Stepping into fully GPU accelerated biomedical applications

Caroline Mendonca Costa[2], Gundolf Haase[1], Manfred Liebmann[1], Aurel Neic[1], and Gernot Plank[2]

[1] Institute for Mathematics and Scientific Computing, University of Graz, Austria**
`gundolf.haase@uni-graz.at`,
WWW home page: `http://http://www.uni-graz.at/~ghaase`
[2] Institute of Biophysics, Medical University of Graz, Austria

Abstract. We present ideas and first results on a GPU acceleration of a non-linear solver embedded into the biomedical application code CARP. The linear system solvers have been transferred already in the past and so we concentrate on how to extend the GPU acceleration to larger portions of the code. The finite element assembling of stiffness and mass matrices takes at least 50% of the CPU time and therefore we investigate this process for the bidomain equations but with focus on later use in non-linear and/or time-dependent problems. The CUDA code for matrix calculation and assembling is faster by a factor up to 90 compared to a single CPU core. The routines were integrated to CARP's main code and they are already used to assemble the FE matrices of the bidomain model. Further performance studies are still required for the bidomain-mechanics model.

1 Introduction

During the last years GPUs became very attractive to reduce simulation time by porting linear solvers to the accelerator card. Due to the large problem size multigrid methods have been preferred by parts of the community. GPU accelerated geometrical multigrid has been carefully investigated by several authors, see [6, 5] for structured grids and its further development for locally structured grids in [3] as examples for success in a monolithic code. Starting from third party demands and large unstructured discretizations, the algebraic multigrid (AMG) has to be applied. Here, the authors proved one order of magnitude acceleration by using GPUs [7, 16] also in the multi-GPU context [13]. While the AMG setup in our code still remains on the CPU there is an interesting attempt to move also the AMG setup completely onto the GPU [1] but only with an acceleration of two.

** The final publication can be found in I. Lirkov, S. Margenov, and J. Wasniewski, editors, Large Scale Scientific Computing LSSC13, volume 8353 of Lecture Notes in Computer Science, pages 314. Springer, 2014 and it is available at http://www.springer.com/computer/theoretical+computer+science/book/978-3-642-29842-4

The situation changes when the linear solvers are embedded into a larger framework. When solving the Bidomain equations to simulate cardiac electrophysiology via the Finite Element Method, the stiffness and mass matrices have to be assembled only once, since the spatial domain is not modified during computation. Thus, in this case, the assembly of the FE matrices it is not a bottleneck of computation. On the other hand, when solving the bidomain-mechanics model, which involves non-linear elasticity, the FE matrices must to be updated at each Newton step, as the spatial domain is deformed, which becomes very expensive as the system increases in size. Therefore, it is the goal of this paper, to implement a highly efficient FE assembly routine using CUDA [14] to increase performance when solving this model. This paper investigates the GPU acceleration for the FE matrix computation in the bidomain model in order to study whether a GPU implementation might pay off for the real challenging bidomain-mechanics equations [15].

We will introduce the bidomain model in §2 providing the equations for the numerical tests. Section 3 starts with a brief primer on the simulation code CARP and presents the strategy how to reduce critical data transfer between CPU and GPU memory in the non-linear solver when AMG is used as linear solver therein. The improvement of the FE matrix calculations by vectorization and GPU acceleration is described in §4. The paper finishes with speedups regarding the matrix computation and assembling on GPU and with some conclusions.

