Computing and Visualization in Science (2018) 19:3–18
https://doi.org/10.1007/s00791-018-0288-z

SPECIAL ISSUE FEM SYMPOSIUM 2017

A massively parallel Eikonal solver on unstructured meshes

Daniel Ganellari[1] · **Gundolf Haase**[1] · **Gerhard Zumbusch**[2]

Received: 17 June 2017 / Accepted: 3 November 2017 / Published online: 2 February 2018

Abstract
Algorithms for the numerical solution of the Eikonal equation discretized with tetrahedra are discussed. Several massively parallel algorithms for GPU computing are developed. This includes domain decomposition concepts for tracking the moving wave fronts in sub-domains and over the sub-domain boundaries. Furthermore a low memory footprint implementation of the solver is introduced which reduces the number of arithmetic operations and enables improved memory access schemes. The numerical tests for different meshes originating from the geometry of a human heart document the decreased runtime of the new algorithms.

Keywords Eikonal equation · GPU · Domain decomposition · Tetrahedral mesh · Parallel algorithm

1 Introduction

Recent work in [1–3] has shown that building an efficient 3D tetrahedral Eikonal solver for multi-core and SIMD (single instruction multiple data) architectures poses many challenges. It is important to keep the memory footprint low in order to reduce the costly memory accesses and achieve a good computational density on GPUs and other SIMD architectures with limited memory and register capabilities. But also in the general case of parallel architectures with limited high-bandwidth memory, the memory footprint of the local solver becomes a bottleneck to performance. We addressed the implementation of an Eikonal solver for Shared Memory (OpenMP) and for streaming architectures in CUDA with a low memory footprint in our previous work [2] wherein we achieved already a very short execution time of the algorithm and good quality results. We use the fast iterative method (FIM) [1,4,5] for solving the Eikonal equation, especially the version by Fu, Kirby and Whitaker [1] wherein the memory footprint has been reduced by storing temporarily the inner products needed to compute the 3D solution of one vertex. This step is performed in the wave front computations and some data have to be transferred from main memory to compute the solution for each tetrahedron. The application background for our paper are cardiovascular simulations that use Eikonal solvers for determining the initial excitation pattern in a human heart [6].

We address a new method to reduce the memory footprint of the Eikonal solver, see Sect. 3. We will reduce the number of memory transfers as well as the local memory footprint by precomputing all the needed inner products for each tetrahedron in reference orientation, 18 floats in total. The mapping problem of the precomputed data to the orientation of the tetrahedron under investigation is also addressed. A second step replaces 12 memory accesses per tetrahedron by on-the-fly computations from 6 given data per tetrahedron which reduces the memory footprint to these 6 numbers in total.

The following sections introduce our algorithms for CPU (sequential C), for shared memory parallelization (OpenMP) and for GPU accelerators (CUDA). Data structures and algorithms for streaming architectures are described in detail and we show that a careful management of data allows a higher computational density on the GPU which yields then to a satisfactory speed up compared to the OpenMP implementation.

The domain decomposition approach to solve the Eikonal equation is proposed in Sect. 6. We present two different

Communicated by Thomas Apel.

Support from FWF project F32-N18 and Erasmus Mundus JoinEUsee PENTA scholarship.

✉ Daniel Ganellari
ganellari.d@gmail.com

Gundolf Haase
gundolf.haase@uni-graz.at

[1] Institute for Mathematics and Scientific Computing, University of Graz, Heinrichstr. 36, 8010 Graz, Austria

[2] Institut für Angewandte Mathematik, Friedrich-Schiller-Universität Jena, 07743 Jena, Germany

strategies of load balancing. The first approach dynamically maps one sub-domain to several thread blocks in CUDA, using a sequence of different kernel invocations. It takes better advantage of the GPU shared memory since it shares the workload of one sub-domain between potentially many thread blocks exploiting in this way the total shared memory space. The second approach uses a single kernel on a single thread block per sub-domain. It avoids host synchronization and memory transfers nearly completely. Numerical tests and a discussion of the observed performance gains are presented in Sect. 7 wherein we investigate GPUs using CUDA [7] and CPUs using OpenMP.

2 Mathematical description of the Eikonal equation

The Eikonal equation is a special case of the Hamilton-Jacobi partial differential equations (PDEs), encountered in problems of wave propagation

$$\begin{cases} |\nabla\varphi(x)| = F(x), & x \in \Omega \quad (1) \\ \varphi = 0, & x \in \partial\Omega_D \quad (2) \end{cases}$$

where Ω is an open set in $\mathbb{R}^n$ with well-behaved boundary, $F(x)$ is a function with positive values and ∇ denotes the gradient. The term $|.|$ represents the Euclidean norm taking into account the anisotropic symmetric velocity information $M(x) : \mathbb{R}^3 \mapsto \mathbb{R}^3$ as a property of the domain Ω, i.e.,

$$|v(x)|^2 := \|v(x)\|_M^2 = \langle v(x), v(x)\rangle_M = v(x)^T \cdot M(x) \cdot v(x) \quad \forall x \in \Omega.$$

The velocity information has to fulfill $\sum_{i,j=1}^{3} M_{i,j}(x)\xi_i\xi_j \geq \overline{\mu}|\xi|^2 \quad \forall \xi \in \mathbb{R}^3 \quad \forall x \in \Omega$ with $\overline{\mu} > 0$.

Physically, the solution $\varphi(x)$ is the shortest time needed to travel from the boundary $\partial\Omega_D$ to x inside Ω, with $F(x)$ being the time cost (not speed) at x. The solution in the special case $F = 1$ expresses the signed distance from $\partial\Omega_D$. Reformulating (1) with the relation above results in the variational formulation of the Eikonal equation

$$\sqrt{(\nabla\varphi(x))^T M(x)\nabla\varphi(x)} = 1 \qquad x \in \Omega \quad (3)$$

describing a traveling wave through the domain Ω with given heterogeneous, anisotropic velocity information M. The solution $\varphi(x)$ denotes the time when the wave arrives at point x. The computational domain Ω for solving the Eikonal equation is discretized by planar-sided tetrahedrons whose vertices are the discretization points storing the discrete solution. The continuous solution is represented by a linear interpolation within each tetrahedron.

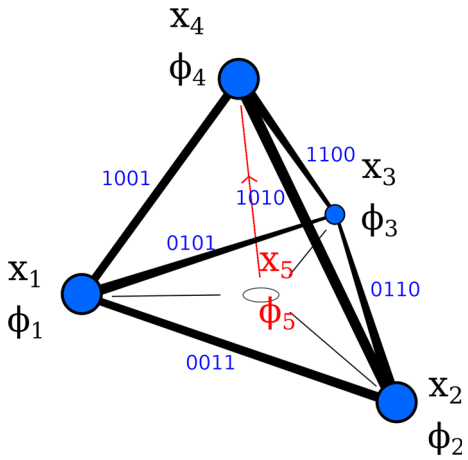

Fig. 1 Notations in the tetrahedron. Vector $e_{5,4}$ denotes the wave propagation direction that intersects with the triangle $\triangle 1, 2, 3$ at x_5

3 Local solver

We use the fast iterative method (FIM) [4,5] in its description by Fu, Kirby and Whitaker [1] as baseline Eikonal solver for our improvements in this paper. The original FIM is introduced in Sect. 3.1 and we modify it in Sect. 3.2 by precomputing the needed inner products for each tetrahedron. A further significant reduction in memory footprint is achieved in Sect. 3.3 by applying some analysis.

3.1 Fast iterative method

The FIM is based on local solvers for (3) in each tetrahedron. An upwind scheme is used to compute an unknown solution ϕ_4, assuming that the given values ϕ_1, ϕ_2, and ϕ_3 comply with the causality property of the Eikonal solutions [8]. Since we assume a constant speed function within each tetrahedron the arrival time ϕ_4 is determined by the time associated with that segment from x_5 to x_4 that minimizes the solution value at x_4, see Fig. 1. Therefore we have to determine the coordinates of that auxiliary point x_5 where the wave-front intersects first the plane defined by the vertices x_1, x_2 and x_3. If x_5 is located outside the surface triangle of the three nodes then x_5 is projected to the nearest boundary of this surface triangle, e.g., to edge (x_1, x_3), and ϕ_4 is calculated via a wave propagation in the triangle (x_1, x_3, x_4).

Based on Fermat's principle [9] the goal is to find locally that location of x_5 which minimizes the travel time from x_5 to x_4, i.e.,

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

We interpret x_5 as the center of mass of the surface and express it in barycentric coordinates as $x_5 = \lambda_1 x_1 + \lambda_2 x_2 + (1 - \lambda_1 - \lambda_2)x_3$. ϕ_5 is rewritten in the same way. If we plug

ϕ_5 into equation (4) then we get the following expression for ϕ_4:

$$\phi_4(\lambda_1,\lambda_2) = \lambda_1\phi_1+\lambda_2\phi_2+(1-\lambda_1-\lambda_2)\phi_3+\sqrt{e_{5,4}^T M e_{5,4}}. \tag{5}$$

In order to minimize ϕ_4, we have to calculate the partial derivatives of (5) with respect to λ_1, λ_2 and equate them to zero which results in a non-linear system of equations.

