GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

HANDLING MASSIVE TRANSFORM UPDATES IN A SCENEGRAPH

Markus Tavenrath, March 5th 2016

Senior Developer Technology Engineer

PRESENTED BY

NVIDIA.

MOTIVATION

Nvpro-pipeline good for static scenes

What about dynamic content?

Update cost dominates rendering time

Big potential: Transform updates

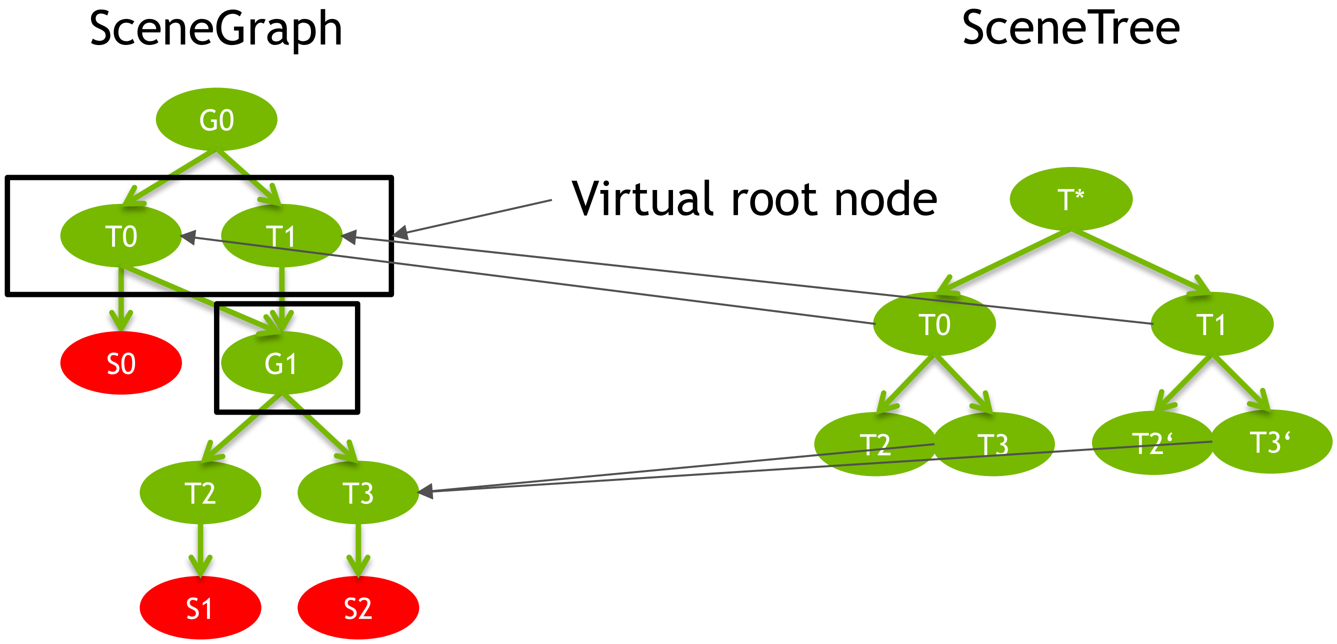
TransformTree is now a separate module

Adds CUDA support

TRANSFORM TREE

What is a TransformTree

TransformTree is unfolded SceneGraph with nothing more than the transforms



TRANSFORM TREE

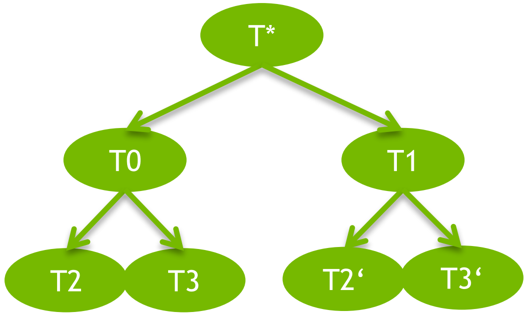
Usage?

What can a TransformTree be used for?

- Incremental computation of world matrices
  - i.e. T2.world = T2.local * T0.local * T*.local

What is the world transform be used for?

