Fast many-core solvers for the Eikonal equations in cardiovascular simulations.

Daniel Ganellari

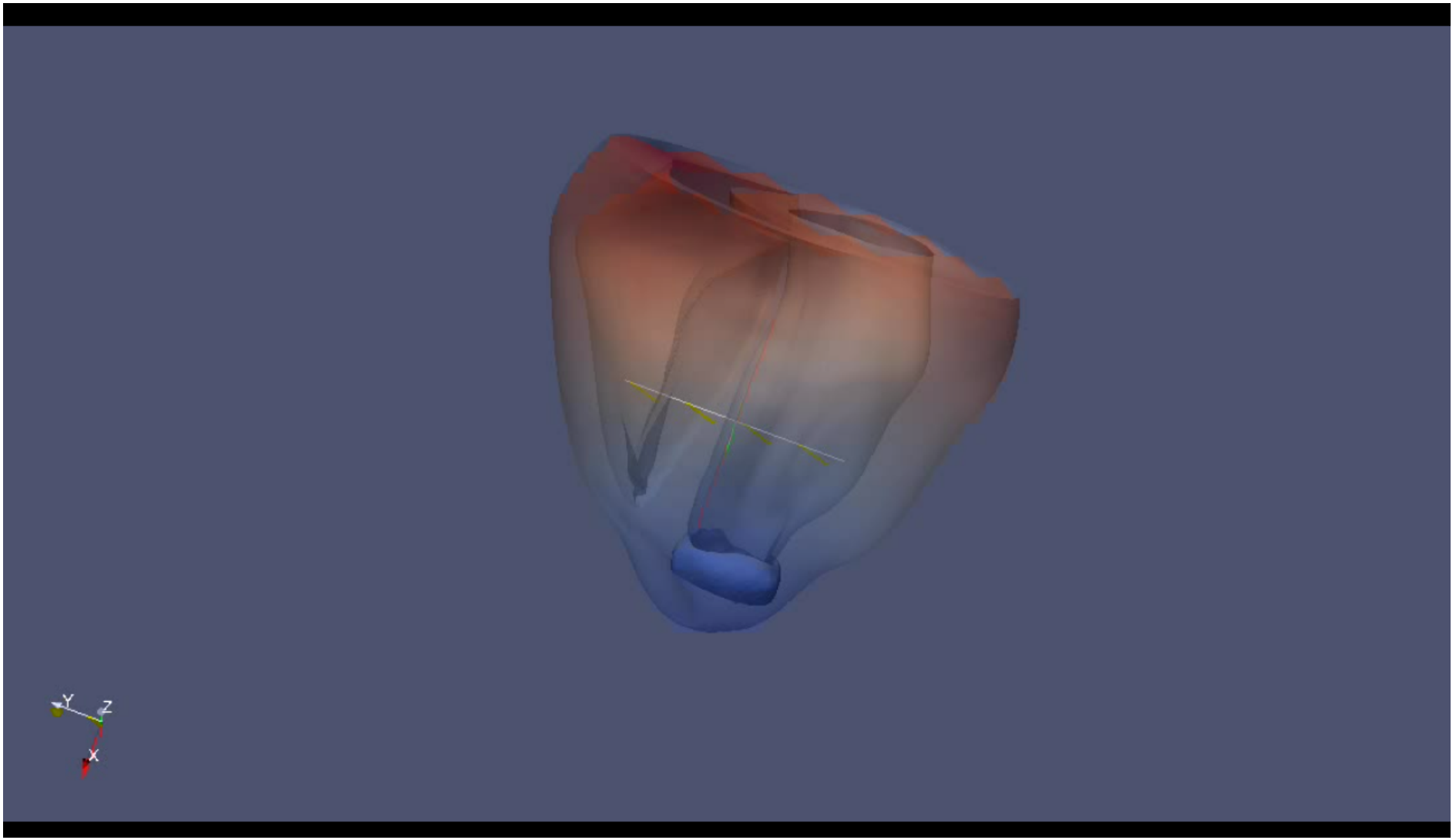
Karl-Franzens-University Graz
Institute for Mathematics and Scientific Computing

daniel.ganellari@edu.uni-graz.at

April 26, 2017

Overview

Isosurface Animation



Eikonal Equation

Non-linear PDE encountered in problems of wave propagation.