Kirby and Whitaker [1] built a low memory footprint solver by reducing the computations and memory storage based on the observation that $e_{5,4}$ can be expressed as:

$$e_{5,4} = x_4-x_5 = x_4-(\lambda_1 x_1+\lambda_2 x_2+\lambda_3 x_3) = [e_{1,3}\, e_{2,3}\, e_{3,4}]\lambda \tag{6}$$

with $\lambda = [\lambda_1\ \lambda_2\ 1]^T$. Hence they obtain the substitution into (5)

$$e_{5,4}^T M e_{5,4} = \lambda^T [e_{1,3}^T\, e_{2,3}^T\, e_{3,4}^T]^T M [e_{1,3}\, e_{2,3}\, e_{3,4}]\lambda = \lambda^T M'\lambda \tag{7}$$

with the symmetric matrix

$$M' := \begin{cases} \underline{\alpha} &= [e_{1,3}^T M e_{1,3},\ e_{2,3}^T M e_{1,3},\ e_{3,4}^T M e_{1,3}]^T \\ \underline{\beta} &= [e_{1,3}^T M e_{2,3},\ e_{2,3}^T M e_{2,3},\ e_{3,4}^T M e_{2,3}]^T \\ \underline{\gamma} &= [e_{1,3}^T M e_{3,4},\ e_{2,3}^T M e_{3,4},\ e_{3,4}^T M e_{3,4}]^T. \end{cases} \tag{8}$$

Finally, the optimization problem (5) transforms into the non-linear system of equations for λ_1 and λ_2:

$$\begin{cases} \phi_{1,3}\sqrt{\lambda^T M'\lambda} = \lambda^T\underline{\alpha} \\ \phi_{2,3}\lambda^T\underline{\alpha} = \phi_{1,3}\lambda^T\underline{\beta}. \end{cases} \tag{9}$$

Solving equations (9) needs the symmetric matrix M' requiring temporarily only six floats per tetrahedron. Additionally all the coordinates and the symmetric velocity matrix M ($4*3+6$ floats) have to be transferred from main memory with a (GPU) non-coalesced memory pattern.

3.2 Precomputing the inner products

In order to reduce the non-coalesced memory accesses whenever system (9) has to be solved in a tetrahedron we precompute all possible 18 inner products, listed in Table 1.

This approach does not reduce the memory footprint significantly (we still have to transfer 18 floats from main memory) but we achieve a reduction of the non-coalesced accesses to global memory. No computation of the scalar products is done at all during the wave front calculations now. Instead only the access of the needed values of M's is considered. All the accesses to matrix M and the edges are avoided, reducing in this way also the computations and the global memory access needed for these computations. The strike-through elements in Table 1 have no physical meaning in the tetrahedron and therefore these entries are never stored. Those entries needed for solving (9) in the reference configuration to determine ϕ_4, see also Fig. 1, are marked in blue.

Table 1 M-Scalar products stored in the 6×6 symmetric matrix T_M

0	1	2	3	4	5
$e_{1,2}^T M e_{1,2}$	$e_{1,2}^T M e_{1,3}$	$e_{1,2}^T M e_{2,3}$	$e_{1,2}^T M e_{1,4}$	$e_{1,2}^T M e_{2,4}$	~~$e_{1,2}^T M e_{3,4}$~~
	$e_{1,3}^T M e_{1,3}$	$e_{1,3}^T M e_{2,3}$	$e_{1,3}^T M e_{1,4}$	~~$e_{1,3}^T M e_{2,4}$~~	$e_{1,3}^T M e_{3,4}$
		$e_{2,3}^T M e_{2,3}$	~~$e_{2,3}^T M e_{1,4}$~~	$e_{2,3}^T M e_{2,4}$	$e_{2,3}^T M e_{3,4}$
			$e_{1,4}^T M e_{1,4}$	$e_{1,4}^T M e_{2,4}$	$e_{1,4}^T M e_{3,4}$
				$e_{2,4}^T M e_{2,4}$	$e_{2,4}^T M e_{3,4}$
					$e_{3,4}^T M e_{3,4}$

Table 2 Local Gray-code numbering of edges

x_4	x_3	x_2	x_1	edge: d	matrix: f(d)
0	0	1	1	3	0
0	1	0	1	5	1
0	1	1	0	6	2
1	0	0	1	9	3
1	0	1	0	10	4
1	1	0	0	12	5

$\mathbf{f(d)} = \text{round}(0.547826 * \mathbf{d} - 1.608695)$
$\mathbf{f_{int}(d)} = \mathbf{d}/2 - 1$

The challenge consist in accessing the needed 6 inner products for M' from the the 18 precomputed entries in Table 1 without branched code whenever we face a non-reference configuration, e.g., ϕ_i is the unknown value with $i \in \{1,2,3\}$. The needed tools are described in the remaining subsection.

3.2.1 Gray code like indexing

We use a local Gray-code [10] of 4 bits length to identify uniquely all possible objects in a tetrahedron, see Fig. 1. Each vertex $k \in \{1,2,3,4\}$ is represented by a 4-bit number 2^{k-1} with exactly one bit set and the Gray index of the connecting edge $e_{k,\ell}^{\text{Gray}} := \text{OR}(k^{\text{Gray}}, \ell^{\text{Gray}})$ contains exactly two bits.

We are only interested in the edge numbers for accessing the precomputed M-scalar products of one tetrahedron. Table 2 presents the available Gray-codes indices for the edges, i.e., the three edges connected to x_3 have the Gray-code indexing 5, 6 and 12. This Gray-code numbering will simplify the rotation of the tetrahedron in Sect. 3.2.3. The six edge numbers are spread between 3 and 12 in the Gray-code. In order to access the precomputed scalar products in the 6×6 matrix from Table 1 we have to transform these Gray-code

edge numbers to the edge index set $\{0, \ldots, 5\}$. This transformation is performed by $\mathbf{f}_{\text{int}}(\mathbf{d}) = \mathbf{d}/2 - 1$ derived from the linear regression function $\mathbf{f}(\mathbf{d})$ such that edge $e_{1,3}$ from x_1 to x_3 has the Gray-code number $d = 5$ with $\mathbf{f}(\mathbf{5}) == 1$ and so we will store the scalar products with this edge in row/column 1 of Table 1.

3.2.2 Accessing the precomputed scalar products via Gray indexing

We have to find out which scalar products are needed for solving for a certain vertex. In our reference case we are solving again for vertex 4. We observe from matrix M' in (10) that vertex 3 (marked in red) is the vertex that is part of all the possible scalar products needed to solve for x_4. It is not only part of all possible scalar products but it is also contained in both edges of each product.

$$M' = \begin{cases} [e_{1,3}^T M e_{1,3},\ e_{2,3}^T M e_{1,3},\ e_{3,4}^T M e_{1,3}]^T \\ [e_{1,3}^T M e_{2,3},\ e_{2,3}^T M e_{2,3},\ e_{3,4}^T M e_{2,3}]^T \\ [e_{1,3}^T M e_{3,4},\ e_{2,3}^T M e_{3,4},\ e_{3,4}^T M e_{3,4}]^T. \end{cases} \tag{10}$$

Therefore we call vertex 3 the common vertex. Based on this common vertex we can extract M' from the 6×6 matrix T_M depicted in Table 1 by the following general procedure: The reference case for unknown x_4 calls Algorithm 1 with common vertex 3 and returns those precomputed entries in Table 1 marked in blue. The procedure in Algorithm 1 can be applied similarly with the remaining tetrahedral vertices. The challenge of applying the procedure in the general case without code branching and lookup tables will be addresses in the next section.

Algorithm 1 Accessing M' wrt. common vertex j

```
1: procedure COMMON2MPRIME(j, M')
2:     Get Gray-code edge numbers d_0(j), d_1(j), d_2(j)
3:     Get edge indices k_i := f_int(d_i)      i = 0, 1, 2
4:     M' := T_M(k, k)                        ▷ extract 3 × 3 matrix
5: end procedure
```

3.2.3 Rotating elements into the reference configuration

The previous section explained the computation of the wave front with the unknown quantity in ϕ_4. If the wave front hits a tetrahedron differently then we have to determine the needed edge indices based on the index $k \in \{1, 2, 3, 4\}$ of an unknown quantity ϕ_k.

Thanks to the Gray-code indexing, we can perform the needed edge index transformation from the reference configuration simply by bit shift operations and the same holds for the sign changes caused by redirected edges, see "Appendix A" for details. The appropriate edge indices after applying $\mathbf{f}_{\text{int}}(\mathbf{d})$ are presented in row 2 of Table 3 for all configurations.