- Rendering
  - Shader
  - Culling
  - Bounding box computation
- Collision detection
- …



TRANSFORM TREE

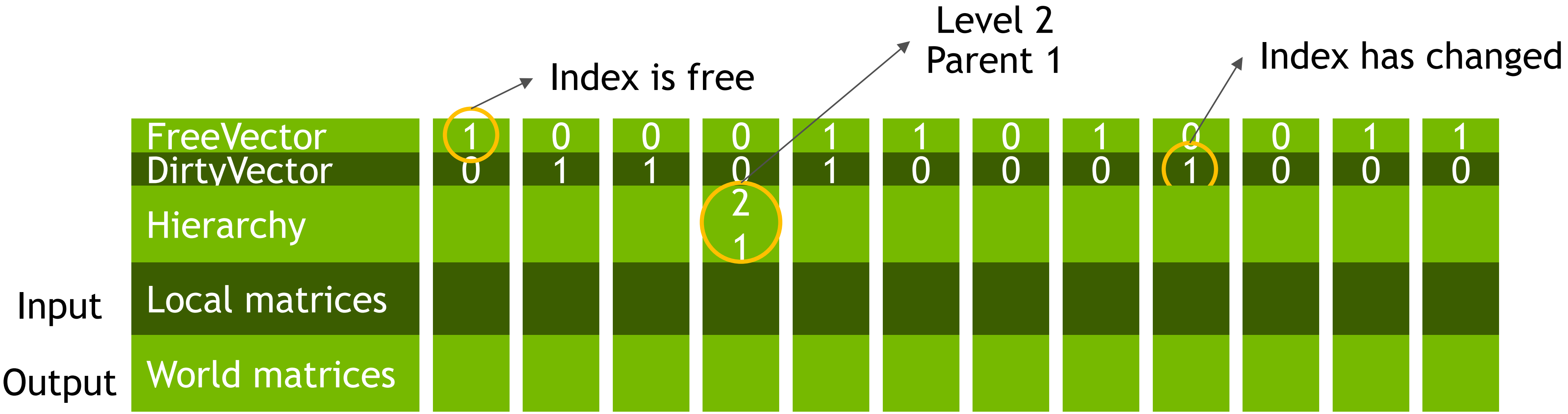
Interface

```
class TransformTree {
public:
  TransformIndex addTransform(TransformIndex parent, Mat44f const & local);
  void removeTransform(TransformIndex transformIndex);

  void setLocalTransform(TransformIndex parent, Mat44f const & local);

  // getting world matrices will have a cost if data is on the GPU
  Mat44f   const & getWorldTransform(TransformIndex index);

  // interface, implementation for CPU or GPU (CUDA/Vulkan/OpenGL)
  virtual void process() = 0;
};
```

TRANSFORM TREE

General Data Structure

Common data structure has 4 arrays



	FreeVector	1	0	0	0	1	1	0	1	0	0	1	1
	DirtyVector	0	1	1	0	1	0	0	0	1	0	0	0
	Hierarchy				2 1								
Input	Local matrices												
Output	World matrices												

Keep everything as local as possible, no pointer chasing

IMPLEMENTATION

CPU

CPU implementation keeps list of indices for each level to minimize traversal

```
void process()
{
    for (auto level : levels) {
        for (auto index : level.indices) {
            if (dirtyLocal[index] || dirtyWorld[index.parent]) {
                world[index] = local[index] * world[index.parent];
                dirtyWorld[index] = true;
            }
        }
    }
    notify(dirtyWorld);
    clear(dirtyLocal);
    clear(dirtyWorld);
}
```

