SpezialForschungsBereich F 32

Karl–Franzens Universität Graz
Technische Universität Graz
Medizinische Universität Graz

Reducing the memory footprint of an Eikonal solver

D. Ganellari G. Haase

SFB-Report No. 2017-003 February 2017

A–8010 GRAZ, HEINRICHSTRASSE 36, AUSTRIA

Supported by the
Austrian Science Fund (FWF)

Der Wissenschaftsfonds.

SFB sponsors:

- **Austrian Science Fund** (FWF)
- **University of Graz**
- **Graz University of Technology**
- **Medical University of Graz**
- **Government of Styria**
- **City of Graz**

Reducing the memory footprint of an Eikonal solver

Daniel Ganellari
Karl Franzens University of Graz
Institute for Mathematics and Scientific Computing
Graz, Austria
Email: daniel.ganellari@uni-graz.at

Gundolf Haase
Karl Franzens University of Graz
Institute for Mathematics and Scientific Computing
Graz, Austria
Email: gundolf.haase@uni-graz.at

***Abstract*—The numerical solution of the Eikonal equation follows the fast iterative method [1] with its application for tetrahedral meshes [2]. Therein the main operations in each discretization element τ contain various inner products in the M-metric as $\langle \vec{e}_{k,s}, \vec{e}_{s,\ell} \rangle_{M^\tau} \equiv \vec{e}_{k,s}^T \cdot M^\tau \cdot \vec{e}_{s,\ell}$ with $\vec{e}_{s,\ell}$ as connecting edge between vertices s and ℓ in element τ. While the authors of [2] pass all coordinates of the tetrahedron together with the 6 entries of M^τ we precompute these inner products and use only them in the wave front computation. This first change requires less memory transfers for each tetrahedron.**

The second change is caused by the fact that $\langle \vec{e}_{k,s}, \vec{e}_{s,\ell} \rangle_{M^\tau}$ ($k \neq \ell$) represents an angle of a surface triangle whereas $\langle \vec{e}_{k,s}, \vec{e}_{k,s} \rangle_{M^\tau}$ represents the length of an edge in the M-metric. 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 a tetrahedron and compute the remaining 12 angle data on-the-fly which reduces the memory footprint per tetrahedron to 6 numbers.

The efficient implementation of the two changes requires a local Gray-code numbering of edges in the tetrahedron and a bunch of bit shifts to assign the appropriate data. First numerical experiments on CPUs show that the reduced memory footprint approach is faster than the original implementation. Detailed investigations as well as a CUDA implementation are ongoing work.

***Keywords*—Eikonal equation, tetrahedral mesh, Gray-code, reduced memory footprint, parallel algorithm, shared memory multiple-processor computer system, CUDA, GPU**

I. INTRODUCTION

Simulation of one heart beat which faithfully account for biophysical details involved in cardiac electro-physiology and mechanics are still far away from real time performance, even when employing several thousands of compute nodes. Therefore, a simpler model based on the Eikonal equation is considered. This model could be of great utility as a tool for generating activation and repolarisation sequences and its concomitant electrocardiogram by replacing the bi-domain equations with the Eikonal equation.

The basis equations in cardiac electrophysiology are the bidomain equations describing the intercellular and the extracellular electrical potential via a system of two PDEs coupled nonlinearly by a bunch of ODEs. Its difference, the transmembrane potential, is responsible for the excitation of the heart and its steepest gradients form an excitation wavefront propagating in time.

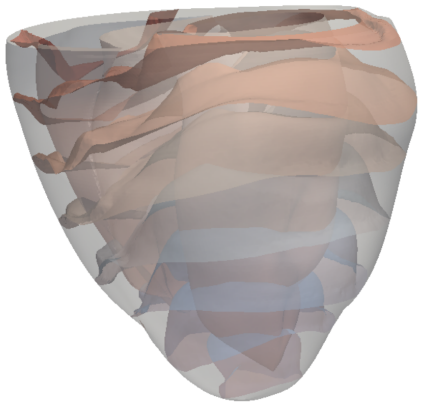

Figure 1. Wave position with increasing time (bottom to top).

This arrival time $\varphi(x)$ of the wavefront at some point $x \in \Omega$ can be approximated by the Eikonal equation (3) with given heterogeneous, anisotropic velocity information.

In our previous work [3] we address the implementation of an Eikonal solver for Shared Memory (OpenMP) and for streaming architectures in CUDA with a low memory footprint. A very short convergence time of the algorithm and good quality results are achieved, see Figure 1 wherein the wave propagation looks very smooth. We transfered the solver for a coarse model onto a tablet computer with limited high-bandwidth memory for clinical use.

Recent work in [2]–[4] has shown that building an efficient 3D tetrahedral Eikonal solver for multicore and SIMD architectures poses many challenges. Especially in the SIMD architecture where the available memory and registers are limited, it is important to keep the memory footprint low in order to reduce the costly memory accesses and achieve a good computational density on the GPU. 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.