2 The Bidomain model

The bidomain equations in the elliptic-parabolic form are given by

$$\begin{bmatrix} -\nabla \cdot (\boldsymbol{\sigma}_i + \boldsymbol{\sigma}_e)\, \nabla \phi_e \\ -\nabla \cdot \sigma_b \nabla \phi_e \end{bmatrix} = \begin{bmatrix} \nabla \cdot \boldsymbol{\sigma}_i \nabla V_m + I_i \\ I_e \end{bmatrix} \tag{1}$$

$$I_m = (\nabla \cdot \boldsymbol{\sigma}_i \nabla \phi_i)$$

$$I_m = C_m \frac{\partial V_m}{\partial t} + I_{ion}(V_m, \boldsymbol{\eta}) - I_i \tag{2}$$

$$\frac{d\boldsymbol{\eta}}{dt} = f(t, \boldsymbol{\eta}) \tag{3}$$

$$V_m = \phi_i - \phi_e \tag{4}$$

where ϕ_i and ϕ_e are the intracellular and extracellular potentials, respectively, $V_m = \phi_i - \phi_e$ is the transmembrane voltage, $\boldsymbol{\sigma_i}$ and $\boldsymbol{\sigma_e}$ are the intracellular and extracellular conductivity tensors, respectively, β is the membrane surface to volume ratio, I_m is the transmembrane current density, I_e are extracellular stimuli applied in the extracellular space, I_i is an intracellular current stimulus, C_m is the membrane capacitance per unit area, and I_{ion} is the membrane ionic current density which depends on V_m and a set of state variables, $\boldsymbol{\eta}$ which is defined by f.

At tissue boundaries, no flux boundary conditions are imposed for ϕ_i, with the potential ϕ_e and the normal component of the extracellular current being continuous. At boundaries of the conductive bath surrounding the tissue, no flux boundary conditions for ϕ_e are imposed.

Combining the interstitial and bath spaces into the extracellular space, the bidomain equations can be written as follows

$$\begin{aligned}
-\nabla \cdot \boldsymbol{\sigma}_e \nabla \phi_e &= \nabla \cdot \boldsymbol{\sigma}_i \nabla \phi_i + I_e && (5)\\
\beta I_m &= \nabla \cdot \boldsymbol{\sigma}_i \nabla \phi_i &&\\
I_m &= C_m \frac{\partial V_m}{\partial t} + I_{ion}(V_m, \boldsymbol{\eta}) - I_i && (6)\\
\frac{d\boldsymbol{\eta}}{dt} &= f(t, \boldsymbol{\eta}) && (7)\\
V_m &= \phi_i - \phi_e && (8)
\end{aligned}$$

With no-flux boundary conditions imposed for ϕ_i and ϕ_e.

The matrix representation for the FE discretization for the bidomain equations, written for V_m and ϕ_e only, is given by

$$\begin{aligned}
K_{ie}\phi_e &= -P(K_i V_m) - M_e I_e && (9)\\
K_i V_m &= -\beta M_i I_m - K_i(P^T \phi_e) && (10)
\end{aligned}$$

where K_* and M_* are stiffness and mass matrices, respectively, with $_* = e|i$ being either the extracellular space, Ω_e, or the intracellular space, Ω_i, P is a prolongation operator from Ω_i to Ω_e and its transpose, P^T, is a restriction operator from Ω_e to Ω_i.

3 GPU strategy for non-linear FE solvers

3.1 CARP environment

The CARP environment [18, 19] (Cardiac Arrhythmia Research Package) is a collection of various contributors for the detailed simulation of cardiovascular phenomena, see Fig. 1 for the software scheme. The gray box in the center contains the kernel for the linear algebra that has to be combined with the non-linear iteration in case of the bidomain-mechanics model. The FE assembly routine is implemented within the CARP environment. The assembly involves the module FEM, which comprises all the finite element computations, particularly the stiffness and mass matrices assembly, and contains the “Matrix Market”, which comprises matrix basic operations and is implemented within the Module FMatrix. This module is subject to GPU acceleration in this paper for the matrices resulting from the bidomain equations (5). This is meant as a study whether a GPU implementation might pay off for the real challenging bidomain-mechanics equations, see [15]. We use unstructured tetrahedral FE meshes with linear test functions.