Table 3 Edge indices and signs after rotation

ϕ_k	ϕ_4	ϕ_3	ϕ_2	ϕ_1
Edges	5,1,2	2,4,0	0,1,3	3,4,5
Signs	+,+,+	+,–,+	+,–,–	–,+,+

3.3 Further memory footprint reduction

The possibility to reduce the memory footprint from 18 to 6 floats originates from the fact that $\langle \mathbf{e}_{k,s}, \mathbf{e}_{s,\ell} \rangle_{M^\tau}$ $(k \neq \ell)$ represents an angle of a surface triangle whereas $\langle \mathbf{e}_{k,s}, \mathbf{e}_{k,s} \rangle_{M^\tau}$ represents the length of an edge in the M-metric which are also the products in the main diagonal. Basic geometry as well as vector arithmetics yield to the conclusion that the angle information can be expressed by the combination of three edge lengths. Therefore we only have to precompute the 6 edge lengths of one tetrahedron and compute on-the-fly only the 3 needed angle data therein instead of the potential 12 angle data. This finally reduces the memory footprint per tetrahedron to 6 numbers.

The determination of the involved edges and the sign changes are presented in detail in "Appendix B". We finally achieve the generalized formula

$$e_k^T M e_l = esgn * \frac{1}{2} \left(e_k^T M e_k + e_l^T M e_l - e_s^T M e_s \right) \tag{11}$$

with $s^{\text{Gray}} = \text{XOR}(k^{\text{Gray}}, \ell^{\text{Gray}})$ and $k^{\text{Gray}} \neq \ell^{\text{Gray}}$ are the Gray-code edge numbers from Table 2. The sign has to be calculated only for the off-diagonal elements in Table 1 as follows:

$$\begin{aligned} esgn = ((2 * (s^{\text{Gray}} > k^{\text{Gray}}) - 1) \\ * (2 * (s^{\text{Gray}} > \ell^{\text{Gray}}) - 1)). \end{aligned} \tag{12}$$

Finally, we have to store only the diagonal part $e_k^T M e_k$ as in Table 4 and all mixed elements are computed via formula (11). The mapping system remains the same. You only address 3 diagonal products out of 6 now and compute the 3 mixed products needed. In the reference case when we are computing for x_4 the resulting edge numbers from the mapping system are (5, 2, 1). The reduced memory footprint approach has to access now only the three products referenced from edge numbers (1, 1), (2, 2) and (5, 5). These are the products marked in blue in Table 4. The other 3 needed products for the mixed edge numbers, respectively (1, 2), (1, 5) and (2, 5) which reference exactly the off-diagonal products in blue in Table 1, are calculated based on formula (11).

Table 4 Reduced number of scalar products

0	1	2	3	4	5
$e_{1,2}^T Me_{1,2}$	$e_{1,3}^T Me_{1,3}$	$e_{2,3}^T Me_{2,3}$	$e_{1,4}^T Me_{1,4}$	$e_{2,4}^T Me_{2,4}$	$e_{3,4}^T Me_{3,4}$

The memory footprint reduction is obvious. Instead of precomputing and storing all the 18 scalar products in Table 1, only the 6 scalar products of the main diagonal are precomputed and stored as in Table 4.

4 Global solution algorithms

Our numerical solution of the Eikonal equation is based on local solvers: After some initialization of ϕ with boundary values and with $+\infty$ in the domain, we run the local solver on every tetrahedron. The simple Algorithm 2 is constructed to execute the local solvers on all tetrahedra until no values ϕ located at the tetrahedron vertices change any more.

Algorithm 2 Simple Eikonal Scheme

```
procedure INITIALIZE(Φ)
    for all vertices x do
        if x is source then
            Φ_x = Source(x)
        else
            Φ_x = ∞
        end if
    end for
end procedure

procedure ITERATION(Φ)
    repeat
        for all tetrahedra t do
            for all vertices x of t do
                Φ_x = min(Φ_x, Solver(t, Φ))
            end for
        end for
    until no changes in Φ
end procedure
```

More efficient solution algorithms will cycle over the tetrahedra in some specific order: Fast marching algorithms on structured grids track the solution from known parts into unknown parts of the computational domain [11].

This algorithm can be generalized to unstructured grids [12] wherein vertex updates are tracked via a dynamic data structure. Once a value ϕ_i is changed, one has to check only all tetrahedra with vertex i, see Algorithm 3. The vertices in the set of this tracking wave front set are checked in some specific order. It pays off to take into account the direction of the wave, which results in different heuristics. Alternatively, all vertices of an active set are checked and a new active set is formed. This can be described as an iterative process leading to fast sweeping algorithms, see [1,4,5] and Algorithm 4.

Algorithm 3 Active Set Eikonal Scheme

```
procedure ITERATION(Φ)
    L = { boundary vertices }
    while L not empty do
        remove vertex x from L
        for all tetrahedra t containing x do
            U_x = min(U_x, Solver(t, U))
        end for
        if Φ_x changed then
            add neighbor vertices of x to L
        end if
    end while
end procedure
```

Algorithm 4 Sweeping Eikonal Scheme

```
procedure ITERATION(Φ)
    L = { boundary vertices }
    while L not empty do // outer iteration
        M = {}, create new set
        for all vertices x in L do
            for all tetrahedra t containing x do
                U_x = min(U_x, Solver(t, U))
            end for
            if Φ_x changed then
                add neighbor vertices of x to M
            end if
        end for
        L = M
    end while
end procedure
```

The proposed algorithm uses a modification of the active set update scheme combined with the local solver described above designed for unstructured tetrahedral meshes with inhomogeneous anisotropic speed functions.

Another opportunity for solving the Eikonal equation would be the Hopf-Lax update which is explained in [13] for a 2D finite element discretization. A mathematically detailed representation for the Eikonal equation using finite elements with the Hopf-Lax update is given in [13] which has been extended to 3D in [14].

5 Task based parallel Eikonal solver

There are several strategies to derive a parallel Eikonal solver [3,15–17]: The simple Algorithm 2 can execute the "for all" loop in parallel [18]. Each thread or processor i is responsible for a number of tetrahedra, e.g. forming a sub-domain Ω_i of the computational domain Ω. However, only a few tetrahedra have to update the solution ϕ at the same time. So this code is not efficient.

5.1 Multithreading

The marching and sweeping versions of the Eikonal solver restrict updates to a moving wave front which covers just a fraction of the domain. In Algorithm 3 the "for all" loop over the elements of the active set can be run in parallel. The efficiency of the resulting code depends on the way this loop can be executed. In a shared memory computing system with task based parallelism, this loop can be partitioned dynamically, as described in Algorithm 5.

Algorithm 5 Task based parallelism example

1: partition L dynamically into sub-sets L_i
2: launch kernel on GPU processors or start threads on thread i
3: **for all** vertices x in L_i **do**
4: $\quad \Phi_x = min(\Phi_x, Solver(t, \Phi))$
5: **end for**
6: wait for threads or processors to terminate

Using many-core processors and many threads, programming paradigms like OpenMP allow this kind of parallelism. The efficiency depends on the size of the active set, which corresponds to the length of the wave front which changes during computation. Starting with a single excitation point, the Eikonal solver will start a wave with growing wave front size, until the wave reaches the boundary and is cut off, see Fig. 2. Furthermore, the amount of work per set element and the general memory layout play an important role.

Therefore, the multi threaded version of the algorithm is derived by just partitioning the active list for each iteration and assigning the work of each sublist to one thread. Each thread is updating its own active sublist but the solution is synchronized against the solution vector Φ where all the values for each node are kept. In practice, we simply divide the active list arbitrarily into N sublists, assign the sublists to the N threads. We choose N to be the number of virtual CPU cores in multi-threading.

We have a very short convergence time of the algorithm and good quality results, see Fig. 2 wherein the wave propagation looks very smooth.

5.2 CUDA offloading kernels

A GPU version of the Eikonal solver can be derived in the same way. An accelerator program, e.g. in CUDA for Nvidia GPUs is executed on the host. Whenever a large parallel loop occurs, a compute kernel on the GPU is launched including some data copy to and from the GPU. Host code offloads the computational tasks and continues, when all GPU processors have successfully executed the compute kernel code. This way the "for all" loop in Algorithm 3 is split into a number of kernels, which have to be executed successively. Efficiency

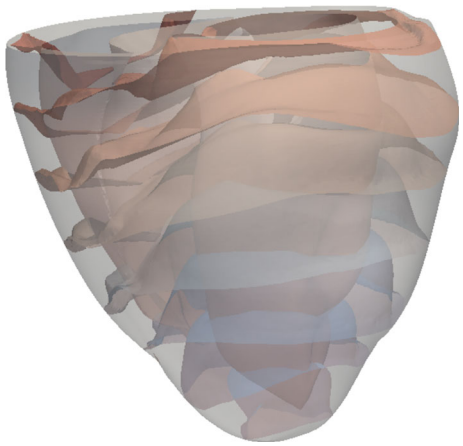

Fig. 2 Arrival time $\varphi(x)$ ranging from 0 (bottom) to 1 (top)

depends on the size of the active set, the amount of work per element and memory considerations.