With the growth of smart devices or embedded systems, low-memory-footprint programs have regained importance

once more. Low-memory-footprint programs are of paramount to applications on embedded platforms where memory is often constrained to within a few MBs so much so that developers typically sacrifice efficiency (processing speeds) just to make program footprints small enough to fit into the available RAM.

In the fast iterative method (FIM) [1], [2], [5] which is a method to solve the Eikonal equation, Kirby and Whitaker [2] showed the memory footprint reduction from 18 to 6 floats by storing temporarily the inner products needed to compute the 3D solution of one vertex in a symmetric matrix M'. This is a step which is performed in the wave front computations and still 18 floats have to be transferred from main memory to compute the solution.

In this paper we address a new method to reduce the memory footprint of the Eikonal solver. It is a method which reduces the number of memory transfers as well as the memory footprint by precomputing all the needed inner products for the reference tetrahedron, 18 floats in total, and then work on a mapping problem when we rotate the tetrahedron for computing the solution of another vertex. The second change precomputes only the 6 edge lengths of a tetrahedron and computes the remaining 12 angle data on-the-fly which reduces the memory footprint per tetrahedron to 6 numbers in total. In order to achieve the rotation of the tetrahedron from the reference position we use the local Gray-code numbering of edges. As a consequence we managed to reduce the memory footprint and decrease the costly memory accesses.

This paper provides mathematical and implementation details of the method followed by comparative results for the shared memory and GPU implementations using CUDA (parallel computing platform and programming model for CUDA-enabled GPUs [6]). We have implemented this method to the Eikonal solver for OpenMP and CUDA with satisfactory results. We get a significant memory footprint reduction and a good performance improvement.

II. 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.

This Eikonal equation is a non-linear PDE

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

where Ω is an open set in $\mathbb{R}^n$ with well-behaved boundary, $F(x)$ is a function with positive values, ∇ denotes the gradient and $|.|$ represents the Euclidean norm. The right-hand side $F(x)$ is typically supplied as known input. Physically, the solution u(x) is the shortest time needed to travel from the boundary $\partial\Omega$ to x inside Ω, with $F(x)$ being the time cost (not speed) at x.

In the special case when F=1, the solution represents the signed distance from $\partial\Omega$.

A. Variational form of Eikonal equation.

The variational formulation of Eikonal equation [7]

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

describes 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 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 interpolating within each tetrahedron.

B. Definition of the local solver.

The local solver calculates the solution at a vertex based on given solution of the Eikonal equation on three other vertices of a given tetrahedron.

The solution $\phi(x)$ representing the travel time for each vertex is computed using the linear approximation on its one-ring tetrahedrons (all the neighboring tetrahedrons of vertex x). This is also the bulk of the work computed by the local solver.

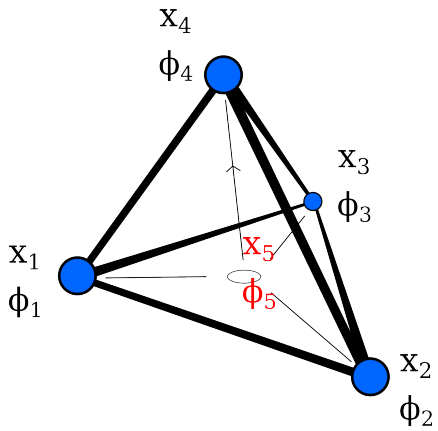

Figure 2. The local solver updates the value of the approximation at the vertex x_4 from the values φ_1, φ_2 and φ_3 of the other three vertices. The wave propagates towards x_4 and enters the triangle $\triangle_{1,2,3}$ at x_5.

We use an upwind scheme to compute the solution ϕ_4, assuming the values ϕ_1, ϕ_2, and ϕ_3 comply with the causality property of the Eikonal solutions [8]. Since the speed function is constant within each tetrahedron the travel time to x_4 is determined by the time associated with the segment from x_5 to x_4 as depicted in Figure 2, that minimizes the solution value at x_4. For this reason we first compute the coordinates of x_5 which is the point the wave-front intersects first the plane defined by the vertices x_1, x_2 and x_3. Then we check whether x_5 is contained in that plane.