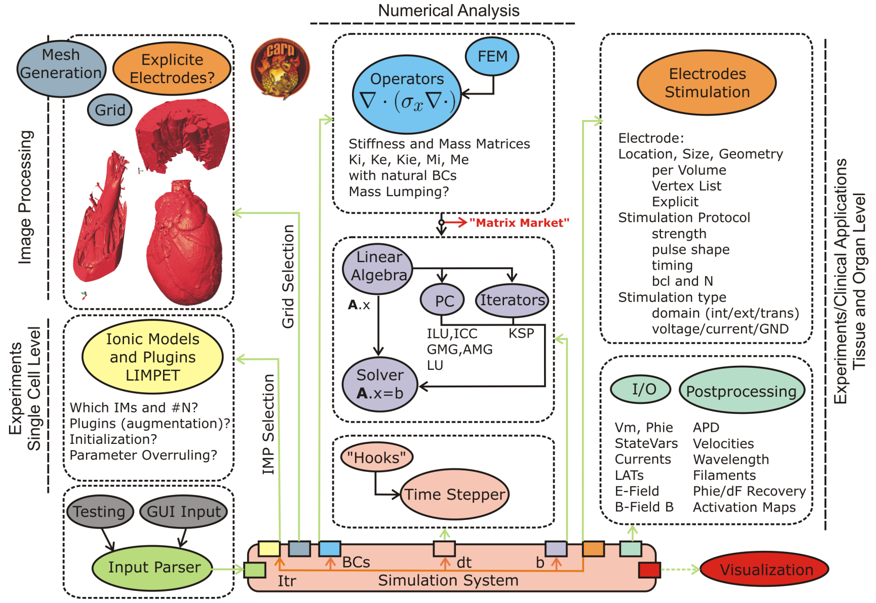

Fig. 1. Structure of the CARP code.

3.2 Draft for a non-linear solver on GPUs

In the context of a non-linear setting where we have to solve frequently a linear system as

$$K\left(u^{\text{old}}\right) u^{\text{new}} = f\left(u^{\text{old}}\right) \quad . \tag{11}$$

It makes not much sense to accelerate the application of the linear solver (cg with AMG preconditioning) by a factor of 10 on the GPU when the setup of the AMG solver as well as the re-calculation and the re-assembling of the stiffness matrix $K\left(u^{\text{old}}\right)$ are still performed on the CPU. Additionally we have to avoid costly data transfer between CPU and GPU memory. We should also take into account that calculations are very fast on the GPU in contrast to the slow performing search and reorder routines.

The CARP code does not use spatial adaptivity and therefore the topology of the mesh will remain unchanged throughout the non-linear computation, and even during the outer time integration. Therefore, we have to determine the matrix pattern only once in a matrix setup on the CPU, afterwards transfer that pattern once to the GPU and perform (re-)calculation and the (re-)assembling of the stiffness on the GPU repeatedly. Clearly, that requires that the mesh information and the material properties are also available in GPU memory. Section §4 will report on first experiences regarding the matrix calculation on GPU.

The AMG preconditioner setup contains parts which do not accelerate well on a GPU [1] so a closer look at it is necessary. The AMG setup consists of the following parts:

1. Find coarse/fine nodes.
2. Determine interpolation pattern.
3. Calculate interpolation matrix entries.
4. Determine coarse matrix pattern.
5. Calculate coarse matrix entries.

Due to CPU performance issues items 2./3. as well as items 4./5. are handled usually in one routine. In the CARP context we can again assume that material anisotropies will not change dramatically, i.e., that the coarse/fine splitting will remain the same in all (or many) non-linear steps. The same will be assumed for the pattern determinations in items 2. and 4. This indicates a splitting of the setup such that items 1., 2. and 4. are still performed once on the CPU while items 3. and 5. can be handled very efficiently on the GPU. This splitting is still subject to investigation.