$$\begin{cases} H(x, \nabla\phi) = \sqrt{(\nabla\phi)^T M(\nabla\phi)} = 1, & \forall x \in \Omega \subset \mathbb{R}^3 \quad (1) \\ \phi(x) = B(x), & \forall x \in B \subset \Omega \quad (2) \end{cases}$$

- Ω - 3D domain approximated by planar side-sided tetrahedralization.
- $\phi(x)$ - The shortest time needed to travel from the boundary to x inside Ω.
- M(x) - 3x3 symmetric positive-definite matrix encoding speed information on Ω.
- B - set of smooth boundary conditions which adhere to the consistency requirements of the PDE.

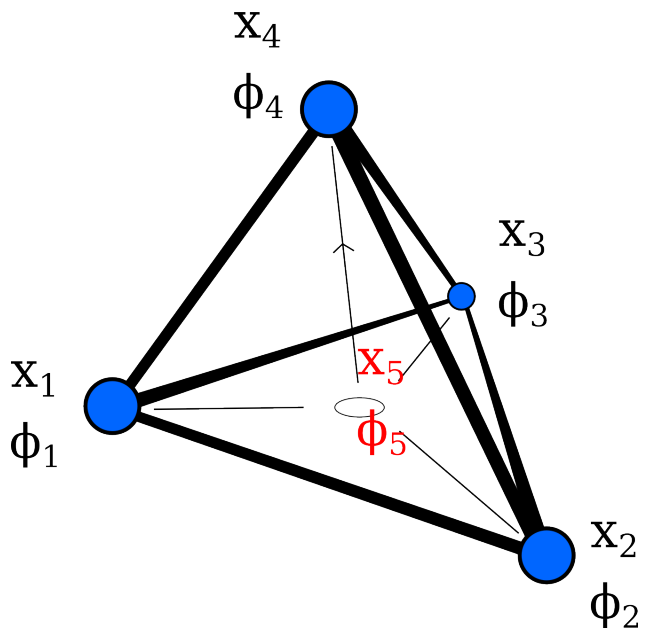
FIM Description (Local Solver and Update Scheme)

The goal is to find the location of x_5 that minimizes ϕ_4 (Fermat's Principle).

Denote the travel time from x_5 to x_4:

$$\phi_{5,4} = \phi_4 - \phi_5 = \sqrt{e_{5,4}^T M e_{5,4}} \quad (3)$$

The solution (travel time) $\phi(x)$ at each vertex is computed from the linear approximations on its one-ring tetrahedra.



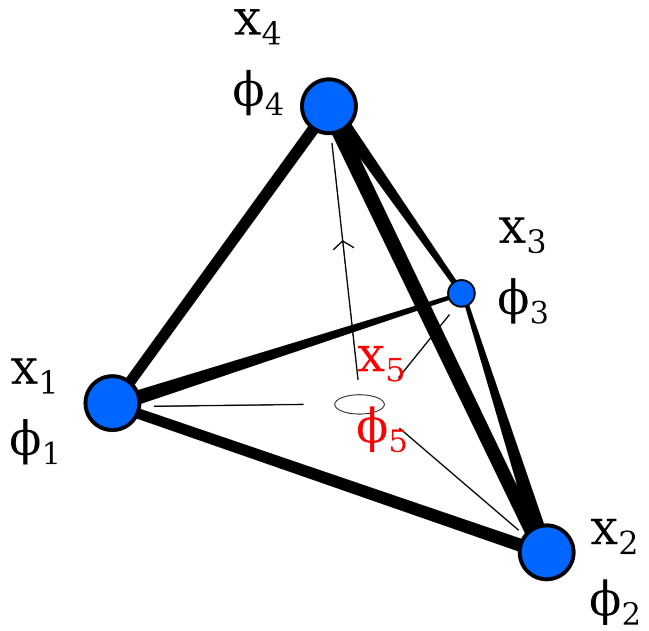
FIM Description (Local Solver and Update Scheme)

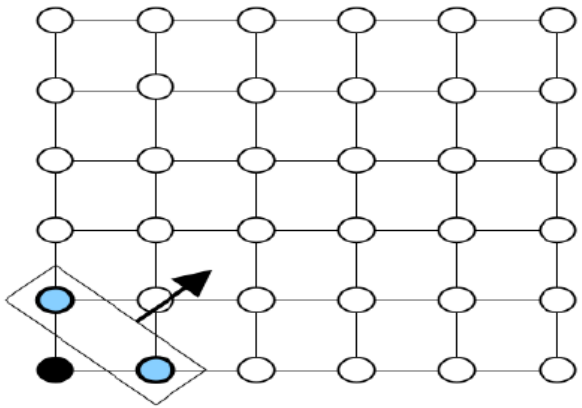
The goal is to find the location of x_5 that minimizes ϕ_4 (Fermat's Principle).