When designing an algorithm that will be used in a streaming unit such as a GPU, it is very important to optimize for throughput and not for latency. Together with the concept of coalesced memory access pattern we designed our algorithm based on one critical point: allowing data redundancy in order to achieve good coalesced memory access, throughput and occupancy.

The main idea of the Eikonal update scheme is to find the solution of one node calculating for all the one-ring tetrahedra. It means that in the general case one thread is supposed to solve for one node by doing the computations and solving for all its neighboring tetrahedra. Unfortunately, we would have non–coalesced access pattern, smaller number of threads (low level of parallelism), low throughput and a poor occupancy. A better solution consists in calculating a priori all the neighboring tetrahedra indices for each node in the active list, store them in a global array and then assign each element (tetrahedra) to one thread. Of course the information will be redundant as shown in Fig. 4 on the last array, but we trade in memory to gain in performance. In this way we achieve a better coalesced memory access, more threads to run (increased parallelism) and a better bandwidth which are the three most important concepts for a fast GPU algorithm.

5.2.1 Using SCAN for compaction

In order to successfully apply this idea we use the parallel SUM SCAN algorithm and its most important applications such as stream compaction and address generation for efficiently gathering the results into memory as presented in [19], see Algorithm 6.

Because we do not use any specific parallel data structure for managing the active list in our CUDA implementation we

Algorithm 6 Parallel scan algorithm on 2^d threads

```
1: procedure SCAN(x[]) on thread number p
2:     for i=0..d-1 do
3:         if p ≥ 2^i then
4:             x[p] := x[p - 2^i] + x[p]
5:         end if
6:         synchronize
7:     end for
8: end procedure
```

use a standard C vector with ones and zeros in order to distinguish between nodes in the active list and nodes not in the active list as you can notice on the INITIALIZEACTIVELIST procedure in Algorithm 7 lines 8 and 12. This list can be easily managed by atomic update operations which allow to add, to remove and also to update the active list. Unfortunately, this approach results in branched code because we solve only for the active elements and the threads processing non–active (zero) elements are going to remain idle resulting in a poor performance. We use compaction in order to avoid this branched code.

Algorithm 7 Eikonal CUDA Initialization

```
1: procedure INITIALIZEACTIVELIST(X,Φ,L)
2:     for each x in X do
3:         if x is source then
4:             Φ_x = Solver(x)
5:         else
6:             Φ_x = ∞
7:         end if
8:         L_x = 0
9:     end for
10:    for each x in Source do
11:        x_Nbh = Find_Neighbors_Of(x);
12:        L_{x_Nbh} = 1; // add neighbors of x to L
13:    end for
14:    ActiveNodes = Count_Active_Nodes(L);
15: end procedure
```

Stream compaction requires two steps, a scan and a gather as shown in Fig. 3. By using scan on the predicate array we build another array of accumulated sum excluding the last element (Algorithm 8, line 2). That is why it is called exclusive SUM SCAN. The result values are the generated gather addresses. The gray elements are the exact addresses which will be used for the gather step. Based on these addresses the active elements are gathered to the compacted (last) array of Fig. 3. Compaction is useful when there is a large number of elements to compact and the computation on each remaining element is expensive. This is exactly our case and it happens to be one of the kernels with a very good performance in our implementation.

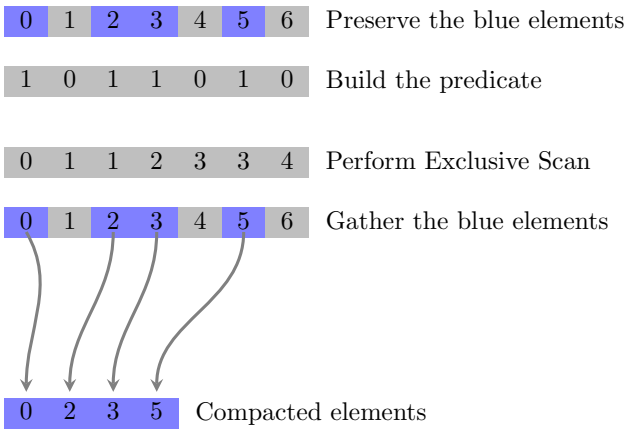

Fig. 3 Address generation and compaction process

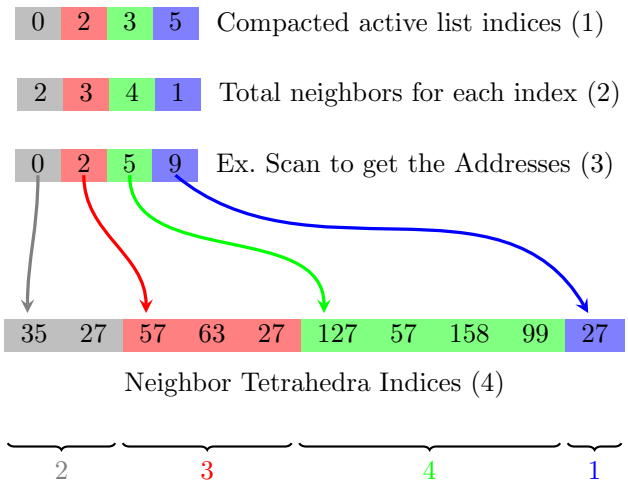

Fig. 4 Data management for achieving a high computational density on the GPU

5.2.2 Using SCAN for data management

The first array in Fig. 4 denotes the compacted active list generated by using the process from Fig. 3. Based on the compacted active list, we compute the total number of neighboring tetrahedra for each node of the active list (Algorithm 8, line 4) and store them to a temporary array as in step 2 in Fig. 4. Then the scan algorithm is applied to this temporary array to generate the addresses which will be used to gather the neighboring tetrahedra indices to the last result array (Algorithm 8, line 5–7). For example, from the Fig. 4 the first two elements are copied starting at address 0, the next 3 elements starting at address 2 and so on. The information on the last array is redundant. This scheme allows us to achieve coalesced access and also to increase the parallelism by increasing the working number of threads. One thread per element is used for solving the elements into the last array as in Algorithm 8, line 8. The algorithm uses the same scheme

Algorithm 8 Eikonal CUDA Update Scheme

procedure UPDATEPOINTSINL(X,Φ,L)
  while ActiveNodes > 0 **do**
    $sAddr = Exclusive\_SCAN(\mathrm{L})$;
    // calculate gather address from L.
    $cList = Compact\_Active\_List(sAddr, L)$;
    $nbhNr = Count\_Nbhs(cList)$;
    // count the total number of neighboring tetrahedra for each x in L
    $sAd = Exclusive\_SCAN(nbhNr)$;
    // calculate addresses from $nbhNr$
    $elemList = Gather\_elements(sAd, cList)$;
    // find all the neighboring tetrahedra for each x in L
    // and store them based on the address generated in previous line
    for each $elem$ in $elemList$ **do**
      $p = \Phi_x$;
      $q = Solver(x, elem)$;
      // where x is the reference vertex of the element $elem$ (the vertex to be solved)
      $predicate_x = (|p - q| < d_\epsilon * (1 + (|p| + |q|)/2))$;
    end for
    $s = Exclusive\_SCAN(predicate)$;
    $conList = Compact\_Predicate(s, predicate)$;
    // generate the converged active list.
    $nbhNr = Count\_Nbhs(conList)$;
    // count the total number of neighboring points for each converged x in $conList$
    $L_x = 0$; // Remove x from L
    $sAd = Exclusive\_SCAN(nbhNr)$;
    // calculate addresses from $nbhNr$
    $ptList = Gather\_elements(sAd, conList)$;
    // find all the neighboring Points for each converged x in L
    // and store them based on the address generated in previous line
    $nbhNr = Count\_Nbhs(ptList)$;
    // count the total number of neighboring tetrahedra for each x in $ptList$
    $s = Exclusive\_SCAN(nbhNr)$;
    // calculate addresses from $nbhNr$
    $elemList = Gather\_elements(s, ptList)$;
    // find all the neighboring tetrahedra for each x in $ptList$
    // and store them based on the address generated in previous line
    for each $elem$ in $elemList$ **do**
      $p = \Phi_{x_{nbh}}$;
      $q = Solver(x_{nbh}, elem)$;
      // where x_{nbh} is the reference neighbor vertex of the converged point x.
      if p > q **then**
        $\Phi_{x_{nbh}} = q$;
        $L_{x_{nbh}} = 1$; // add x_{nbh} to L
      end if
    end for
    $ActiveNodes = Count\_Active\_Nodes(L)$;
  end while
end procedure

later on when it solves for all the neighboring nodes of the converged active list nodes.