RESULTS

CPU

CPU Xeon-E5 2630-v3 processes ~15 mio transforms per second

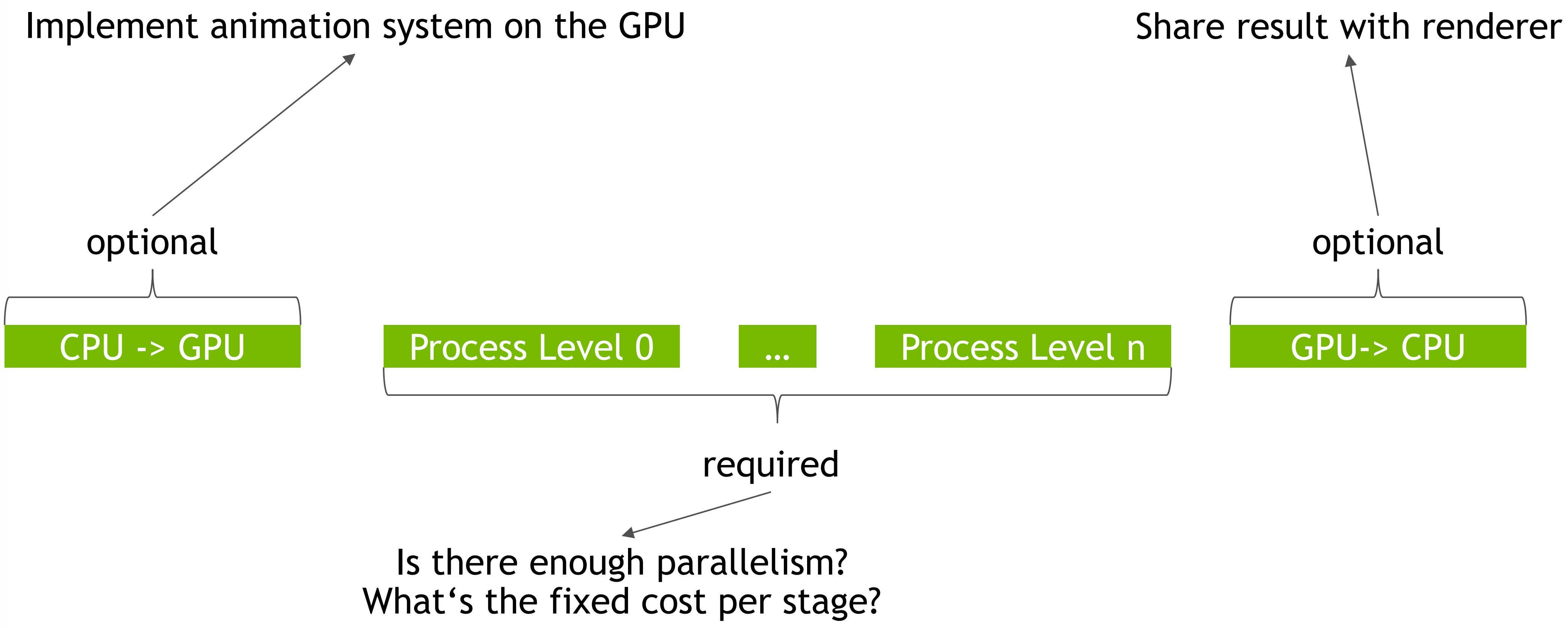
Assume 1ms budget for the transform hierarchy -> 16k transforms possible

Problem is somewhere else

- Pass updates transforms through pipeline on GPU
- Cost can be multiple times the cost of this update

Is it possible to keep transform computations completely on GPU?

CUDA IMPLEMENTATION



Implement animation system on the GPU

Share result with renderer

optional

optional

CPU -> GPU

Process Level 0

…

Process Level n

GPU-> CPU

required

Is there enough parallelism?
What‘s the fixed cost per stage?

CUDA IMPLEMENTATION

Paralellism

Quadro K6000 can execute 2,880 threads in parallel

How to sature this high number of threads?