Denote the travel time from x_5 to x_4:

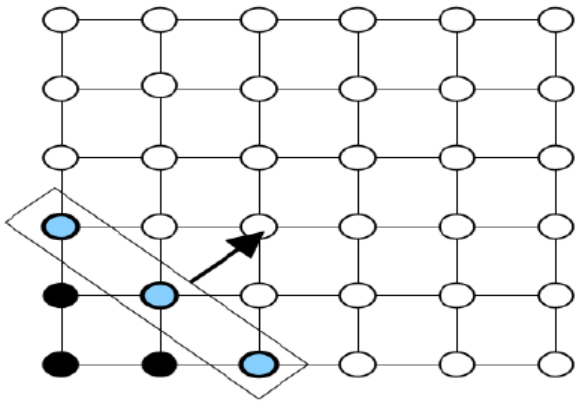
$$\phi_{5,4} = \phi_4 - \phi_5 = \sqrt{e_{5,4}^T M e_{5,4}} \quad (3)$$

The solution (travel time) $\phi(x)$ at each vertex is computed from the linear approximations on its one-ring tetrahedra.

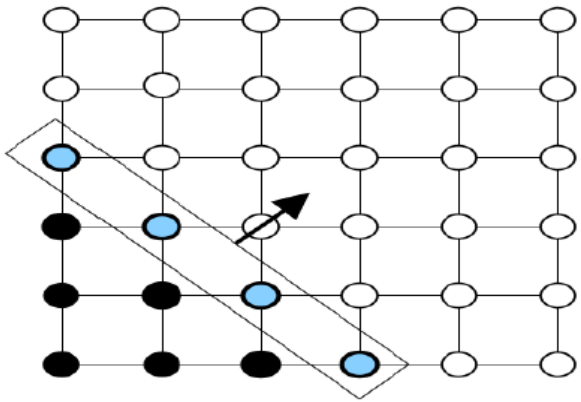




(a) Initial stage



(b) After first update



(c) After second update

Results

Hardware Specifications

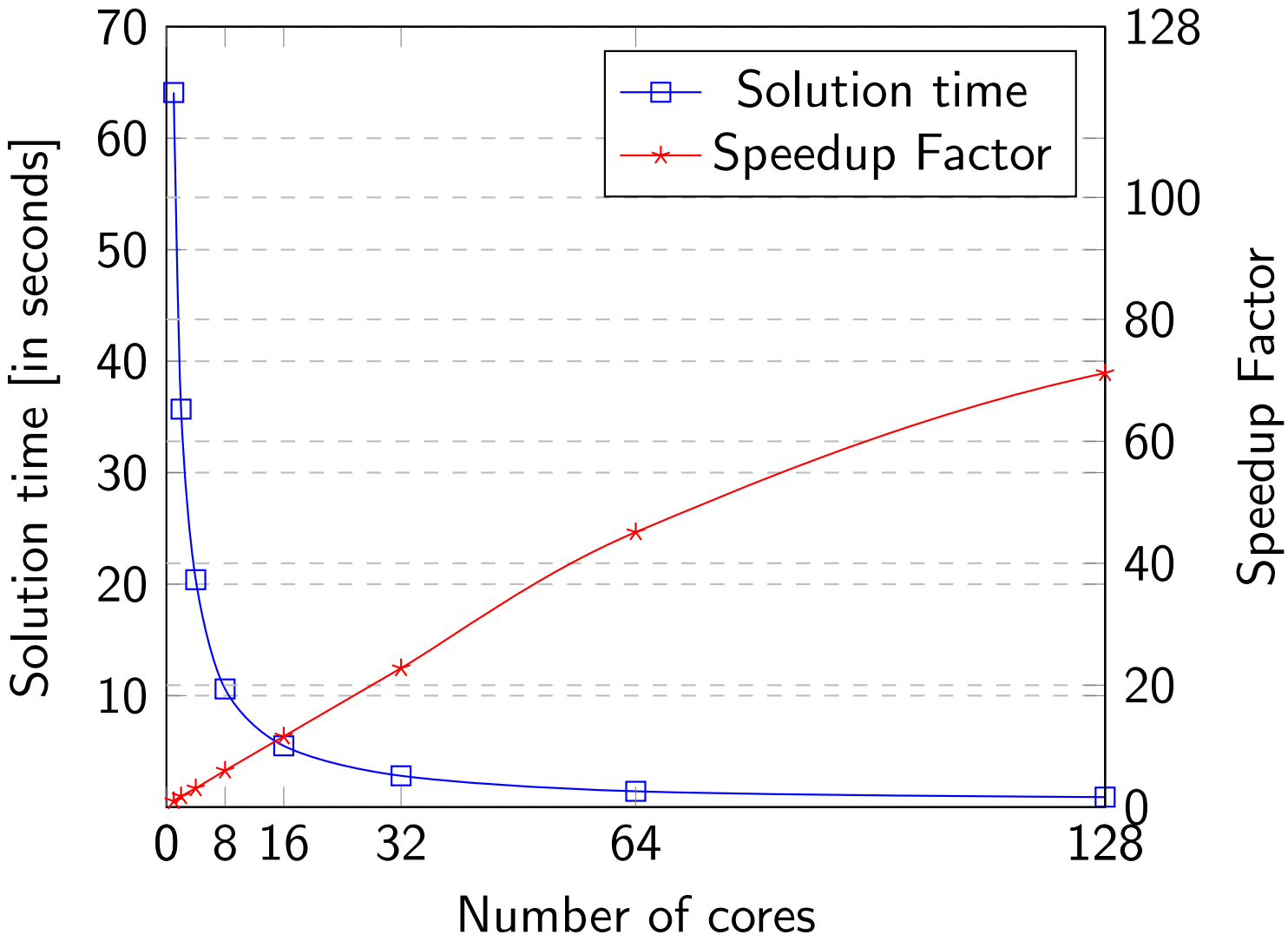