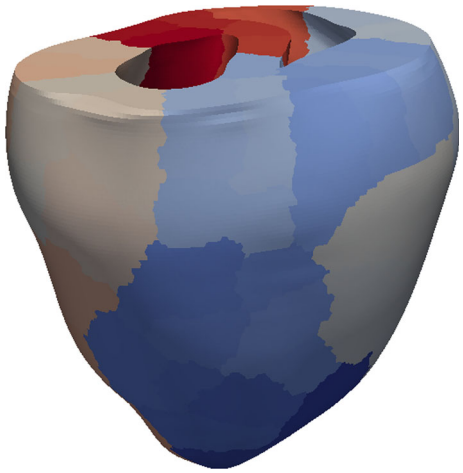

Fig. 5 Domain decomposition. Computational domain Ω and sub-domains Ω_i

6 Domain decomposition parallel Eikonal solver

For large scale problems, the task based parallel model will run into difficulties: There might be not enough (shared) memory on a single host or on a GPU, the computing power of a single compute unit is not sufficient, or the parallel efficiency is not satisfactory. In all cases, a distributed memory model is needed. Hence a coarser decomposition of the algorithm is needed, namely a domain decomposition approach.

The domain Ω is statically partitioned into a number of non-overlapping sub-domains Ω_i, see Fig. 5, each of them is assigned to a single processor. Synchronization and communication of the processors is to be reduced to a minimum. In our case, a single processor i can efficiently solve the Eikonal equation on Ω_i, as long as its boundary data on $\partial\Omega_i$ is correct. However, this data may belong to the outer boundary $\partial\Omega$ or to other processors. Hence inter-processor communication is needed. Algorithm 3 can be adapted in several ways.

6.1 Parallel sweep

The loop over all vertices is restricted to the vertices that belong to the current processor. The new set M is assembled as the union of the local sets. The local Φ values are exchanged at the end of the loop. This implements the communication. The results may differ from the sequential algorithm, since the order of computation is different and the ϕ updates are delayed. This may result in a slightly larger iteration count. The main challenge of a static decomposition is load balancing. Since the wave front moves, the amount of elements per sub-domain varies largely. In the beginning and at the very end of the computation, many sub-domains will have an empty active set. A way to overcome this is a

Algorithm 9 Parallel sweeping Eikonal Scheme

```
1: procedure ITERATION ON PROCESSOR i(Φ_i)
2:     L_i = { boundary vertices on Ω_i }
3:     while at least one L_i not empty do // outer iteration
4:         M_i = {}, create new set
5:         for all vertices x in L do
6:             for all tetrahedra t contain x do
7:                 Φ_x = min(Φ_x, Solver(t, Φ))
8:             end for
9:             if Φ_x changed then
10:                add neighbor vertices of x to M_i
11:            end if
12:        end for
13:        L_i = M_i
14:        synchronize local copies of Φ
15:    end while
16: end procedure
```

different mapping of elements to a processor. Several smaller sub-domains may increase the chance of a processor to have a non-empty active set.

6.2 Local solves

In order to reduce communication even further, we do not update Φ and M in the loop at all. We consider Algorithm 3 as a black box able to solve the local problem. Once we have

Algorithm 10 Parallel Eikonal scheme with local solvers

```
1: procedure ITERATION ON PROCESSOR i(Φ)
2:     repeat
3:         solve in Ω_i
4:         synchronize local copies of Φ
5:     until no changes in Φ
6: end procedure
```

a solution, we have to update the boundary values on $\partial\Omega_i$. In case of any changes, we have to run the local Eikonal solvers on Ω_i again. Now the number of sweeps is slightly larger than in the sequential case. In the case of a GPU this scheme can be mapped to single GPU cores (CUDA, streaming multiprocessors (SM)). A single kernel is able to solve the Eikonal equation on a small sub-domain. This allows for memory optimizations like the use of "shared memory" local to an SM for the storage of Φ, local matrices etc. Furthermore, the amount of host interaction and synchronization is limited.

6.3 CUDA implementation

We use two different strategies to distribute the workload of the kernels to the available blocks in CUDA. Our first approach consist in mapping the work of a sub-domain into different blocks or SMs on the GPU where one thread can process for only one tetrahedron as explained in Sect. 5.2.2. This allows the shared memory of all blocks solving for a certain sub-domain to be used from the local sub-domain solver. It means more data are going to fit into the shared memory since the blocks share the load. The other approach is one block computes one sub-domain where the shared memory is limited to the shared memory of the block but the threads have more work to do increasing in this way the granularity. Every scan and scatter or any other kernel which prepares the data for the solution kernel computes independently for each domain. The only difference between the two versions is in the way the workload of the main kernel is distributed. Since the main difficulty of the static decomposition is the load balancing because the wavefront moves and many domains remain idle during execution, empty active sets, then distributing the load not only in one block but on many blocks increases the number of utilized blocks and multiprocessor units (SM) on the GPU. This has also the benefit that the wave front velocity information can be stored into the shared memory since more shared memory is available now compared to the model one block one sub-domain.

The advantage of the other model, where one block is responsible for one domain is the granularity or the number of elements processed by one thread. In this model one thread has more work to do and with the right type of access it can increase the efficiency especially if that fits into the registers of the thread. Access optimization of this feature is a future work. One idea is to use the CUB block load primitive from CUB library [20] which provides collective data movement methods for loading a linear segment of items from memory into a blocked arrangement across a CUDA thread block.

The other advantage compared to the first approach is the almost complete reduction of host synchronization. This results from the possibility to synchronize within each block for each sub-domain since the computation is independent from each other. In this case we do not need any host synchronization for each kernel as in the first approach where we had to make sure that all the blocks ended execution before continuing with the next kernel. This was also possible by using the dynamic memory allocation in CUDA. The only remaining synchronize is needed to check the termination condition.

6.3.1 First DD approach

In the first domain decomposition approach we use a scheduling strategy to distribute the computations of each active domain to the available blocks. This first approach consists of many kernels performing tasks such as scan, scatter or compaction, i.e, data management as explained in Sect. 5.2 before the kernel with the main load starts execution. All

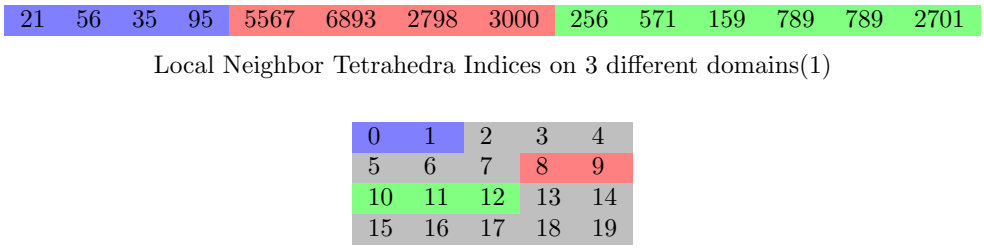

Fig. 6 Blocks allocated by the GPU to solve for the tetrahedrons on 3 active domains assuming one block runs with only 2 threads. (Gray means inactive block.)

these kernels compute independently for each domain. The distribution strategy is only used for the main kernel.

- Work load management strategy.
  During each iteration we keep track of the active domains. The number of active domains changes with thread synchronization. In this way the data preparation by other kernels and solution computations are done only for the active domains reducing in this way unnecessary computations that might come by preparing data or solving for inactive domains.
  The work load of one domain will be distributed into several blocks where each block computes for a subset of the tetrahedron elements of that domain calculated in advance from the local active list as shown in Fig. 6. One thread in a block computes for one tetrahedron element and the number of threads in the block is chosen such that the occupancy is not decreased. The number of blocks solving for one domain is defined based on the total number of tetrahedron elements divided by the number of threads used per one block.
  The algorithm is such that in the beginning of the execution and at the end of the execution the number of active domains is very small. This means that many blocks or processing units remain idle. This approach increases the utilizations because it uses more blocks for one domain reducing in this way the number of inactive processing units (SMs). Another way to overcome this is to have several smaller sub-domains which may increase the chance of a processor to have a non-empty active set.
- Synchronization.
  At the end of the loop local values are exchanged based on local to local precomputed mapping which reduces the access to global memory. The domain Ω is statically decomposed into a number of non-overlapping sub-domains Ω_i, which enables the exchange of the local solution values only on common nodes in the boundary between two domains. In our case the solution of the Eikonal equation on Ω_i is efficient as long as its boundary data on $\partial\Omega_i$ is correct. This data may belong to other sub-domains processed by other blocks which requires inter-process communication. Communication is realized on one GPU between blocks by host synchronization. For this reason the synchronization kernel takes place at the end of the loop. In order to achieve a quick mapping, a precomputed local to local interface of each domain with all the other neighboring domains is used. It contains the local indices of the boundary nodes and the indices of two domains that form that boundary where these nodes are contained. In this way we can easily start a kernel which has all the needed information in order to exchange solution values for all the node in the interface. The exchange happens only if the value of a certain boundary node in the first domain is smaller than the value of the same node in the other domain. If that happens then the larger value is updated with a better solution, the node is added to the local active list of the sub-domain of the changed value, and the sub-domain of the changed value becomes active.
- The termination condition.
  It checks if the total number of nodes for all local active lists is zero which is the fulfillment of the termination condition. This is a separate kernel since it needs to wait for the synchronization kernel to end, where nodes are added to the local active lists, for getting a correct result.

