output offsets for all input elements
    - Starts superfluous threads for all, and many irrelevant, input elements
- HistoPyramids is output-centric
  - Minimal amount of computations per input element
  - Uses one thread per output element to write output (gather)
    - But: requires HP traversal instead of a simple array look-up.

HistoPyramid-based Marching Cubes

- Recall the 3-step subdivision of marching cubes:
  1. For each cell, determine case and find required # vertices
     - Embarrassingly parallel
     - Performed in CUDA
  2. Find total number of vertices and output-input index mapping
     - Build 5-to-1 HistoPyramid
     - Performed in CUDA
  3. For each vertex, calculate positions
     - Embarrassingly parallel
     - Performed directly in an OpenGL vertex shader

Step 1: Cell MC Case and Vertex Count

- Adjacent MC cells share corners
  - Let a CUDA warp sweep through a 32x5x5 chunk of MC cells
    - Process XZ-slices slice by slice:
      - Check in/out state of 6 corners along Z, (1 state per cell)
      - exchange for cells processed by this thread (2 states per cell)
      - Pull results from previous slice, (4 states per cell)
      - Exchange results across warps (X-axis), (8 states per cell)
      - Use a 256-byte table to find number of vertices required for cell



- Recycles scalar field fetches and in-out classifications
  - 32x5x5 MC cases in 33x6x6 fetches = 1.5 fetches per cell

Step 2: HistoPyramid 5-way Reduction

- HistoPyramid built level by level, from bottom to top
  - Reduction kernel uses 160 threads (5 warps)
  - All five warps fetch input sideband element as uint's into shmem
    - Adjacent shared memory writes, no bank conflicts
  - Synchronize
  - One single warp sums and stores results in global mem
    - Each thread reads 5 adjacent elements from shared mem
      - Fetches with stride = 5, no bank conflicts
    - Output 4 elements to HistoPyramid Level (as uint4's)
    - Store sum of the 5 elements in HistoPyramid sideband (as single uint's)

Optimizing the HistoPyramid Reduction

- Reduce global mem traffic:
  - Sidebands are streamed through global mem between reductions
    - Combine two reductions into one kernel
      - Requires 800+160 uint's of shmem (3.8 K), free of bank conflicts
    - Combine three reductions into one kernel
      - Requires 800+800 uint's in shmem (6.3 K), free of bank conflicts
    - Combine step 1 and three reductions into one kernel
      - Each warp processes 32x5x5 = 800 MC cells, 4000 per block
      - Shares shared mem with reduction, no extra shared mem required
- Reduce kernel invocation overhead
  - Build the apex of the HistoPyramid using a single kernel
    - Reduces the number of kernel invocations

Step 3: Extract output vertices

- Performed directly on the fly in OpenGL vertex shader:
  - No input attributes
  - `gl_VertexID` is used as key for HistoPyramid traversal
    - Terminates in corresponding MC cell
    - MC case gives template tessellation
    - Key remainder specifies lattice edge for vertex in template tessellation
  - Vertex position found by sampling scalar field at edge end points
- Uses OpenGL 4's indirect draw
  - Number of vertices to render fetched from buffer object
  - No CPU-GPU synchronization needed

Results: MC Implementation Approaches

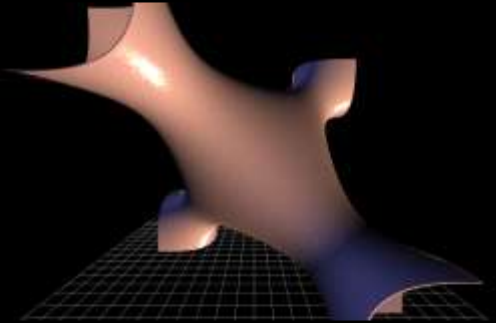
- NVIDIA Compute SDK's MC sample uses CUDPP
- HPMC library [http://www.sintef.no/hpmc]:
  HistoPyramids (4:1) in OpenGL GPGPU approach
- Our new development of HPMC uses CUDA HistoPyramid (5:1)

- Key characteristics:
  - Most often: 0 triangles per cell
  - Maximally: 5 triangles per cell (=15 vertices)
  - On average: 0.05 - 0.15 triangles per cell
    - Input (#cells) grows with cube of lattice grid resolution
    - Output (#triangles) grows with square of lattice grid resolution