4 FE matrix calculation on GPU

4.1 Stiffness matrix

The entries of stiffness matrices K from (9) and (10) are computed as

$$K = \{K_{i,j}\} = \sum_{e=0}^{nElem-1} K_e = \sum_{e=0}^{nElem-1} \left(-C_e\ G_e\ C_e^T\ vol_e\right), \tag{12}$$

where K_e is the stiffness matrix of element e, C_e is the matrix of basis coefficients, G is the matrix of conductivity tensors and vol is the element volume [4, 9, 10]. Using tetrahedral elements, the basis coefficients of each element are given by the inverse of the matrix

$$C_e^{-1} = \begin{bmatrix} 1\ x_1\ y_1\ z_1 \\ 1\ x_2\ y_2\ z_2 \\ 1\ x_3\ y_3\ z_3 \\ 1\ x_4\ y_4\ z_4 \end{bmatrix} \tag{13}$$

The conductivity tensor of each element is computed as

$$G = g_f\ f \times f + g_s\ s \times s + g_n\ n \times n, \tag{14}$$

for the orthotropic case, where f, s, n are the longitudinal, transverse and normal eigenaxis, g_f, g_s, g_n are the principal eigenvalues. The volume of each tetrahedra is computed as $vol = \frac{|det(C_e)|}{6}$.

The mass matrix $M = \{M_{ij}\}$ in (10) is also computed element wise and simplifies to

$$M_{ij} = \begin{cases} \sum_{e=0}^{nElem-1} M_{e,ij} = 2 \text{ factor } vol_e, & i = j \\ \sum_{e=0}^{nElem-1} M_{e,ij} = \text{factor } vol_e, & i \neq j \end{cases} \tag{15}$$

for linear elements with the factor depending on simple material coefficients.

4.2 The FMatrixArray structure

The current interface between matrix element calculation and its accumulation into the global matrix consists of a structure FMatrixArray containing a large array of size *number of elements* × *number of nodes per element* with some additional information that stores all the element matrices. This allows to calculate the elements matrices in parallel on many-core chips without fatal data races. Additionally, all the information needed for element matrix calculation as coordinates and material coefficients is also stored as FMatrixArray structures with 1D arrays of appropriate size. Although this approach requires additional temporal memory it outperforms the classical approach without redundant storing of input data and of accumulating the local entries directly into the global matrix by a factor of 5 on a single CPU core.

Specialized explicit expressions were written to compute matrix determinant and matrix inversion, which are the most expensive routines. An example of the vectorized code using an FMatrixArray `ent` to compute the determinant is shown below for a triangular element. The code for a tetrahedra looks the same just much longer. Note that only local variables and explicit expressions are used.

Listing 1.1. Code to compute matrix determinant (vol_e) for a triangular element

```
switch (rows)
  case 3: {
    for(int i = 0; i < nmats; i++) {
      const Real a11 = ent[i*matSize],   a12 = ent[1+i*matSize],
                 a13 = ent[2+i*matSize], a21 = ent[3+i*matSize],
                 a22 = ent[4+i*matSize], a23 = ent[5+i*matSize],
                 a31 = ent[6+i*matSize], a32 = ent[7+i*matSize],
                 a33 = ent[8+i*matSize];
      det[i] = (a12*a23 - a13*a22)*a31
             - (a11*a23 - a13*a21)*a32
             + (a11*a22 - a12*a21)*a33;
    }
```

This calculation of the determinant belongs to the volume computation in listing 1.2. The code below gets the element list and the node list as input parameters and computes all local stiffness matrices after the appropriate setup of volume, C_e from (13) and G from (14) for each element. The listing 1.1 is representative for the data handling in all subroutines involved.

Listing 1.2. Code to compute element stiffness matrices K_e

```
int fl_tetFillLocalStiffnessMatrixArray_
   ( const ElemList *elst, const NodeList *nlst,
           FMatrixArray *nodes, FMatrixArray *coeffs,
           FMatrixArray *g,     FMatrixArray *lK,     Real *vols )
{
  fl_fillNodes(elst, nodes, nlst);         // Fill coordinates
  fl_computeVolumes(nodes,vols);           // Compute Volumes
  FMatrix_InvArray(nodes,coeffs);          // Comp. Basis Coefficients C_e
  Real pevs[] = {1.0, 1.0, 1.0};           // simple material parameters
  fl_getCondTensorGPU(elst,g,pevs);        // Comp. Conductivity Tensor G
  fl_integrateStiffness(lK,g,coeffs,vols);// Comp. local matrices
}
```

The calculation of the mass matrices is handled the same way.