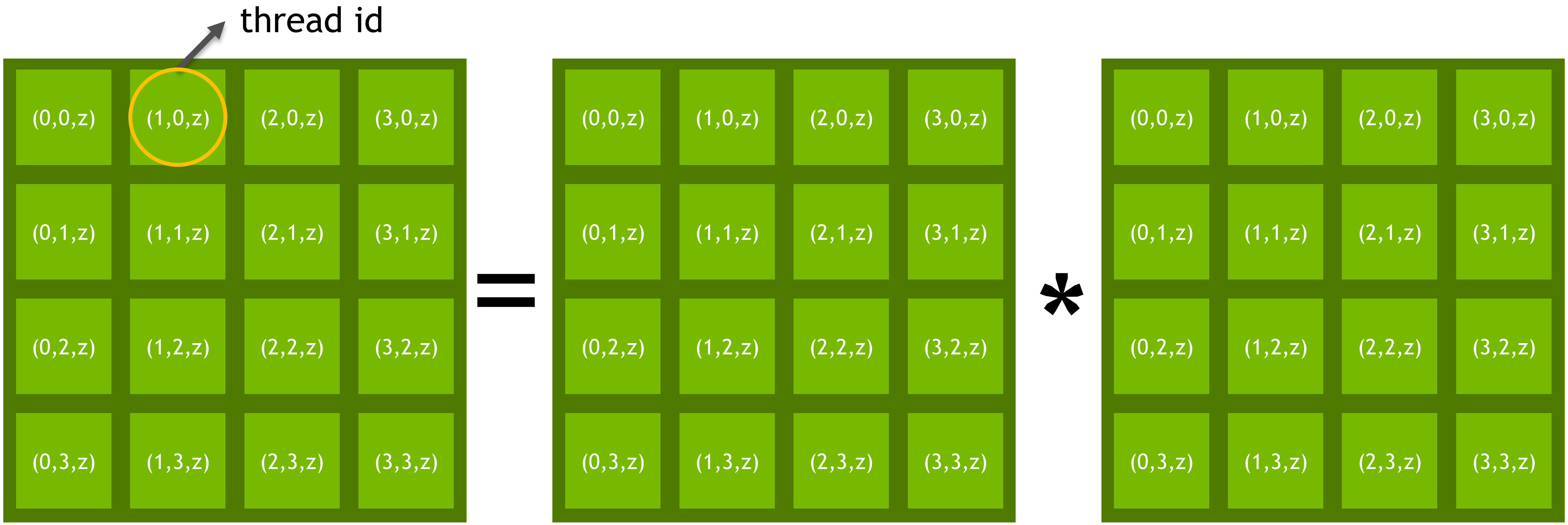
One matrix multiplication per thread is not enough

- Compute one matrix component per thread
- Increases number of required threads by 16 ☺

PROPAGATION

Matrix multiplication

Each half warp (16 threads) execute one multiplication



thread id

(0,0,z)	(1,0,z)	(2,0,z)	(3,0,z)
(0,1,z)	(1,1,z)	(2,1,z)	(3,1,z)
(0,2,z)	(1,2,z)	(2,2,z)	(3,2,z)
(0,3,z)	(1,3,z)	(2,3,z)	(3,3,z)

=

(0,0,z)	(1,0,z)	(2,0,z)	(3,0,z)
(0,1,z)	(1,1,z)	(2,1,z)	(3,1,z)
(0,2,z)	(1,2,z)	(2,2,z)	(3,2,z)
(0,3,z)	(1,3,z)	(2,3,z)	(3,3,z)