6.3.2 Second DD approach

The second DD approach is based on the local solve in Sect. 6.2 where a single kernel is able to solve the Eikonal equation on a small sub-domain. This allows for host interaction and synchronization reduction and granularity optimizations. One sub-domain is processed within one block in CUDA which limits the optimization of shared memory usage more than in the previous approach. Still the solution vector and local matrices can be stored in the shared memory of one block.

- Work load management strategy.
  We keep track of active domains in order to reduce the unnecessary computations for the inactive domains and allow for better load balancing of the work into the blocks. Since one block is computing for a single sub-domain one thread has do to more work. This might might become a

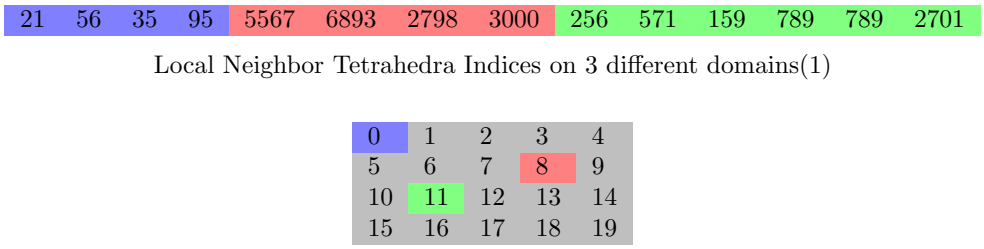

Fig. 7 Blocks allocated by the GPU to solve for the tetrahedrons on 3 active domains assuming one block runs with only 2 threads

very efficient approach if done in the correct way by preserving the coalesced memory access and by increased granularity as in Fig. 7. Several smaller sub-domains increase the chance of a block to have a non-empty active set.

- Host synchronization reduction.
  The domain decomposition in CUDA allows for kernel optimizations and host interaction reduction because the independent computations within a block for each sub-domain allows for block synchronization instead of host synchronization. This reduces the number of kernels and improves the code. The memory allocation is done dynamically which means a total reduction of memory transfer from the device to host. In summary this improves the performance thanks to the domain decomposition approach.
- Synchronization.
  Fusing the number of kernels into one kernel is not straight forward because of incorporating the synchronization kernel and the termination condition kernel. The safest way to guarantee correct results consists in a synchronization with the host in order to ensure that all blocks ended execution which is mandatory for the termination condition kernel. We finally managed to reduce all computations to one large synchronization kernel and a second termination condition kernel.
  The change in the local to local interface model as explained in Sect. 6.3.1 allowed us to incorporate the synchronization kernel. The first domain decomposition approach used a local to local mapping for all boundaries between two sub-domains. This can be no longer applied in the new approach where one blocks solves for one sub-domain because the information in the interface is not grouped for each sub-domain. Now the mapping contains the total information on the boundary for all domains. One block does not know which part of the information belongs to the sub-domain it is solving for. Therefore, we changed the way of pre-computing the interface such that every sub-domain has its own separated interface information with its neighboring sub-domains. As a consequence one block knows exactly which interface maps to which process and updates accordingly.

Table 5 Run times in sec. on the workstation

Implementations	# Tets	CUDA	OpenMP 8 threads
Without Gray-code	3,073,529	1.49	5.66
With Gray-code	3,073,529	0.73	3.65
Without Gray-code	24,400,999	11.48	56.63
With Gray-code	24,400,999	5.16	36.43

This change allowed us to incorporate all computations into one main kernel but it does not solve the synchronization problem. Placing the synchronization kernel within the main kernel it is no longer thread safe. When the synchronization kernel tries to update a boundary solution value of another domain, that block computing on that specific other domain might update it as well. Of course this update is protected with atomic operations but this alone is not enough. Fortunately, the order of this update is not relevant to the overall solution and so we achieve finally a correct solution.

7 Numerical tests and performance analysis

We present the results for the numerical tests in single precision performed on a workstation with Intel Core i7-4700MQ CPU @2.40GHz processor and GeForce GTX 1080 GPU (Nvidia Pascal). We use a coarser mesh of a human heart with 3,073,529 tetrahedra and 547,680 vertices and a finer mesh of the same heart which contains 24,400,999 tetrahedra and 4,380,375 vertices. Results and analysis will be shown for the Gray-code and the domain decomposition method which includes the Gray-code improvements.

7.1 Gray code

In Table 5 we compare between our new implementation including the Gray-code method and our old implementation without that. We achieved an acceleration by 35% for the OpenMP implementation and by 50% in the CUDA implementation by using the local Gray-code numbering of edges to reduce the memory footprint.

A profiling of the CPU performance (Intel's Vtune Amplifier) shows that the number of loads has been reduced to 70% and the number of stores dropped to 40% in the Gray-code version. The last level cache (LLC) miss count has been reduced to 84%. The significant reduction in stores is caused by avoiding many local temporary variables from the original approach. The reduction of non-coalesced memory accesses is documented by the reduced number of loads and the lower LLC miss counts.

We see these improvements even more clearly with the CUDA implementation wherein we get an even larger performance improvement caused by avoiding expensive non-coalesced memory accesses. The reduced number of non-coalesced memory accesses to global memory is the same as on CPU but the GPU profits more from that. We compared the profiling results of both CUDA versions (Nvidia's Visual Profiler nvvp). The old implementation had a high local memory overhead which accounted for 54% of total memory traffic which indicated excessive register spilling. After implementing the Gray-code method that problem disappeared and the number of register spills decreased significantly. We have only 60 bytes spill stores and 84 bytes spill loads in the new main kernel in contrast to the old kernel with 352 bytes spill stores and 472 bytes spill loads. The L2 cache utilization is maximized in the new kernel. The L2-bandwidth is doubled and achieves now 952.08 GB/s, compared to the version with only 456.103 GB/s. The number of loads and stores is also decreased for the L2 cache as well as for the device memory. For this reason the local memory overhead is also decreased and the achieved device memory bandwidth is increased to above 90% compared to the old version where the achieved memory bandwidth was below 60% indicating latency issues. And lastly the divergent execution also dropped and improved the warp execution efficiency from 75 to 80%.

We tested our code also on Intel Phi (KNL) server with 64 physical cores, 1.3GHz for both meshes. The OpenMP implementation scales almost linearly until 64 threads, see Fig. 8. Then the performance to drops because of the hyper-threading. We achieve similar scaling results for the larger mesh for which the algorithm converges in 5.6 s using 256 threads. This is a comparable result with the CUDA implementation where we achieved a convergence time of 5.16 s.

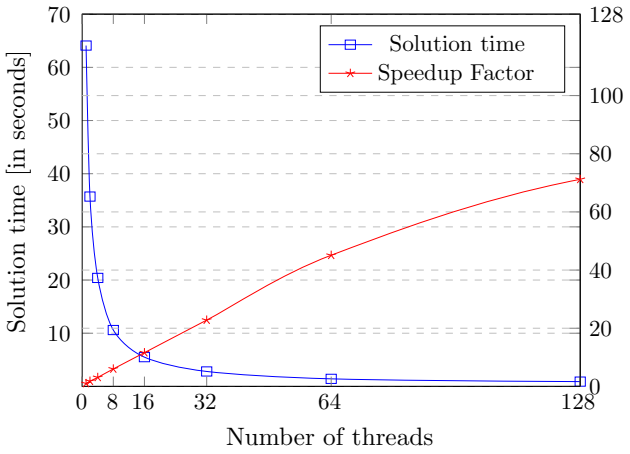

Fig. 8 Scaling results on Intel Xeon Phi 1.3GHz (KNL) for the coarser mesh

Table 6 Run times in sec. on the GTX 1080 for the coarser mesh

# Sub-domains	First DD approach	Second DD approach
74	0.48	0.69
160	0.52	0.60
320	0.58	0.51

7.2 Domain decomposition

Let us compare the numerical results between the first and the second DD approaches, and with the non-DD approach. We focus especially on hardware limitations observed on the GTX 1080 GPU and how they could be overcome.