4.3 Implementation of the CUDA kernels

Non-coalesced memory allocation and access Matrix entries (stored linearly) are reordered as shown in Fig. 2, so that each thread has access to the first element of its corresponding matrix in cache memory. In the GPU implementation the stride value is 32, which is half the number of maximal threads per block, i.e. 64 threads per block. When allocating memory for an array in non-coalesced format the size of the memory chunk will depend also on the size of this stride. Keeping arrays in both formats is important to obtain maximum performance in both CPU and GPU, as the non-coalesced format is inefficient in the CPU, but the most efficient in the GPU. Therefore, whenever we copy data over to or from the GPU, a conversion routine has to be used.

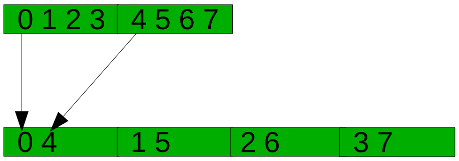

Fig. 2. Coalesced (top) *versus* non-coalesced (bottom) memory access. Matrix entries are reordered using a predefined stride size.

Data structure on the GPU In order to copy data to the GPU, some changes were required in the Element list and Nodes list structures implemented in the standard code. In the standard code, this lists are implemented as general structures holding detailed information about each element and node in the mesh. In the CUDA version, this lists are implemented as one-dimensional arrays, which size varies depending on the element type and number of elements in the mesh. More detailed information is given below and in Fig. 3, where N and L are the global and local indices describing each element, respectively; Lon, Sheet and Sheet normal are the fiber orientation arrays; and Pts and Exp. Pts are the arrays of points in regular and exploded format, respectively.

Element list: array of nodal (local or global) indices describing each element, it is copied in non-coalesced form to the GPU.

Axes lists: arrays of longitudinal, sheet and sheet normal fiber orientations, it is copied to the GPU in non-coalesced form.

Nodes list: an array of point coordinates is copied to the GPU in coalesced form. Can be copied using the regular format, where each point is unique in the list and the element list is used to access the nodes of each element, or in the exploded format, where the points are duplicated and copied in element index order. The latter one uses the nodal indices to describe the elements and the nodes are duplicated in the array. In this case, the element list is not copied to the GPU, as the nodes list can be accessed sequentially.

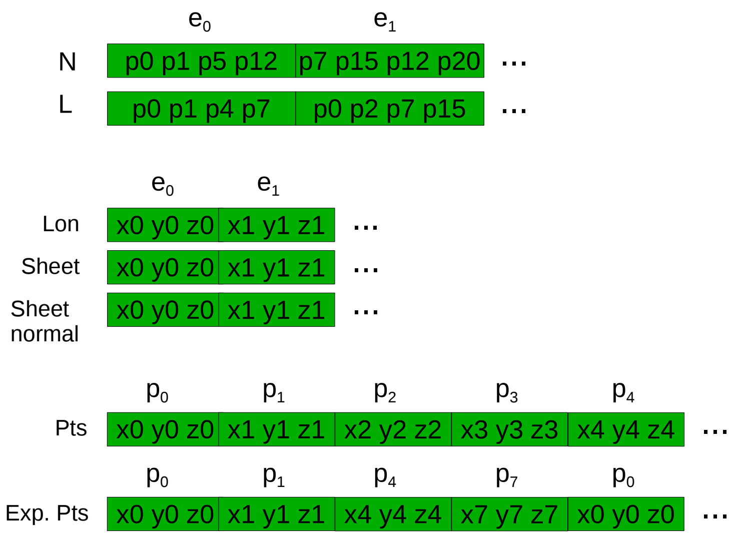

Fig. 3. Data structure organization in the GPU. Elements, axes and nodes lists are structured as linear arrays.