Denoting $\phi_{5,4} = \phi_4 - \phi_5 = \sqrt{e_{5,4}^T M e_{5,4}}$ as travel time from x_5 to x_4, the minimization problem regarding ϕ_4 can be formulated as

$$\phi_4 = \phi_5 + \sqrt{e_{5,4}^T M e_{5,4}} \ . \qquad (4)$$

Based on Fermat's principle [9] the goal is to find the location of x_5 that minimizes ϕ_4. We specify the location of x_5 as the center of mass and by using the barycentric coordinates we can express the position of x_5 as $x_5 = \lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3$ and approximate the solution at x_5 as $\phi_5 = \lambda_1\phi_1 + \lambda_2\phi_2 + \lambda_3\phi_3$ where $\lambda_1 + \lambda_2 + \lambda_3 = 1$. If we plug ϕ_5 into equation (4) then we get the following expression for ϕ_4:

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

We take the partial derivatives of (5) with respect to λ_1 and λ_2 and equate them to zero in order to find the location of x_5 that minimizes ϕ_4.

Kirby and Whitaker [2] 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:

$$\begin{aligned} e_{5,4} &= x_4 - x_5 = x_4 - (\lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3) \\ e_{5,4} &= [e_{1,3} e_{2,3} e_{3,4}]\lambda \quad \text{where} \quad \lambda = [\lambda_1 \lambda_2 1]^T \ . \end{aligned} \quad (6)$$

Hence they obtain equation:

$$\begin{aligned} e_{5,4}^T M e_{5,4} &= \lambda^T [e_{1,3}^T e_{2,3}^T e_{3,4}^T]^T M [e_{1,3} e_{2,3} e_{3,4}]\lambda \\ e_{5,4}^T M e_{5,4} &= \lambda^T M' \lambda \end{aligned} \quad (7)$$

where

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

Finally they end up with the following system of equations:

$$\begin{cases} \phi_{1,3}\sqrt{\lambda^T M' \lambda} = \lambda^T \alpha \\ \phi_{2,3}\lambda^T \alpha = \phi_{1,3}\lambda^T \beta \ . \end{cases} \quad (8)$$

Solving equation (8) only requires to store M', which is symmetric and therefore only six floats per tetrahedron are required. This is only valid when we know in advance which vertex $\in \{x_1, x_2, x_3, x_4\}$ is unknown. Computing M' requires to store all coordinates of the vertices and the components of M, which is 18 floating point values in total that have to be transferred from global memory.

We derive in this paper a different method that stores/transfers only 6 floats per tetrahedron and computes on the fly all the other data based on this 6 floats.

III. Local Gray-code numbering of edges.

In section II-B we showed that the matrix M' is needed to find the solution to ϕ_4. This matrix is symmetric and only 6 values per tetrahedron are needed to be stored in the memory. But except that we need to store also the 4 coordinates of 4 nodes of the tetrahedron and 6 matrix M, $(4 \times 3 + 6)$ floats from global memory + local storage for edge $e_{i,j}$. A further reduction of the memory footprint requires some precomputations and a local Gray-code numbering of edges.

A. Precomputing the scalar products.

The first idea is to precompute M'. The derivations in the previous section always assume ϕ_4 as unknown as in Figure 3 and for this vertex the matrix M' has to be precomputed. During the wave front propagation one of the other 3 vertices of the tetrahedron could be also the unknown. Therefore we have to precompute all 4 local M'-matrices using only one of them during the computation. This will not reduce the memory footprint but leads additionally to some poor branchy code which is not exactly what we are looking for. Moreover it results in an increased memory footprint. The first

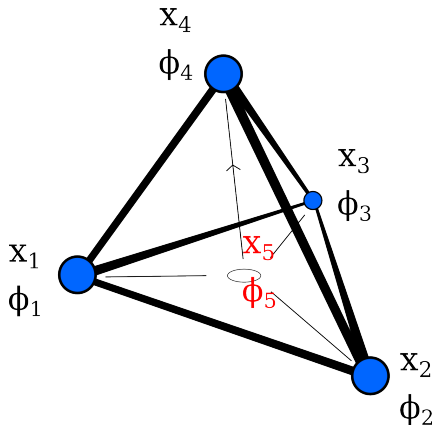

Figure 3. The reference case where the local solver updates the value of the approximation at the vertex x_4 from the values φ_1, φ_2 and φ_3 of the other three vertices.

solution to this problem consists in precomputing all 18 values potentially needed for all M's (4×3 angles + 6 edges), store them in a symmetric sparse 6×6 matrix per tetrahedron and then work on a mapping problem without branchy code and lookup tables. Access to lookup tables would be totally random and would produce expensive non-coalesced memory accesses especially in streaming architectures. Instead of optimizing the memory access patterns we will reduce the costly accesses to global memory.

This approach does not reduce the memory footprint significantly 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.

B. Storing the precomputed scalar products.

Using a Gray-code [10] of 4 bits length we are able to identify uniquely all possible objects in a tetrahedron, see Figure 4. Each vertex is represented by a 4-bit number with exactly one bit set and edges are represented by a 4-bit number with exactly two bit set. And just by the *bitwise or* operation between two vertices we determine the edge connecting them.