CUDA implementation tested on GeForce GTX 1080 Pascal. OpenMP implementation tested on KNL, Intel Xeon Phi 1.3 GHz , 64 cores and 256 threads.

Result comparison between CUDA and OpenMP implementations

Meshes	**# Tetrahedras**	**CUDA**	**OpenMP**
TBunnyC	3,073,529	0.65 sec	0.68 sec
Human Heart	24,400,999	5.3 sec	5.6 sec

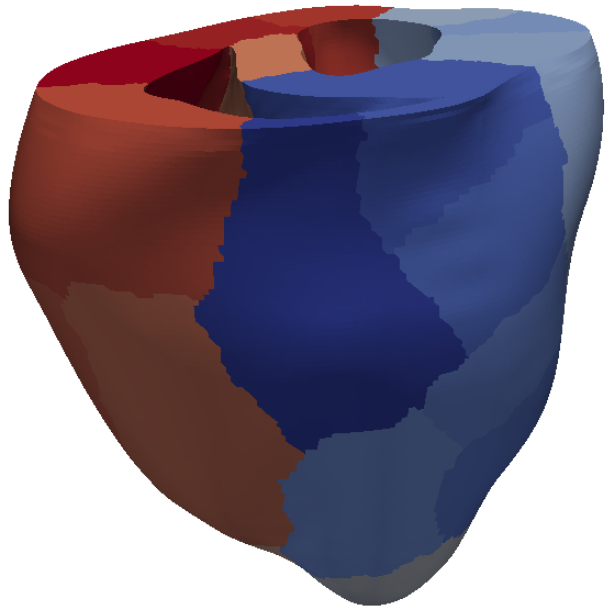
OpenMP Scaling Results

Scaling results on Intel Xeon Phi 1.3GHz for **TBunnyC** mesh.