Device routines For each kernel, an interface routine was implemented in C, in which a structure with all kernel parameters is assembled before the kernel is called. The kernel is then called with $(p.nElem + \_N - 1)/\_N$ blocks and $\_N$ threads per block. In this case, the total number of threads might be larger than the number of elements, but only $nElem$ threads execute the calculations.

In the device routines, i.e. the kernels, the start indexes are computed by each thread for each array that is accessed in the routine. In the example below (code 1.3), mStart, lStart and pStart stores the initial index of mPtr, dLon, and pevs, respectively, for each thread. Local variables are used to access the register directly. The calculations are done using explicit expressions to avoid loops in the kernel routine. Again the following listing is representative for all kernel routines needed in the element matrix calculations.

Listing 1.3. Kernel to compute conductivity tensor G

```
__global__ void
__device_fl_getCondTensorGPU(_device_fl_getCondTensorGPU_params p)
{
 int mStart = p.msz*_N*blockIdx.x + (threadIdx.x/_L)*p.msz*_L
              + (threadIdx.x%_L);
 int lStart   = 3*_N*blockIdx.x + (threadIdx.x/_L)*3*_L
              + (threadIdx.x%_L);
 int pStart   = 3*_N*blockIdx.x + 3*threadIdx.x;
 int maxsize  = _N*blockIdx.x + threadIdx.x;

 if(maxsize < p.nElem) {
   double f1 = p.dLon[lStart];
   double f2 = p.dLon[lStart+=_L];
```

```
    double f3 = p.dLon[lStart+=_L];

    double gf = p.pevs[pStart++];
    double gs = p.pevs[pStart];

    // (gf-gs)* f x f + gs * I
    p.mPtr[mStart]      = f1*f1*(gf - gs) + gs;
    p.mPtr[mStart+=_L] = f1*f2*(gf - gs);
    p.mPtr[mStart+=_L] = f1*f3*(gf - gs);
    p.mPtr[mStart+=_L] = f1*f2*(gf - gs);
    p.mPtr[mStart+=_L] = f2*f2*(gf - gs) + gs;
    p.mPtr[mStart+=_L] = f2*f3*(gf - gs);
    p.mPtr[mStart+=_L] = f1*f3*(gf - gs);
    p.mPtr[mStart+=_L] = f2*f3*(gf - gs);
    p.mPtr[mStart+=_L] = f3*f3*(gf - gs) + gs;
  }
}
```

Further optimization The computation of the matrix inverse for the $4-by-4$ (tetrahedra) case exhibited register spilling, which was removed by rearranging computations such that compiler generated temporary expressions have been reused. Additionally we removed the asserting function, used to stop the computations in case the determinant is zero. Memory (de)allocation overhead was identified when computing the stiffness and mass matrix. Thus, the memory (de)allocation calls were moved outside the main loop. Therefore, it is only done once and it is not included in the final assembly times. When using the nodes list with elemental indexing, as described in Sec. 4.3, the array with the points was accessed via texture cache to compensate for the overhead of accessing the nodes in non-sequential order. This approach saves memory, as the points in the nodes list are not duplicated, and computation time is only marginally affected.

4.4 Global assembly

The accumulation of the element matrices stored in a FMatrixArray structure into one global matrices implemented within the parallel toolbox [12, 13]. Therein the element matrix entries will be reordered according to their global row and column indices such that these entries can be accumulated for each global matrix entry in parallel afterwards. Again, the permutation vector for this mapping is determined in an a priori setup. Another approach which would save a lot of temporary memory requires the coloring of the finite elements such that no data races will appear in the accumulation process. This has been applied successfully on vector processors [17] as well as on GPUs [2] general many-core environments [11]. Our own improved version of these parallel matrix accumulations is still ongoing research.