256^3 8bit performance (Tesla C2050)

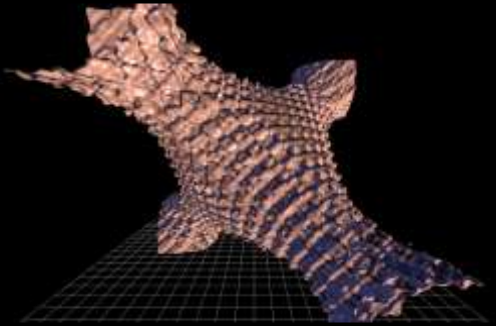
Smooth Cayley (iso=0.5)

Triangles	445 522	(0.027 tris/cell)
NV SDK sample	72 fps	(1201 mvps)
OpenGL HP4MC	113 fps	(1868 mvps)
CUDA-OpenGL HP5MC	301 fps	(4985 mvps)
Speedup	**2.6x / 4.2x**	



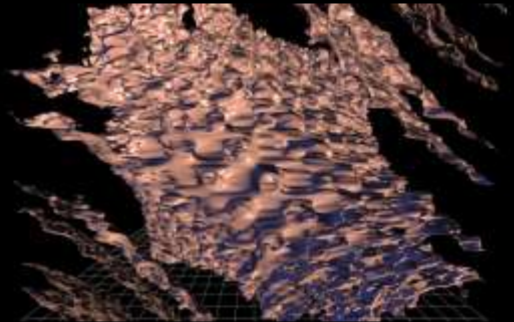
Bumpy Cayley (iso=0.5)

Triangles	643 374	(0.039 tris/cell)
NV SDK sample	66 fps	(1098 mvps)
OpenGL HP4MC	102 fps	(1689 mvps)
CUDA-OpenGL HP5MC	242 fps	(4006 mvps)
Speedup	**2.4x / 3.6x**	



Superbumpy and layered Cayley (iso=0.5)

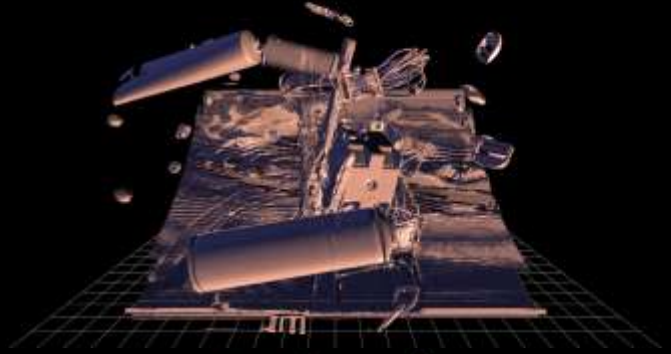
Triangles	3 036 608	(0.183 tris/cell)
NV SDK sample	34 fps	(571 mvps)
OpenGL HP4MC	47 fps	(774 mvps)
CUDA-OpenGL HP5MC	72 fps	(1199 mvps)
Speedup	**1.5x / 2.1x**	



512^3-ish 16-bit performance (Tesla C2050)

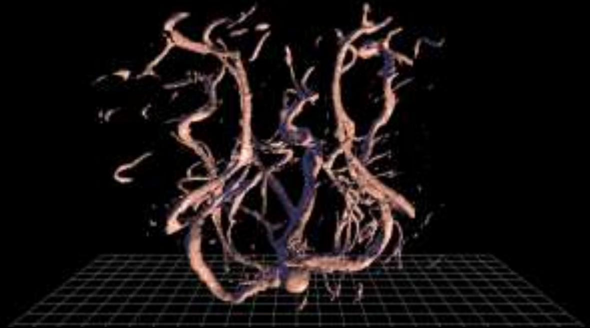
Backpack (iso=0.4) (www.volvis.org)

Size	512x512x373	(187 mb)
Triangles	3 745 320	(0.039 tris/cell)
OpenGL HP4MC	13 fps	(1291 mvps)
CUDA-OpenGL HP5MC	43 fps	(4129 mvps)
Speedup	**3.2x**	