Domain Decomposition with CUDA

- It will use the Block-Wide Scan from CUB
  - The sub domain items fit into the Shared Memory
- It will run on a Single GPU
  - Management strategy to run the active sub-domains into the available blocks



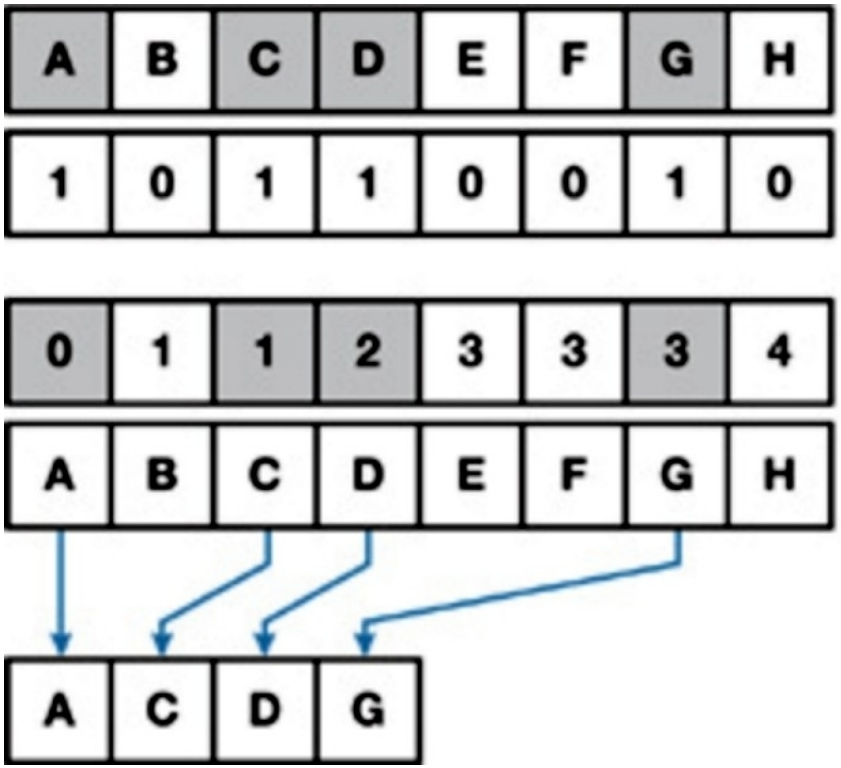
- Solution vector of each sub domain fits into the shared memory
  - Its implementation increases the execution dependency

0.234	1.345	5.654	6.123	6.805	2.017	2.514	6.251	6.463

Solution Vector **U[0,VertexSize]**

SCAN

Stream compaction requires two steps, a scan and a scatter.



Preserve the gray elements

Build Predicate

Scan

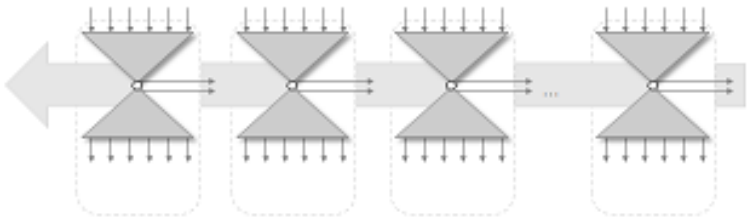
Scatter gray inputs to output, using scan result as scatter address

When to use compaction?

- Large Number of elements to compact
- The computation on each surviving element is expensive

CUB Device-Wide Primitive Scan

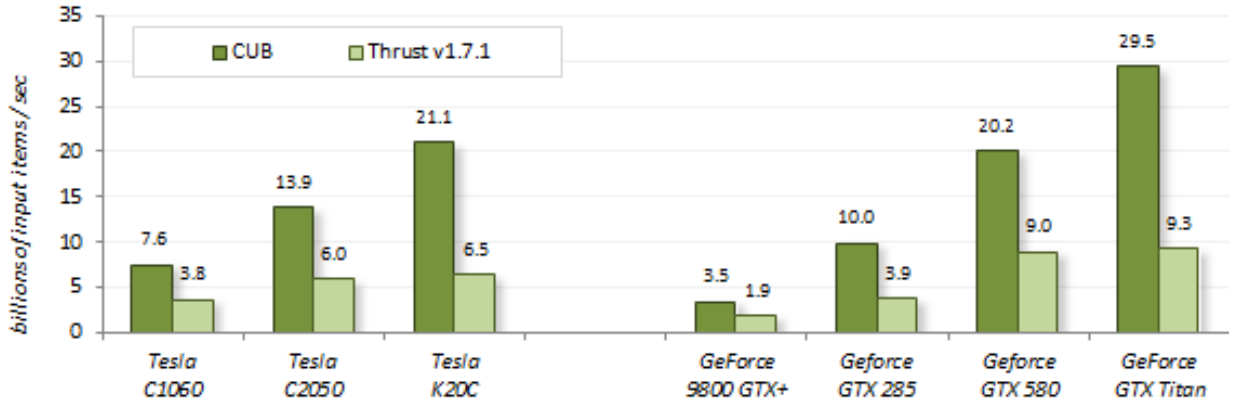
Provides device-wide, parallel operations for computing a prefix scan.