5 Results

We used the following configuration for our experiments. The CPU was an Intel Xeon E5645 with 6 cores (12 threads), clock Speed of 2.4 GHz, 12MB of L2-cache and 24 GB DDR3 memory with 32 GB/sec of memory bandwidth. The

GPU is an NVidia GTX 680 with 1536 CUDA cores, 1006 MHz Base Clock and with 2048 GB GDDR5 memory with 192.2 GB/sec of memory bandwidth. The Performance was measured with one vectorized CPU core and compared to the GPU CUDA code for tetrahedral meshes of different sizes. The .cu files, where the GPU kernels are implemented, are compiled with NVCC using architecture sm_20 and -O3 option. The .c files for the CPU are compiled with GCC, using options -g -O3 -std=gnu99, but when compiling for the GPU, the output file of the kernels and the CUDA libraries must be linked to the resulting output file of the .c files.

The numerical tests have been performed for the bidomain equations (5) and several discretization of the unit cube. The assembling process has been performed 10 times in order to get average run times and all the data needed have been initialized and transferred before the timing started. Table 1 presents the assembly times of the CPU and GPU as well as the related speedups achieved which are also depicted in Fig. 4. It can be concluded that the GPU speedup is approximately 80 for larger numbers of tetrahedrals and up even 90 for the assembling of the simpler mass matrices. If we assume a perfect speedup of 8 when all 8 CPU cores are used then a quite good GPU speedup of 10 still remains. Our algorithms are bandwidth limited and therefore the 8 CPU cores sharing that bandwidth will perform worse. Therefore, even a (good) consumer card as the GTX 680 achieves a significant speedup for matrix calculation and assembling in double precision.

	Time in sec.				Speedup	
	Stiffness		**Mass**		**Stiffness**	**Mass**
n elements	CPU	GPU	CPU	GPU		
12,500	0.007289	0.000216	0.002725	0.000131	33.74	20.80
50,000	0.034600	0.000517	0.013445	0.000239	66.92	56.25
112,500	0.078769	0.001038	0.030778	0.000439	75.88	70.11
450,000	0.312762	0.003712	0.124095	0.001373	84.26	90.38
1,250,000	0.745111	0.010108	0.300238	0.003620	73.71	82.93

Table 1. Assembly time and speedup for two matrices in double precision.

6 Conclusion

The implementation of the FE matrices assembly using vectorized code for the CPU and CUDA for the GPU has proved to be highly efficient, with the CUDA code reaching a maximum speedup of 90. On the other hand, it appears that there is a limit in performance when using CUDA, as the speedup drops when a mesh with more than 450000 elements is used. This might be due to memory bandwidth limitation, as only a limited number of threads can have access to the cache memory at the same time. The work presented is still ongoing.

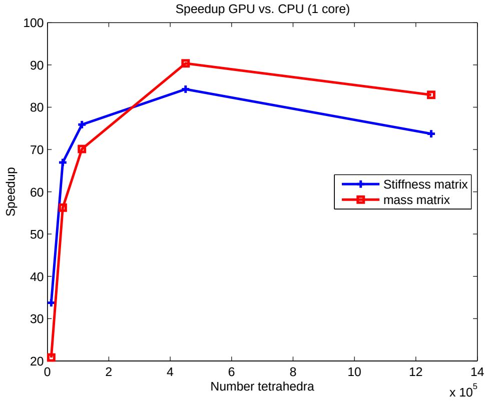

Fig. 4. Speedup of GPU vs. one CPU core for assembling routines in double precision.

Nevertheless, we expect that vectorized code as well as the CUDA codes will increase performance when used within the mechanical model, where the matrices have to be re-assembled at each step. Moreover, the modifications required in the main branch to include the new implementation are minimized by modularity, and only a few routines within FEM and FMatrix have to be modified. The next step will consist in applying the described methodology to mechanical problems and test the bidomain-mechanical problem [15] on NVidia's Tesla 20K with eight time more double precision compute units available. Together with the already available a priori calculation of the matrix patterns and the splitting of the AMG preconditioner setup the whole non-linear iteration in the solution process of the bidomain-mechanical problem should run on the GPU.

Acknowledgment

This work has been partially supported by Austrian Science Fund (FWF): F 3201-N18.