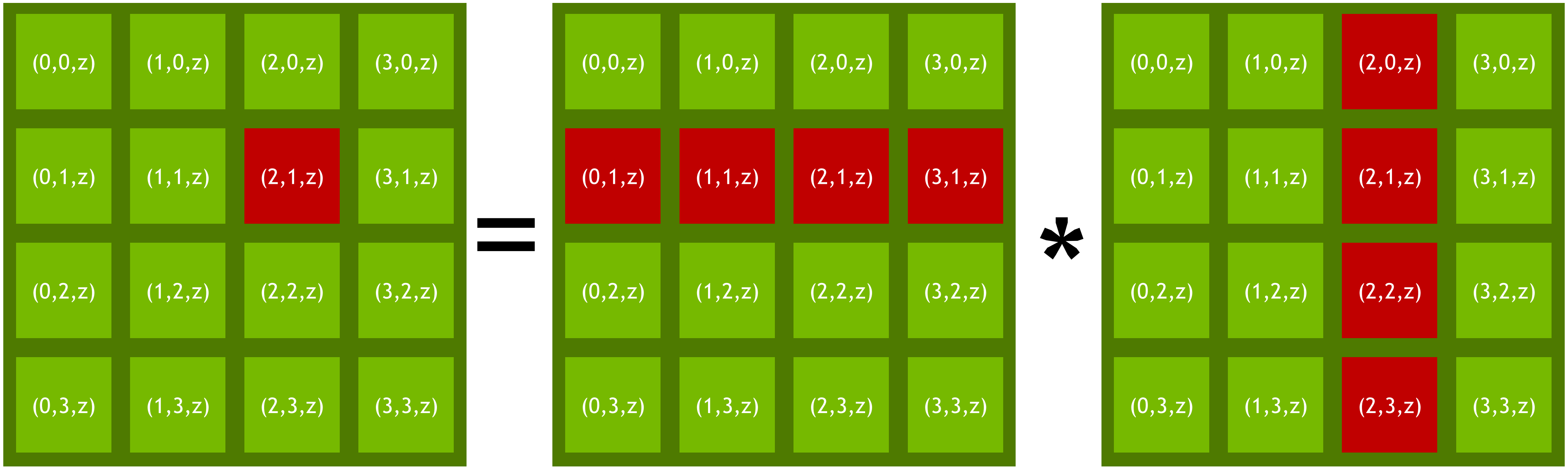
*

(0,0,z)	(1,0,z)	(2,0,z)	(3,0,z)
(0,1,z)	(1,1,z)	(2,1,z)	(3,1,z)
(0,2,z)	(1,2,z)	(2,2,z)	(3,2,z)
(0,3,z)	(1,3,z)	(2,3,z)	(3,3,z)

PROPAGATION

Matrix multiplication

Each thread computes one element of the matrix



(0,0,z)	(1,0,z)	(2,0,z)	(3,0,z)
(0,1,z)	(1,1,z)	**(2,1,z)**	(3,1,z)
(0,2,z)	(1,2,z)	(2,2,z)	(3,2,z)
(0,3,z)	(1,3,z)	(2,3,z)	(3,3,z)

=

(0,0,z)	(1,0,z)	(2,0,z)	(3,0,z)
(0,1,z)	**(1,1,z)**	**(2,1,z)**	**(3,1,z)**
(0,2,z)	(1,2,z)	(2,2,z)	(3,2,z)
(0,3,z)	(1,3,z)	(2,3,z)	(3,3,z)

*

(0,0,z)	(1,0,z)	**(2,0,z)**	(3,0,z)
(0,1,z)	(1,1,z)	**(2,1,z)**	(3,1,z)
(0,2,z)	(1,2,z)	**(2,2,z)**	(3,2,z)
(0,3,z)	(1,3,z)	**(2,3,z)**	(3,3,z)