Table 6 compares the run times between both DD versions for the coarser mesh with $3 \cdot 10^6$ tetrahedra. There are no global memory limitations for running the smaller example on the GTX 1080 with 8GB global memory and 4KB shared memory per block. All data fit into the global memory by preallocation. Therefore, we do not have to reallocate GPU-memory in each iteration as in the second DD approach. We observe that the second DD approach scales better with the increased number of sub-domains until we get 0.51 s convergence time for 320 sub-domains. It is already faster than the version without DD where the best time we achieved is 0.73 for the coarser mesh as shown in Table 5. The reason consists in the DD versions use the block scan implemented by the CUB library instead of the device scan primitive as in [21]. Besides the block scan we use block load and block store from the CUB library [20] for loading a linear segment of items from memory into a blocked arrangement across a CUDA thread block. The efficiency is increased by the increased granularity $ITEMS\_PER\_THREAD$. Performance is also increased until the additional register pressure or shared memory allocation size causes SM occupancy to fall too low.

The block scan computes a parallel prefix sum/scan of items partitioned across a block. If we decompose the domain in sub-domains small enough such that block scan can use the available shared memory and the register pressure does not affect the occupancy then this method fits to our DD approaches. Additionally, the block scan can be called within a CUDA kernel and so we incorporated it into the one big kernel for the second DD approach. A device scan would require a separate kernel.

Table 7 Run times in s on the GTX 1080 for the finer mesh

# Sub-domains	First DD approach	Second DD approach
2700	5.96	6.89
3000	6.40	7.55
4000	7.55	9.46
8000	14.00	14.74

The drawback consists in the shared memory limitation. The block scan requires shared memory and the shared memory size limits the number of sub-domains. A small number of sub-domains means larger sub-domains for the same mesh and for this reason the data to be scanned for those sub-domains do not fit anymore into the shared memory. Hence we start the testing with 2700 sub-domains for the larger mesh and 74 sub-domains for the coarser mesh. Anyway the idea is always to increase the number of sub-domains as discussed in Sect. 6.3.1 since it improves the load balancing and therefore this limitation is not relevant for our code.

The shared memory usage together with the consecutive items per thread used by the block scan allows us to conclude that the block scan is the reason that the DD approaches are faster. We would like to emphasize that using the block scan is only possible because of the DD approach. One can notice from the convergence results for the finer mesh in Table 7 that the second DD approach is not scaling any more. This is caused by the limited global memory of a single GPU such as GTX 1080 in our case. While the global memory was large enough in the coarser example to allow the preallocation of the memory space needed for each sub-domain during the algorithm execution, it didn't work out anymore for the larger example. As a consequence, the coarser example scales with the increased number of sub-domains as expected. The GPU cannot provide enough memory to preallocate the needed space for each sub-domain in the larger example. In order to get satisfactory results we still preallocate the needed space for each sub-domain but now we do it for each iteration and we free that memory once the blocks terminates. This is done only for the active sub-domains and no reallocation happens during the kernel execution. In this way the global memory suffices and we managed to run the DD approach within a single GPU. The DD version now is just 1 s slower than the version without DD. Now we compare results of Table 7 with the results shown in Table 5 for the CUDA implementation with Gray-code. With the increased number of sub-domains the dynamic allocation is also increased becoming in this way the limiting factor of the scalability. Without these limitation the algorithm would scale very well as it does for the coarser example and of course it would be faster than the non-DD version.

In order to check the scalability of our DD approaches on different GPUs we tested on an Nvidia Titan X Pascal card with 24 multiprocessors (SMs), 4 more than a GTX 1080 card and 4GB more global memory but still not enough to preallocate. The results we get for the mesh using 2700 sub-domains are approximately 20% faster than the results on a GTX 1080 for the first approach and 13% for the second approach. Respectively for the first DD approach we get a convergence time of 4.68 s and for the second approach we get a convergence time of 5.96 s. It scales worse for the second approach because of scalability issue we have on the DD approaches for the larger mesh but that affects more the second approach as explained above.

8 Conclusions and future work

The Gray-code numbering in Sect. 3.3 has significantly reduced the overall memory footprint of the Eikonal solver achieving performance improvements of 35–50%. The analysis showed that this Gray-code version decreased the non-coalesced access level and significantly increased the computational density on the GPU.

The domain decomposition approach solves the Eikonal equation on large scale problems. We achieved to run the domain decomposition approach in one GPU as explained in Sect. 6.3 by using two different strategies in CUDA. The first strategy makes better use of shared memory. For coarser meshes we get a very good convergence time. However, it does not scale well with the increased number of sub-domains since its implementation contains many kernels successively resulting in many host synchronization and memory transfers between the device and the host. Assigning one block to one sub-domain in the second strategy avoids host synchronization and memory transfers nearly completely. This can be seen by the good scalability thanks to dynamic preallocation of global memory for the coarser mesh. We still run into global memory limitation for large scale problems.

The domain decomposition approach is the first step towards the inter-process communication implementation where the limitation of the global memory will be overcome completely by using multiple accelerator cards and cluster computing. As a future work, it will allow the preallocation of global memory which will enable the scalability on large scale problems.

By testing on different GPUs with Pascal architecture such as GTX 1080 and Titan X we concluded that our CUDA implementations, the domain decomposition approaches and the non domain decomposition approach, all scale very well on different Nvidia GPUs. Our OpenMP implementation scales almost linearly on physical KNL cores.

Acknowledgements Open access funding provided by University of Graz.

Appendix A: Indices and signs of edges after rotation

Thanks to the Gray-code indexing, we can perform the edge index transformation needed from the reference configuration in Fig. 1 on page 3 simply by bit shift operations. The appropriate edge indices after applying $\mathbf{f}_{\text{int}}(\mathbf{d})$ are assembled in column 2 of Table 8 for all configurations.

Not only the edge indices change in different configurations but also the signs of the inner products might change. We start by the reference configuration in Fig. 9 (left) with x_3 as the common vertex when solving for vertex x_4. In this case, the edge indices (5, 1, 2) are returned from the mapping system to access the appropriate inner products (marked in blue) from Table 1. These edges are highlighted in blue and bold in Fig. 9 (left). Clearly, no sign change happens in this basis configuration. Looking for the solution in x_3 requires one rotation from the reference case. The common vertex x_2 yields the edge numbers of interest as (2, 4, 0) marked in blue and bold in Fig. 9 (right). Notice that edge $e_{4,2}$ changed its direction with respect to the directions in the reference configuration in Fig. 9 (left) and so one negative sign has to be considered for this edge. Therefore, all precomputed scalar products that contain edge $e_{4,2}$ are going to be multiplied by the negative sign in the computation of ϕ_3. This negative sign is indicated in column 3 of Table 3 as well as in the red bits in row 2. The remaining two cases $k \in \{1, 2\}$ have similar sign changes for certain edges.

The implementation of the four cases avoids branched code, unnecessary memory allocations and costly memory accesses. The bit shift/rotation in Table 8 is realized in the following way. The 2×4 bits per edge for the reference configuration k_0 0000 k_1 0000 k_2 0000 are doubled for each edge, e.g., k_1 k_1 00000000. Afterwards the appropriate bit shift results in the new Gray-code index of that edge represented by the four blue bits and in the four red carry bits. An odd number of carry bits indicates a sign change of this edge.

Table 8 Rotation of elements into the reference configuration

ϕ_k	Edges	Sign	Gray indices of edges and their signs			Shift
ϕ_4	5,1,2	+,+,+	1100 0000	0101 0000	0110 0000	
ϕ_3	2,4,0	+,−,+	0110 0000	1010 1000	0011 0000	$\gg 1$
ϕ_2	0,1,3	+,−,−	0011 0000	0101 0100	1001 1000	$\gg 2$
ϕ_1	3,4,5	−,+,+	1001 1000	1010 1010	1100 1100	$\gg 3$

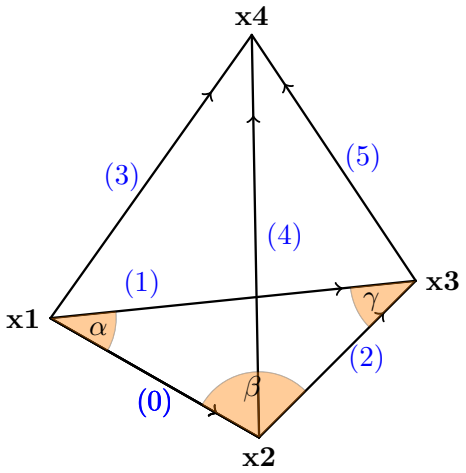

Fig. 10 Angles in a tetrahedron. Edge indices based on Table 2

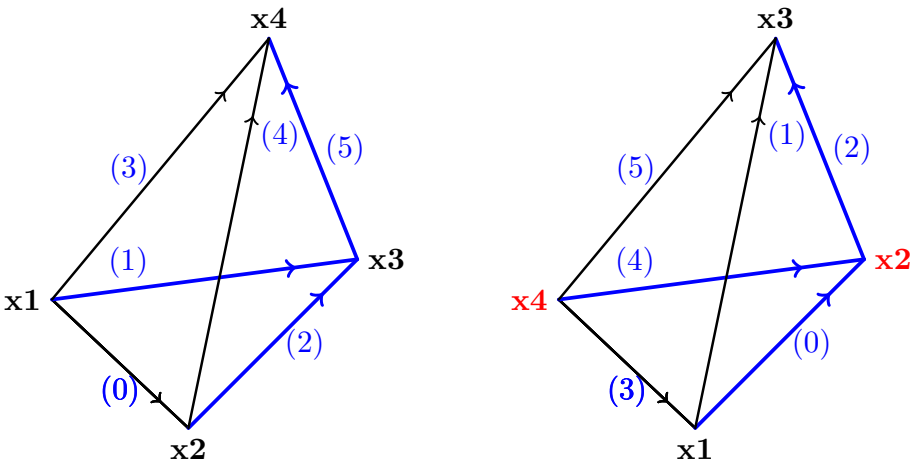

Fig. 9 Reference tetrahedron on the left. Rotation by one position on the right. Edges are named from 0 to 5 based on the edge numbers from Table 2

Appendix B: Derivation of the formula for the mixed inner products

There are actually three different cases in angle computations as showed in Fig. 10 for angles α, β and γ. In this paper the term angle always represents the unnormalized inner product containing the $\cos(\cdot)$ of this angle. It suffices to show what happens to the calculation of these three angles to get to the general formulation.