There are a lot of operations available for the Gray-code giving us the flexibility to build our mapping system. For our solver

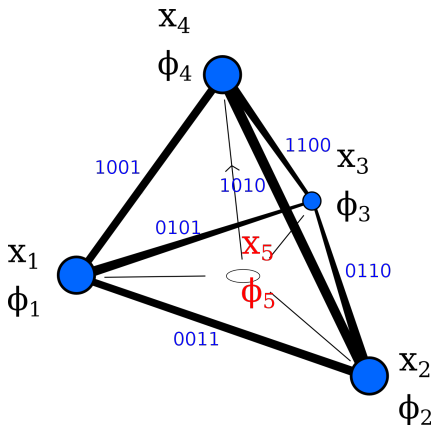

Figure 4. Local Gray-code numbering of edges in a tetrahedron.

we are only interested in the edges since all these M-scalar products include always two edges of the tetrahedron. Table I shows the available Gray-codes numbers for the edges, i.e., the three edges connected to x_3 have the Gray-code numbering 5, 6 and 12. This Gray-code numbering will simplify the rotation of the tetrahedron in §IV.

TABLE I
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

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

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 II we have to transform these Gray-code edge numbers to the edge index set $\{0, \ldots, 5\}$. This transformation is performed by the linear regression function $\mathbf{f(d)}$ from Table I, i.e., the edge $e_{1,3}$ from x_1 to x_3 has the Gray-code number $d = 5$ with $\mathbf{f(5)} == 1$ and so we will store the scalar products with this edge in row/column 1 in Table II.

TABLE II
M-SCALAR PRODUCTS STORED IN THE 6X6 SYMMETRIC MATRIX T_M.

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

The integer formula $\mathbf{f_{int}(d)}$ = **d**/2 - 1 avoids floating point operations completely and only one bit shift followed by one integer decrement is needed in this transformation function.

C. Addressing the precomputed scalar products using a mapping system.

Now it only remains to find which scalar products we will need when we are solving for a certain node. In our reference case we are solving again for vertex 4.

We observe from matrix M' in (9) 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.

$$M' = \begin{cases} [e_{1,3}^T M e_{1,3},\ e_{2,3}^T M e_{1,3},\ e_{3,4}^T M e_{1,3}]^T \\ [e_{1,3}^T M e_{2,3},\ e_{2,3}^T M e_{2,3},\ e_{3,4}^T M e_{2,3}]^T \\ [e_{1,3}^T M e_{3,4},\ e_{2,3}^T M e_{3,4},\ e_{3,4}^T M e_{3,4}]^T \ . \end{cases} \tag{9}$$

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 II by the following general procedure:

Algorithm 1 Accessing M' wrt. common vertex j

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

The reference case for unknown x_4 calls Algorithm 1 with common vertex 3 and returns those precomputed entries in Table II 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.

From this first change the memory footprint is still the same but compared to the previous version only 6 non-coalesced accesses are needed in total to the global memory instead of 18 accesses plus computations. This means that less expensive memory accesses per tetrahedron is required.

IV. ROTATING ELEMENTS INTO THE REFERENCE CONFIGURATION.

The previous section explained the computation of the wave front with the unknown quantity in x_4. If the wave front hits a tetrahedron differently then we have to map all formulas with respect to an unknown quantity $\in \{x_1, x_2, x_3\}$. The required change in the edge numbers can be achieved by appropriate bit shifts.

A. The sign problem.

In addition to the bit shift of the edges we have to take into account that the direction of the edges of interest changes. That will cause different signs in front of the mixed inner products. All 18 needed scalar products have been precomputed using the reference tetrahedron as in Figure 5 (left). The remaining three cases are depicted in Figures 5 and 6 and these different configurations can be interpreted as rotation of the tetrahedron. When solving for x_3 we need to rotate by 1 position from the reference configuration. Similarly, a rotation by 2 positions from the reference position happens for x_2. Lastly when solving for x_1 one needs a rotation by 3 positions. In all cases the directions of the edges with respect to the reference configuration also change. This means that we have to take into account a sign change in the mixed scalar product computations. As a consequence, it is not sufficient to access the needed scalar products for a certain vertex from the precomputed 6×6 symmetric matrix T_M via Algorithm 1 but we have to take into account also the sign changes. Let us investigate for each of the four cases seperately how the signs change and how we handle this issue.

We start by the reference configuration in Figure 5 (left). As explained in section III-C, x_3 is the common vertex when solving for vertex x_4. The mapping system together with the common vertex provides all the edge numbers needed in the computations regarding ϕ_4. The resulting edge numbers $(5, 1, 2)$ are marked marked in blue and bold in Figure 5 (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 Figure 5 (right). Notice that edge $e_{4,2}$ changed the direction with respect to the directions in the reference configuration in Fig. 5 (left) and so one negative sign has to be considered for this edge. Therefore, all precomputed scalar products that contain edge $e_{4,2}$ in the computation of ϕ_3 are going to be multiplied by the negative sign.