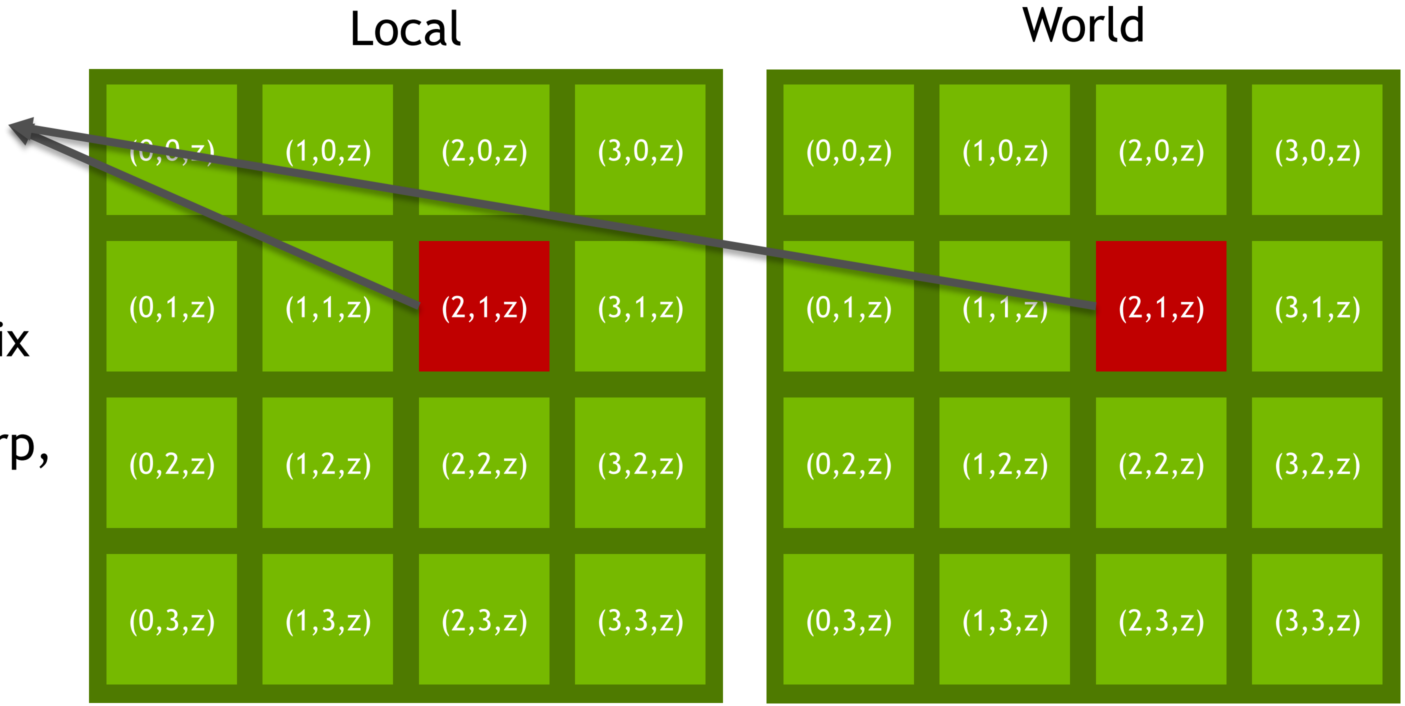
PROPAGATION

Matrix Multiplication

Thread (t.x, t.y, z) reads one local and world component of matrix

Coalesced read
-> One memory transaction per matrix

Whole matrix state is now in half warp, can be distributed with __shfl



Local

(0,0,z)	(1,0,z)	(2,0,z)	(3,0,z)
(0,1,z)	(1,1,z)	(2,1,z)	(3,1,z)
(0,2,z)	(1,2,z)	(2,2,z)	(3,2,z)
(0,3,z)	(1,3,z)	(2,3,z)	(3,3,z)

World

(0,0,z)	(1,0,z)	(2,0,z)	(3,0,z)
(0,1,z)	(1,1,z)	(2,1,z)	(3,1,z)
(0,2,z)	(1,2,z)	(2,2,z)	(3,2,z)
(0,3,z)	(1,3,z)	(2,3,z)	(3,3,z)

PROPAGATION

Multiplication

Full local & world state known through shuffle
Each thread can grab the required values per iteration

```
world(x,y,z) = __shfl(local, base1 + 0) * __shfl(world, base2 + 0)
             + __shfl(local, base1 + 1) * __shfl(world, base2 + 4)
             + __shfl(local, base1 + 2) * __shfl(world, base2 + 8)
             + __shfl(local, base1 + 3) * __shfl(world, base2 + 12);
```

(Tid.y & 3) * 4 + (Tid.z & 1) * 16

Tid.x + (Tid.z & 1) * 16

CUDA IMPLEMENTATION

ALGORITHM

```
void process()
{
    upload();
    for (auto level : levels) { // kernel launch per level
        for (auto index : level.indices)  { // warp id specified index
            if (dirtyLocal[index] || dirtyWorld[index.parent]) {
                parallelMultiply(world[index],local[index],world[index.parent]);
                if (threadIdx.x == 0 && threadIdx.y == 0) //mark dirty only once per matrix
                 atomicOr(dirtyWorld[index / 32], 1 << (index & 31); // atomic, avoid conflicts
            }
        }
    }
    download();
    notify(dirtyWorld);
    clear(dirtyLocal);
    clear(dirtyWorld);
}
```