Can be called within kernel code on devices in which CUDA dynamic parallelism is supported.

Performance

- The work-complexity of prefix scan as a function of input size is linear.



Performance-portability comparison of device-wide prefix scan (32M int32 items)

Code Snippet

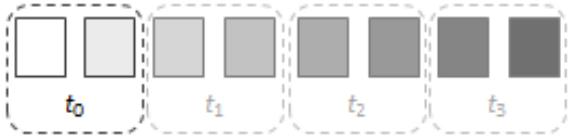
The code snippet below illustrates the exclusive prefix sum of an int device vector.

```
#include <cub/cub.cuh>   // or equivalently <cub/device/device_scan.cuh>

// Declare, allocate, and initialize device-accessible pointers for input and output
int  num_items;      // e.g., 7
int  *d_in;          // e.g., [8, 6, 7, 5, 3, 0, 9]
int  *d_out;         // e.g., [ ,  ,  ,  ,  ,  ,  ]
...

// Determine temporary device storage requirements
void     *d_temp_storage = NULL;
size_t   temp_storage_bytes = 0;
cub::DeviceScan::ExclusiveSum(d_temp_storage, temp_storage_bytes, d_in, d_out, num_items);

// Allocate temporary storage
cudaMalloc(&d_temp_storage, temp_storage_bytes);

// Run exclusive prefix sum
cub::DeviceScan::ExclusiveSum(d_temp_storage, temp_storage_bytes, d_in, d_out, num_items);

// d_out s<-- [0, 8, 14, 21, 26, 29, 29]
```

No total sum computed!

CUB Block-Wide Primitive Scan

Block arrangement of contiguous 2 items per thread



Prefix sum of 256 integer items partitioned as above across 128 threads.

```
#include <cub/cub.cuh>   // or equivalently <cub/block/block_scan.cuh>

__global__ void ExampleKernel(...)
{
    // Specialize BlockScan for a 1D block of 128 threads on type int
    typedef cub::BlockScan<int, 128> BlockScan;

    // Allocate shared memory for BlockScan
    __shared__ typename BlockScan::TempStorage temp_storage;

    // Obtain a segment of consecutive items that are blocked across threads
    int thread_data[4];
    ...

    // Collectively compute the block-wide exclusive prefix sum
    int block_aggregate;
    BlockScan(temp_storage).ExclusiveSum(thread_data, thread data, block aggregate);
```