In the third case when the solution of x_2 is required, a rotation by 2 positions is done. The common vertex is x_1 and the needed edge numbers to compute ϕ_2 are $(0, 1, 3)$ which refer to those tetrahedral edges in Figure 6 (left) marked in blue and bold. In this case edge $e_{4,1}$ and edge $e_{3,1}$ change their direction and so we have to consider two sign changes in difference to the reference case.

Only edge $e_{4,1}$ changed direction so only one sign change with respect to the reference direction is needed. A rotation by 3 positions is done in the final case with an unknown solution in x_1. The common vertex is x_4 and the needed edge numbers are $(3, 4, 5)$, see Fig. 6 (right). Only edge $e_{4,1}$ changes its direction and so only one sign change has to be taken into account.

Ignoring the sign changes derived above would lead to incorrect wave front computations.

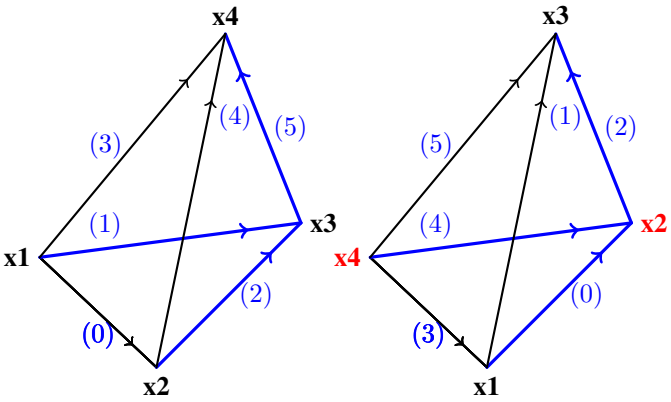

Figure 5. Reference tetrahedron to the left. Rotation with 1 position to the right. Edges are named from 0 to 5 based on the edge numbers from Table I.

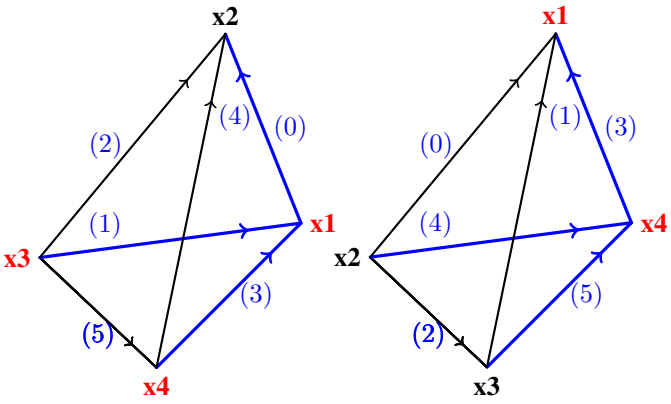

Figure 6. Rotation with 2 positions to the left. Rotation with 3 positions to the right. Edges are named from 0 to 5 based on the edge numbers from Table I.

B. Bit shift.

The implementation of the four cases should avoid branchy code, unnecessary memory allocations and costly memory accesses. In order to achieve these goals we interprete the geometric rotations as bit shifts to the right of the Gray code edge numbering from the reference configuration.

The reference position in this refrence case is the 3-tuple $(5, 1, 2)$ of edge numbers. We represented this numbers in the binary format and applied a bit shift to the right with as many bits as the number of position needed to rotate in order to get to the required position. All the results are showed in Table III. The bit representation for each row is calculated by one bit shift to the right from the previous row. The last column shows how many bit shifts to the right are needed to get to the current representation from the reference configuration. So for calculating solution ϕ_4, 0 bit shifting are needed. For calculating ϕ_3, 1 bit shift is needed. For ϕ_2, 2 bit shifting are needed and for ϕ_1, 3 bit shifts. This is the same number as the number of rotations needed from the reference configuration to get to one of the other configurations. Notice the 4 right bits in red of the bit representation. They give immediately

the sign. If only one bit is set to the last 4 bits marked in bold and red then we get a negative sign for the corresponding edge. If 2 bits or none are set then the sign does not change with respect to the reference configuration. Two bits set means twice negative sign which is positive at the end. In this way

TABLE III
ROTATION OF ELEMENTS TO THE REFERENCE CONFIGURATION BY BIT SHIFTING OBTAINING THE SIGN AND THE EDGE NUMBERS.

ϕ	Edges	Sign	Bit representation of edge and sign			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**	$\gg 3$

by implementing this method we avoid branchy code and unnecessary memory allocations since this is a straight forward method where from a reference configuration one can get the three other configurations just by bit shifting every edge. The first 4 bits result in the edging numbers needed to address the products (second column Table III) and the second 4 bits to the corresponding sign (third column Table III). Moreover this can also be precomputed and referenced on the fly during the wave front computations.