RESULTS

Matrix Multiplication on GPU

Current version can do ~300m transforms/s on a K6000

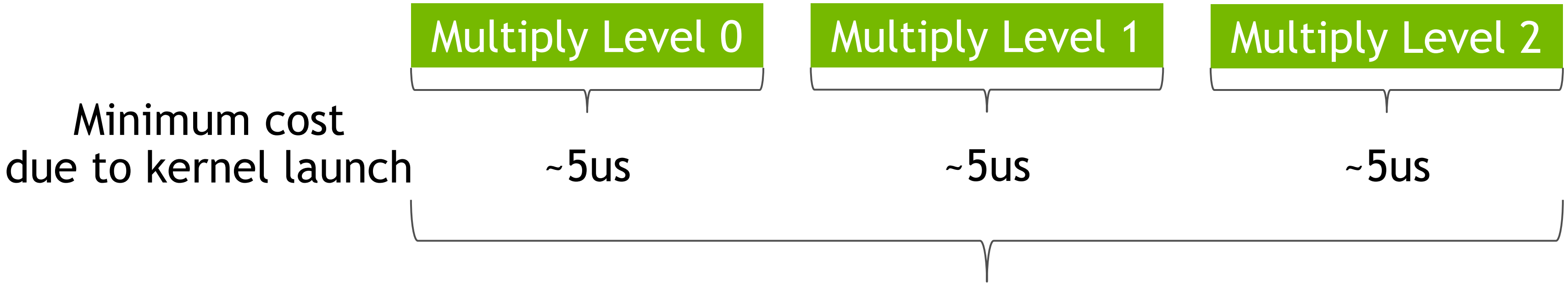
-> bw limited by dirty bits and hierarchy, not transforms

-> no coalesced reads, 128 byte transactions are being generated

Solving inefficient hierarchy memory access pattern could bring ~900m transforms/s

Work in Progress

GPU IMPLEMENTATION



Multiply Level 0

Multiply Level 1

Multiply Level 2

Minimum cost
due to kernel launch

~5us

~5us

~5us

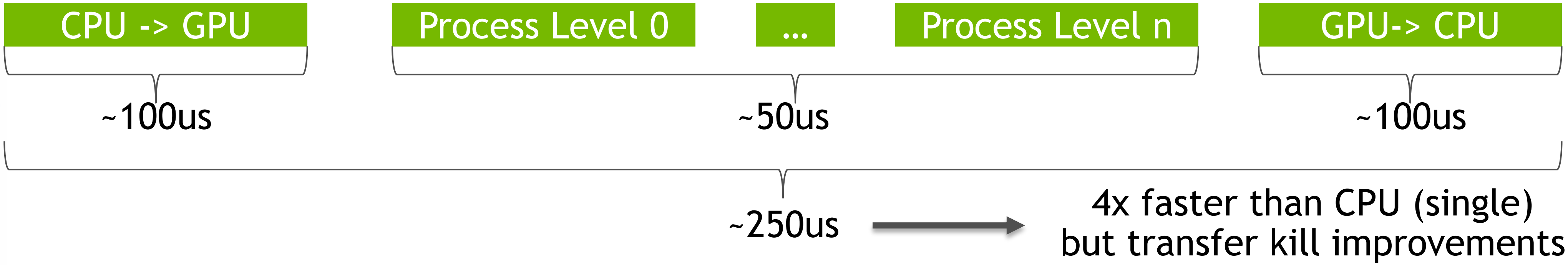
Total minimum cost up 15us for 3 levels, deep hierarchies might be bad

There‘re ways to reduce the cost, not yet addressed

CUDA IMPLEMENTATION

Results

worst case
16k matrices, all changed
2 levels in the hierarchy
Result required on CPU