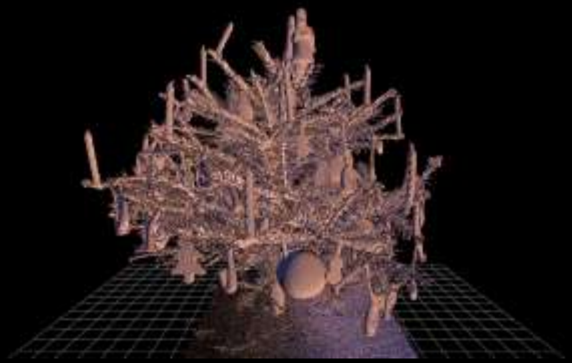
Head aneuyrism (iso=0.4) (www.volvis.org)

Size	512x512x512	(256 mb)
Triangles	583 610	(0.004 tris/cell)
OpenGL HP4MC	15 fps	(2034 mvps)
CUDA-OpenGL HP5MC	78 fps	(10399 mvps)
Speedup	**5.1x**	



Christmas tree (iso=0.05) (TU Wien)

Size	512x499x512	(250 mb)
Triangles	5 629 532	(0.043 tris/cell)
OpenGL HP4MC	10 fps	(1358 mvps)
CUDA-OpenGL HP5MC	28 fps	(3704 mvps)
Speedup	**2.7x**	



CUHP5 Marching Cubes Showcase Video

http://www.youtube.com/watch?v=WS95KjUS_Ww

Summary

- Our SPH visualization approach is based on Marching Cubes
  - Requires high performance data compaction and expansion
  - Output size is considerably smaller than input size
- 5:1 HistoPyramid buildup and traversal
  - Optimizations: 5:1 instead of 4:1, leave out last leaf, shmem
  - Performance comparison for typical input-output ratio of 1-10%
- Implementing Marching Cubes
  - Implementation details
  - Performance
- Fastest Marching Cubes in the world ?

CUHP5 Marching Cubes

Thank you!

Questions?

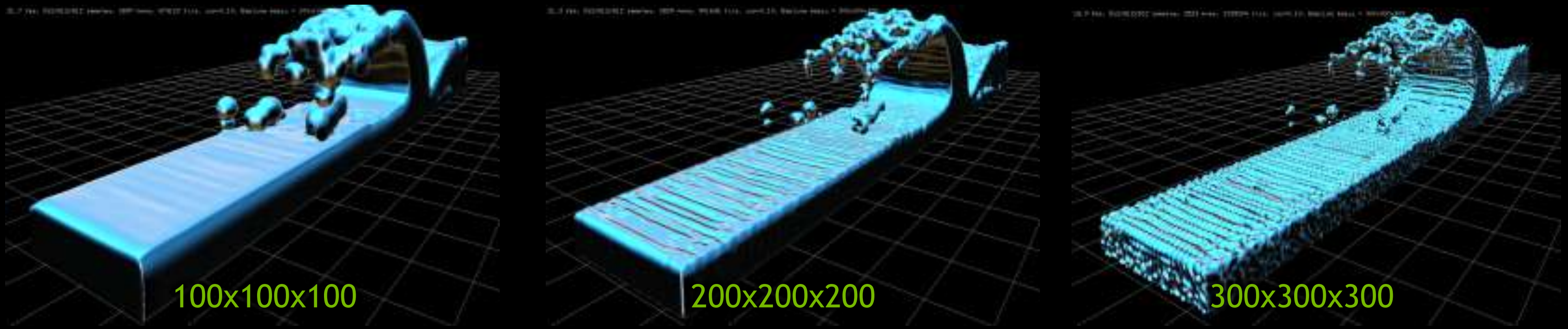
Chris Dyken <christopher.dyken@sintef.no>
Gernot Ziegler <gziegler@nvidia.com>

CUHP5 Marching Cubes

BONUS SLIDES

Build a scalar field from the SPH nodes

- We approximate using a quadratic tensor-product B-spline
  - Simple and runs well on a GPU
  - Spline space size controls blurring versus detail



  - A quasi-interpolant builds the spline
    - Contribution equals basis at position
      - Scatter contributions using atomic adds
      - No need to solve a linear system!