The sign change with respect to edges has to be considered only for scalar products with two different edges, i.e., for the off-diagonal elements of T_M. A redirected edge in a diagonal element of T_M would result in a multiplication by $(-1)\cdot(-1)$ and has no consequences.

V. MEMORY FOOTPRINT REDUCTION FROM 18 TO 6 FLOATS.

The possibility to reduce the memory footprint from 18 to 6 floats originates from the fact that $\langle \vec{e}_{k,s}, \vec{e}_{s,\ell} \rangle_{M^\tau}$ $(k \neq \ell)$ represents an angle of a surface triangle whereas $\langle \vec{e}_{k,s}, \vec{e}_{k,s} \rangle_{M^\tau}$ represents the length of an edge in the M-metric which are also the products of 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 as we will show it below. Therefore we only have to precompute the 6 edge lengths of a tetrahedron and compute the remaining 12 angle data on-the-fly which reduces the memory footprint per tetrahedron to 6 numbers.

There are actually three different cases in angel computations as showed in Figure 7 for angles α, β and γ. The term angle represents in this paper always 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

$$\begin{aligned} e_{2,3}^T M e_{2,3} &= (e_{1,3} - e_{1,2})^T M (e_{1,3} - e_{1,2}) \\ &= e_{1,3}^T M e_{1,3} + e_{1,2}^T M e_{1,2} - 2 e_{1,2}^T M e_{1,3} \end{aligned} \quad (10)$$

such that angel α can be expressed as:

$$e_{1,2}^T M e_{1,3} = \tfrac{1}{2}(e_{1,3}^T M e_{1,3} + e_{1,2}^T M e_{1,2} - e_{2,3}^T M e_{2,3}) \ . \quad (11)$$

The scalar product $e_{1,2}^T M e_{1,3}$ from equation (11) contains Gray-code edges $k = 3$ and $\ell = 5$ that are mapped via f(d) to numbers 0 and 1 (see Table I) such that the scalar product can be found in Table II at position (0,1).

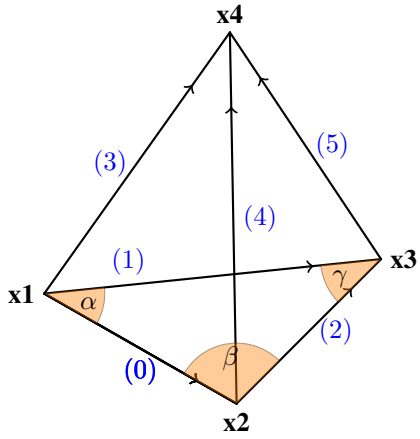

Figure 7. Angle calculation in a tetrahedron. Edges are named from 0 to 5 based on the edge numbers from Table I.

In this case the direction of all edges $e_{i,j}$ in the equation did not change with respect to the reference directions in the tetrahedron in Figure 6. 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 section III-A.

During the angle computations the directions might change with respect the reference edge directions. This will affect the sign of the resulting formula as showed in the other cases to follow.

To calculate $\beta = e_{1,2}^T M e_{2,3}$ we reformat $e_{1,3}$ 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

$$\begin{aligned} e_{1,3}^T M e_{1,3} &= (e_{2,3} + e_{1,2})^T M (e_{2,3} + e_{1,2}) \\ &= e_{2,3}^T M e_{2,3} + e_{1,2}^T M e_{1,2} + 2 e_{1,2}^T M e_{2,3} \ . \end{aligned} \quad (12)$$

We have to express each edge $e_{m,n}$ participating in the angel calculation based on the reference edge direction. Edge $e_{1,3} = e_{2,3} - e_{2,1}$ from above contains $e_{2,1}$ but this edge is not directed in the reference edge direction which would be $e_{1,2}$, see the

directions in Figure 6. For this reason $e_{1,3}$ is reformatted as $e_{1,3} = e_{2,3}+e_{1,2}$ since $e_{2,1} = -e_{1,2}$ as shown in equation (12).

As a result angel β can be expressed as in equation (13):

$$e^T_{1,2}Me_{2,3} = \text{-}\,\tfrac{1}{2}(e^T_{2,3}Me_{2,3}+e^T_{1,2}Me_{1,2}-e^T_{1,3}Me_{1,3}) \ . \quad (13)$$

Since we have only one change of sign for one of the edges only one minus sign is carried to equation (13).

The scalar product $e^T_{1,2}Me_{2,3}$ from equation (13) can be found in position (0,2) or translated to non-converted range in position (3,6) in Table II. Meaning that $k = 3$ and $\ell = 6$ in this case.