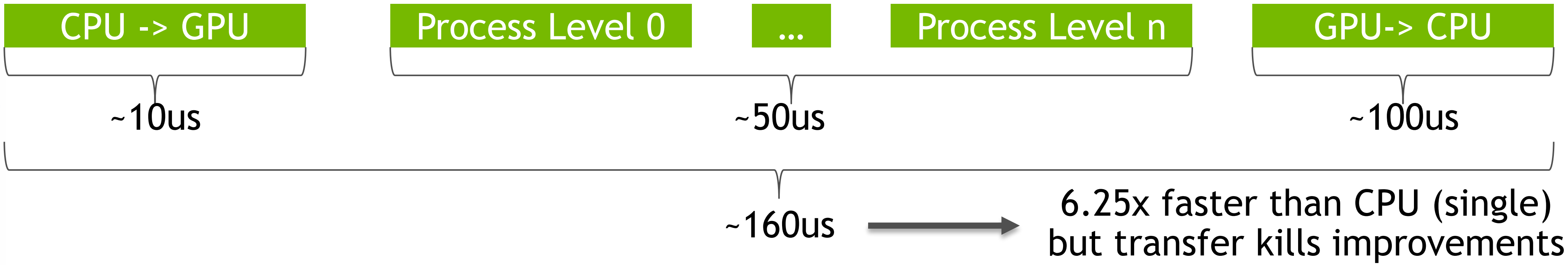


4x faster than CPU (single)
but transfer kill improvements

CUDA IMPLEMENTATION

Results

Medium case
1 matrix changed (top level)
2 levels in the hierarchy
Result required on CPU



CPU -> GPU

Process Level 0

...

Process Level n

GPU-> CPU

~10us

~50us

~100us

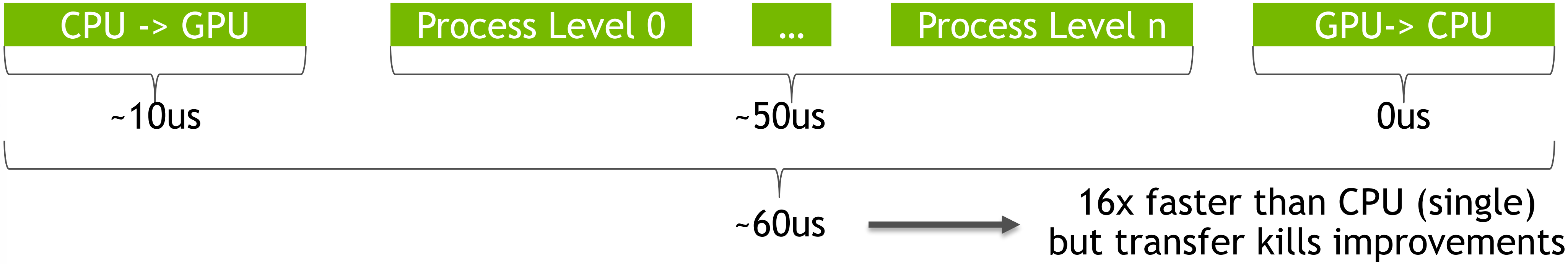
~160us

6.25x faster than CPU (single)
but transfer kills improvements

CUDA IMPLEMENTATION

Results

Medium case
1 matrix changed (top level)
2 levels in the hierarchy
Result not required on CPU



16x faster than CPU (single)
but transfer kills improvements

DATA ON GPU – USE CASES

Graphics Interop

No need to transfer data from CPU to GPU

Saves PCI-E bandwidth (~100us for 16k matrices)

Graphics usually needs inverse tranpose

Compute on GPU, saves CPU time again

Saves even more PCI-E bandwidth (~100us for 16k matrices)

DATA ON GPU – USE CASES

CULLING

Frustum culling

Quite efficient if data is already on GPU

Bounding box generation

For near/far plane computation bounding box of scene might be required

CONCLUSION / FUTURE

Transform hierarchy can be evaluated on the GPU quite fast

- Result is required on CPU -> gain is limited due to transfer time
  - Solvable with interop or by moving algorithm to Vulkan/OpenGL
- Input data comes from CPU -> gain might be limited depending on #changes
  - Animate matrices on the GPU

GPU TECHNOLOGY CONFERENCE

April 4-7, 2016 | Silicon Valley

THANK YOU

JOIN THE CONVERSATION

#GTC16

PRESENTED BY

NVIDIA.