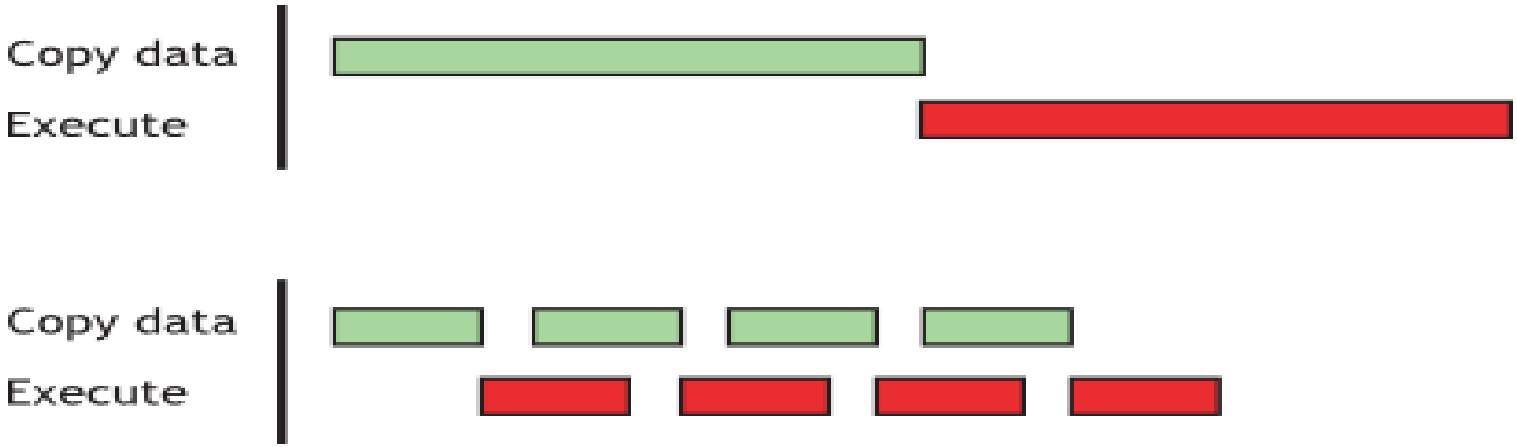
Performance is increased with the increased granularity ITEMS_PER_THTREAD.

Asynchronous and Overlapping Transfer with Computation

Concurrent copy and execute

```
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, stream1);
kernel<<<grid, block, 0, stream2>>>(otherData_d);
```

Staged concurrent copy and execute




```
size=N*sizeof(float)/nStreams;
for (i=0; i<nStreams; i++) {
    offset = i*N/nStreams;
    cudaMemcpyAsync(a_d+offset, a_h+offset, size, dir, stream[i]);
    kernel<<<N/(nThreads*nStreams), nThreads, 0,
             stream[i]>>>(a_d+offset);
}
```

CUDA Events

Asynchronous cuda memset to 0.

```
void setZero(cudaStream_t stream)
{
    if(m_virtualSize > 0)
        checkCudaErrors(cudaMemsetAsync(d_ptr, 0, m_virtualSize*sizeof(T), stream));
}
```

Explicit synchronization using an event.

```
d_ElemNbHNr.resize(2 * nbHElemSize);
d_ElemNbHNr.setZero(2*nbHElemSize,s2);
checkCudaErrors(cudaEventRecord(ev2,s2));

checkCudaErrors(cudaMemcpyToSymbolAsync(d_nbHElemSize, &nbHElemSize, sizeof(nbHElemSize),0,cudaMemcpyHostToDevice,s1));

threads = 1024;
blocks = nbHPtsSize / threads+1;

    //scatter the neighboring nodes based on the scatter addresses
calculateConvergedElemNbH<<<blocks,threads,0,s1>>>(d_ptsNbHList.get(),d_PtsNbHNr.get(),d_scannedPtsNbH.get(),d_converge

checkCudaErrors(cudaStreamWaitEvent(s1,ev2,0));
//calculate the number of neighboring tetrahedrons for each of the neighboring nodes of the converged active list.
calculateElemNbHNr<<<blocks,threads,0,s1>>>(d_ptsNbHList.get(), d_ElemNbHNr.get(),d_n2edsp);
```

References

Daniel Ganellari and Gundolf Haase (2016)

Fast many-core solvers for the Eikonal equations in cardiovascular simulations.

2016 International Conference on High Performance Computing Simulation (HPCS), pp.278-285, 2016

Zhisong Fu, Robert M. Kirby, and Ross T. Whitaker (2013)

Fast iterative method for solving the eikonal equation on tetrahedral domains.

SIAM J. Sci. Comput., vol.35, no.5, pp.C473-C494, 2013

W.-K. Jeong and R. T. Whitaker (2008)

A fast iterative method for eikonal equations.

SIAM J. Sci. Comput., vol.30, pp.2512-2534, 2008

J. Qian and Y.-T. Zhang and H.-K. Zhao (2007)

Fast sweeping methods for eikonal equations on triangulated meshes.

SIAM J. Numer. Anal., pp.83-107, 45, 2007

Duane Merrill and Michael Garland (2016)

Single-pass Parallel Prefix Scan with Decoupled Look-back

https://research.nvidia.com/publication/single-pass-parallel-prefix-scan-decoupled-look-back

Thank you for your attention!