References

1. N. Bell, S. Dalton, and L. N. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM J. Sci. Comput.*, 34(2):C123–C152, 2012.
2. C. Cecka, A. J. Lew, and E. Darve. Assembly of finite element methods on graphics processors. *Int. J. for Numerical Methods in Engineering*, 85(5):640–669, 2011.
3. M. Geveler, D. Ribbrock, D. Göddeke, P. Zajac, and S. Turek. Towards a complete FEM-based simulation toolkit on GPUs: Unstructured grid finite element geometric multigrid solvers with strong smoothers based on sparse approximate inverses. *Computers & Fluids*, 80:327–332, July 2013.

4. M. S. Gockenbach. *Understanding and Implementing the Finite Element Method.* SIAM, Philadelphia, 2007.
5. D. Göddeke. *Fast and Accurate Finite-Element Multigrid Solvers for PDE Simulations on GPU Clusters.* PhD thesis, Technische Universität Dortmund, Fakultät für Mathematik, http://hdl.handle.net/2003/27243, May 2010.
6. D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. S. McCormick, H. Wobker, C. Becker, and S. Turek. Using GPUs to improve multigrid solver performance on a cluster. *International Journal of Computational Science and Engineering*, 4(1):36–55, Nov. 2008.
7. G. Haase, M. Liebmann, C. C. Douglas, and G. Plank. A parallel algebraic multigrid solver on graphics processing units. In W. Zhang, Z. Chen, C. C. Douglas, and W. Tong, editors, *HPCA (China), Revised Selected Papers*, volume 5938 of *Lecture Notes in Computer Science*, pages 38–47. Springer, 2009.
8. K. Jónasson, editor. *Applied Parallel and Scientific Computing - PARA 2010, Part II*, volume 7134 of *Lecture Notes in Computer Science.* Springer, 2012.
9. M. Jung and U. Langer. *Methode der finiten Elemente für Ingenieure.* Lehrbuch. Springer Vieweg, Wiesbaden, 2nd edition, 2013.
10. M. G. Larson and F. Bengzon. *The Finite Element Method: Theory, Implementations and Applications*, volume 10 of *Texts in Computational Science and Engineering.* Springer, Berlin, Heidelberg, 1st edition, 2013.
11. G. R. Markall, A. Slemmer, D. A. Ham, P. H. J. Kelly, C. D. Cantwell, and S. J. Sherwin. Finite element assembly strategies on multi-core and many-core architectures. *Int. J. for Numerical Methods in Fluids*, 71(1):80–97, 2013.
12. A. Neic, M. Liebmann, G. Haase, and G. Plank. Algebraic multigrid solvers on clusters of CPUs and GPUs. In Jónasson [8], pages 389–398.
13. A. Neic, M. Liebmann, E. Hötzl, L. Mitchell, E. Vigmond, G. Haase, and G. Plank. Accelerating cardiac bidomain simulations using graphics processing units. *IEEE Transactions on Biomedical Engineering*, 59(8):2281–2290, 2012.
14. NVIDIA Corporation. CUDA programming guide 5.0, 2012. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
15. P. Pathmanathan and J. P. Whiteley. A numerical method for cardiac mechano-electric simulations. *Ann Biomed Eng*, 37(5):860–73, 2009.
16. B. Rocha, F. Campos, G. Plank, R. Weber dos Santos, M. Liebmann, and G. Haase. Simulations of the electrical activity in the heart with graphic processing units. *Concurrency Computat.: Pract. Exper.*, 23:708–720, 2011.
17. F. T. Tracy. Optimizing finite element programs on the cray x1 using coloring schemes. In *Proceedings of the 2004 Users Group Conference*, DOD_UGC '04, pages 329–333, Washington, DC, USA, 2004. IEEE Computer Society.
18. E. Vigmond, M. Hughes, G. Plank, and L. Leon. Computational tools for modeling electrical activity in cardiac tissue. *J Electrocardiol*, 36:69–74, 2003.
19. E. Vigmond and G. Plank. http://carp.meduni-graz.at. Online, 2009.