We start calculating $\alpha = e_{1,2}^T M e_{1,3}$ by reformatting $e_{2,3}$ as:

$$e_{2,3} = x_3 - x_2 = x_3 - x_1 + x_1 - x_2 = e_{1,3} - e_{1,2}$$

and get

$$\begin{aligned} e_{2,3}^T M e_{2,3} &= (e_{1,3} - e_{1,2})^T M (e_{1,3} - e_{1,2}) \\ &= e_{1,3}^T M e_{1,3} + e_{1,2}^T M e_{1,2} - 2 e_{1,2}^T M e_{1,3} \end{aligned} \tag{13}$$

such that angle α can be expressed as:

$$e_{1,2}^T M e_{1,3} = \tfrac{1}{2} \left(e_{1,3}^T M e_{1,3} + e_{1,2}^T M e_{1,2} - e_{2,3}^T M e_{2,3} \right). \tag{14}$$

The scalar product $e_{1,2}^T M e_{1,3}$ from equation (14) contains Gray-code edges $k = 3$ and $\ell = 5$ that are mapped via f(d) to numbers 0 and 1 (see Table 2) such that the scalar product can be found in Table 1 at position (0,1). In this case the direction of all edges $e_{i,j}$ in equation (14) did not change with respect to the reference directions in the tetrahedron in Fig. 9 (left). This is related to the sign issue inherited from the fact that we compute all the possible scalar products starting from a reference edge directions. The reference direction is due to the precomputations of all edges needed as explained in Sect. 3.2.3.

During the angle computations the directions might change with respect the reference edge directions. This will affect the sign of the resulting formula as shown below. We have to express each edge $e_{m,n}$ participating in the angle calculation based on the reference edge direction. In order to calculate $\beta = e_{1,2}^T M e_{2,3}$ we need edge $e_{1,3} = e_{2,3} - e_{2,1}$ that contains $e_{2,1}$. This edge is not directed in the reference edge direction which would be $e_{1,2}$, see the directions in Fig. 9 (right). Therefore we reformulate $e_{1,3}$ as:

$$\begin{aligned} e_{1,3} = x_3 - x_1 = x_3 - x_2 + x_2 - x_1 = x_3 - x_2 \\ -(x_1 - x_2) = e_{2,3} - e_{2,1} = e_{2,3} + e_{1,2} \end{aligned}$$

an get

$$\begin{aligned} e_{1,3}^T M e_{1,3} &= (e_{2,3} + e_{1,2})^T M (e_{2,3} + e_{1,2}) \\ &= e_{2,3}^T M e_{2,3} + e_{1,2}^T M e_{1,2} + 2 e_{1,2}^T M e_{2,3}. \end{aligned} \tag{15}$$

As a result angle β can be expressed as:

$$e_{1,2}^T M e_{2,3} = - \tfrac{1}{2} \left(e_{2,3}^T M e_{2,3} + e_{1,2}^T M e_{1,2} - e_{1,3}^T M e_{1,3} \right). \tag{16}$$

Because we have only one **sign change** for the edges only one minus sign is carried to equation (16). The scalar product $e_{1,2}^T M e_{2,3}$ therein can be found in position (0,2) in Table 1.

Similarly, the calculation of $\gamma = e_{1,3}^T M e_{2,3}$ requires two sign changes in the reformulation of $e_{1,2}$ resulting finally in

$$e_{1,3}^T M e_{2,3} = (-)(-) \tfrac{1}{2} \left(e_{2,3}^T M e_{2,3} + e_{1,3}^T M e_{1,3} - e_{1,2}^T M e_{1,2} \right), \tag{17}$$

see entry (1,2) in Table 1. For all the other angles of the tetrahedron the calculation is done in the same way and the sign problem falls into one of the previous three cases. The generalization of these cases is summarized in equations (11) and (12) on page 7.

References

1. Fu, Z., Kirby, R.M., Whitaker, R.T.: Fast iterative method for solving the Eikonal equation on tetrahedral domains. SIAM J. Sci. Comput. **35**(5), C473–C494 (2013)
2. Ganellari, D., Haase, G.: Fast many-core solvers for the Eikonal equations in cardiovascular simulations. In: 2016 International Conference on High Performance Computing Simulation (HPCS), pp. 278–285. IEEE, Peer-reviewed (2016)
3. Noack, M.: A two-scale method using a list of active sub-domains for a fully parallelized solution of wave equations. J. Comput. Sci. **11**, 91–101 (2015)
4. Fu, Z., Jeong, W.-K., Pan, Y., Kirby, R.M., Whitaker, R.T.: A fast iterative method for solving the Eikonal equation on triangulated surfaces. SIAM J. Sci. Comput. **33**, 2468–2488 (2011)
5. Jeong, W.-K., Whitaker, R.T.: A fast iterative method for Eikonal equations. SIAM J. Sci. Comput. **30**, 2512–2534 (2008)
6. Neic, A., Campos, F.O., Prassl, A.J., Niederer, S.A., Bishop, M.J., Vigmond, E.J., Plank, G.: A fast iterative method for Eikonal equations. J. Comput. Phys. **346**, 191–211 (2017)
7. NVIDIA, CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide
8. Qian, J., Zhang, Y.-T., Zhao, H.-K.: Fast sweeping methods for Eikonal equations on triangulated meshes. SIAM J. Numer. Anal. **45**, 83–107 (2007)
9. Holm, D.D.: Geometric Mechanics: Part I: Dynamics and Symmetry, 2nd edn. Imperial College London Press, London (2011)
10. Weisstein, E.: Gray code. http://mathworld.wolfram.com/GrayCode.html, from MathWorld–A Wolfram Web Resource
11. Sethian, A.: A fast marching level set method for monotonically advancing fronts. Proc. Natl. Acad. Sci. USA **93**(4), 1591–1595 (1996)
12. Sethian, A., Vladimirsky, A.: Fast methods for the Eikonal and related Hamilton–Jacobi equations on unstructured meshes. Proc. Natl. Acad. Sci. USA **97**(11), 5699–5703 (2000)
13. Bornemann, F., Rasch, C.: Finite-element discretization of static Hamilton–Jacobi equations based on a local variational princi-

ple. Comput. Vis. Sci. **9**(2), 57–69 (2006). https://doi.org/10.1007/s00791-006-0016-y
14. Stöcker, C., Vey, S., Voigt, A.: AMDiS Adaptive multidimensional simulations: composite finite elements and signed distance functions. WSEAS Trans. Circuits Syst. **4**(3), 111–116 (2005)
15. Zhao, H.K.: Parallel implementations of the fast sweeping method. J. Comput. Math. **25**, 421–429 (2007)
16. Ganellari, D., Haase, G.: Reducing the memory footprint of an Eikonal solver. In: Limet, S., Smari W., Spalazzi, L. (eds.) 2017 International Conference on High Performance Computing Simulation (HPCS), pp. 325–332. IEEE (2017). https://doi.org/10.1109/HPCS.2017.57
17. Detrixhea, M., Giboua, F., Minc, C.: A parallel fast sweeping method for the Eikonal equation. J. Comput. Phys. **237**, 46–55 (2013)
18. Mattson, T.G., Sanders, B.A., Massingill, B.L.: Patterns for Parallel Programming. Addison-Wesley, Boston (2004)
19. Harris, M., Sengupta, S., Owens, J.D.: Parallel prefix sum (scan) with CUDA. In: Nguyen, H. (ed.) GPU Gems 3, pp. 851–876. Addison-Wesley, Boston (2007)
20. Merrill, D.: Cub library. https://nvlabs.github.io/cub, NVIDIA Research (2013)
21. Merrill, D., Garland, M.: Single-pass parallel prefix scan with decoupled look-back. NVIDIA Technical Report NVR-2016-002 (2016)