Calculating $\gamma = e^T_{1,3}Me_{2,3}$ requires reformatting of $e_{1,2}$:

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

and yields to the expression

$$\begin{aligned} e^T_{1,2}Me_{1,2} &= (-e_{2,3} + e_{1,3})^T M(-e_{2,3} + e_{1,3}) \\ &= e^T_{2,3}Me_{2,3} + e^T_{1,3}Me_{1,3} - 2e^T_{1,3}Me_{2,3} \ . \quad (14) \end{aligned}$$

In this case we have two sign changes one for edge $e_{3,2}$ and the other for edge $e_{3,1}$ as showed in equation (14) marked in bold. The edges are formatted to their reference configuration. Both of this negative signs are carried to equation (15) converting it again to positive sign.

As a result angel γ can be expressed as in equation (15):

$$e^T_{1,3}Me_{2,3} = (\text{-})(\text{-})\tfrac{1}{2}(e^T_{2,3}Me_{2,3} + e^T_{1,3}Me_{1,3} - e^T_{1,2}Me_{1,2}) \ . \quad (15)$$

The scalar product $e^T_{1,3}Me_{2,3}$ from equation (13) can be found in position (1,2) or translated to non-converted range in position (5,6) in Table II. Meaning that $k = 5$ and $\ell = 6$ in this case.

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. We can write as a generalization

$$e^T_k Me_l = esgn * \tfrac{1}{2}(e^T_k Me_k + e^T_l Me_l - e^T_s Me_s) \quad (16)$$

with $s = (k \hat{} l)$ where $\hat{}$ is the bitwise xor operator and $k \neq \ell$ are the Gray-code edge numbers from range [3-12] from Table I. The sign has to be calculated only for the non-diagonal elements as follows:

$$esgn = ((2 * (s > k) - 1) * (2 * (s > l) - 1)) \ . \quad (17)$$

Finally, we have to store only the diagonal part $e^T_k Me_k$ as in Table IV and all mixed elements are computed via formula (16).

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). Now with the reduced memory footprint to 6 floats only 3 products from the diagonal are used. Respectively the products referenced from edge numbers (1,1), (2,2) and (5,5). These are the products marked in blue in Table IV. The other 3 needed products for the mixed edge numbers, respectively (1,2),(1,5) and (2,5) which reference exactly the non-diagonal products in blue in Table II, are calculated based on formula (16).

TABLE IV
REDUCED NUMBER OF SCALAR PRODUCTS STORED IN A 6X6 SYMMETRIC MATRIX.

0	**1**	**2**	**3**	**4**	**5**
$e^T_{1,2}Me_{1,2}$	$e^T_{1,3}Me_{1,3}$	$e^T_{2,3}Me_{2,3}$	$e^T_{1,4}Me_{1,4}$	$e^T_{2,4}Me_{2,4}$	$e^T_{3,4}Me_{3,4}$

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

VI. NUMERICAL TESTS AND COMPARISONS.

We present the results for the numerical tests performed on a workstation with Intel Core i7-4700MQ CPU @2.40GHz processor and GeForce GTX 1080 GPU. The TbunnyC heart mesh with 3,073,529 tetrahedrons and 547680 vertices was used as a testing mesh.

Table V shows a comparison between our new implementation including the Gray-code method to reduce the memory footprint and our old implementation without the Gray-code method. An acceleration by 20% for the OpenMP implementation and 25% on the CUDA implementation are achieved bu using the local Gray-code numbering of edges method to reduce the memory footprint. On recent Intel CPUs we get similar behavior. For the CUDA implementation we get more performance improvement because the non-coalesced memory accesses to global memory are more expensive. The number of reductions is the same but more significative on the GPUs.

TABLE V
RUN TIMES ON THE WORKSTATION.

Implementations	**# Tetrahedrons**	**CUDA** sec.	**OpenMP** sec.
Without Gray-code	3,073,529	6.3	4.7
With Gray-code	3,073,529	4.7	3.8

The OpenMP implementation is currently faster than CUDA because the CUDA optimizations are still ongoing work. The next version of the domain decomposition for the wave front will exploit the shared memory of a thread blocks much better which should lead to a performance improvement.

Anyway, the point of this paper is the comparison of the implementations with or without the Gray code numbering.

Of course our new method behaves differently in different architectures and this is something to be expected. To prove it we transfered the solver with Gray-code implementation onto a tablet computer, the Nvidia Shield tablet with 2.2 GHz quad ARM Cortex A15 CPU and 2 GB RAM, which incorporates a CUDA–capable GPU for development of embedded and mobile applications such as Tegra K1 with 192 core Kepler GPU. Then in Table VI we compared between our new implementation including the Gray-code method to reduce the memory footprint and our old implementation where the Gray-code method is not included. This comparison is done for both GPU, Tegra K1, and 2.2 GHz Quad ARM Cortex A15 CPU.

TABLE VI
RUN TIMES ON THE SHIELD TABLET (ANDROID).

Implementations	**# Tetrahedrons**	**CUDA** sec.	**OpenMP** sec.
Without Gray-code	3,073,529	30.26	205.22
With Gray-code	3,073,529	20.30	97.19

An acceleration by 33% for the CUDA implementation and 52% on the OpenMP implementation are achieved by using the local Gray-code numbering of edges method to reduce the memory footprint. On embedded systems such as this tablet where memory and registers are limited the benefit is bigger. This is actually what makes our method very interesting since one might benefit more in a different situation where limited high-bandwidth memory architectures are used.

VII. CONCLUSIONS AND FUTURE WORK.

In this paper, we have presented a new method to reduce the memory footprint of an Eikonal solver. We provided mathematical and implementation details of the method and showed how the memory footprint reduction is achieved. Comparisons to previous efforts were introduced and the difference was showed in details. We proposed the local Gray-code numbering of edges as a method which includes the precomputation of the needed scalar products for all vertices and continued with the presentation of a mapping system we use in order to address the scalar products stored in a symmetric matrix. We showed how this method benefits to the performance and how does it reduce the costly memory transfers. Then we present the issues coming after this method, such as the sign problem, how we solve it and how we implement the solution efficiently. Finally we propose a method to reduce the memory footprint from 18 to 6 floats based on the conclusion that the angle information can be expressed by the combination of three edge lengths. Using this method we achieved to reduce the memory footprint significantly achieving in this way a good performance improvement. We showed how the method can be more beneficial on limited high-bandwidth memory architectures such as the embedded systems.

We believe that this is a method that will come handy in many situations when dealing with tetrahedral computations. Not only related to edge-edge computations as it happens in our case but also on vertex or face computation since this is a general method that works for all the objects of a tetrahedron or of any simplex where Gray-code can be used to number the simplex objects. It can be adopted to any problem of the kind. Moreover this might be a very interesting method on applications with limited high-memory bandwidth where we believe much better performance improvement will be achieved compared to our application which is not memory bandwidth bound.

As a future work, we will focus on improving some implementations details which we believe would give to us the possibility to fully exploit the potential of the method. This method is going to be incorporated to the domain decomposition of our CUDA parallel algorithm which we are currently implementing. From our analysis so far and preliminary results we conclude that this version will decrease the non-coalesced access level and significantly increase of computational density. And combined with the implementation of 16-bit Floating Point Data [11] it will allow to run the domain decomposition version also in one GPU and with better results on multiple accelerator cards and cluster computing.

ACKNOWLEDGEMENT

The authors gratefully acknowledge support from FWF project F32-N18, Erasmus Mundus JoinEUsee PENTA scholarship and from NAWI Graz.

REFERENCES

[1] W.-K. Jeong and R. T. Whitaker, "A fast iterative method for eikonal equations." *SIAM J. Sci. Comput.*, vol. 30, pp. 2512–2534, 2008.
[2] Z. Fu, R. M. Kirby, and R. T. Whitaker, "Fast iterative method for solving the eikonal equation on tetrahedral domains." *SIAM J. Sci. Comput.*, vol. 35, no. 5, pp. C473–C494, 2013.
[3] D. Ganellari and G. Haase, "Fast many-core solvers for the eikonal equations in cardiovascular simulations," in *2016 International Conference on High Performance Computing Simulation (HPCS)*. IEEE, 2016, pp. 278–285, peer-reviewed.
[4] M. Noack, "A two-scale method using a list of active sub-domains for a fully parallelized solution of wave equations." *Journal of Computational Science*, vol. 11, pp. 91–101, 2015.
[5] Z. Fu, W.-K. Jeong, Y. Pan, R. M. Kirby, and R. T. Whitaker, "A fast iterative method for solving the eikonal equation on triangulated surfaces." *SIAM J. Sci. Comput.*, vol. 33, pp. 2468–2488, 2011.
[6] NVIDIA, "CUDA C programming guide." http://docs.nvidia.com/cuda/cuda-c-programming-guide.
[7] J. P. Keener, "An eikonal-curvature equation for action potential propagation in myocardium." *Journal of Mathematical Biology*, vol. 29, pp. 629–651, 1991.
[8] J. Qian, Y.-T. Zhang, and H.-K. Zhao, "Fast sweeping methods for eikonal equations on triangulated meshes." *SIAM J. Numer. Anal.*, pp. 83–107, 45 (2007).
[9] D. D. Holm, "Geometric mechanics: Part I: Dynamics and symmetry, 2nd ed." *Imperial College London Press, London, UK*, 2011.
[10] Weisstein. Eric W, "Gray code." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/GrayCode.html.
[11] NVIDIA, "New Features in CUDA 7.5." https://devblogs.nvidia.com/parallelforall/new-features-cuda-7-5.