High locality and increased intra-node parallelism for solving finite element models on GPUs by novel element-by-element implementation

Imre Kiss
Budapest University of
Technology and Economics
H-1521 Budapest Hungary
Email: kiss@evt.bme.hu

Zsolt Badics, *Senior Member, IEEE*
Tensor Research, LLC
Andover, MA 01810, U.S.A
Email: badics@ieee.org

Szabolcs Gyimóthy and József Pávó
Budapest University of
Technology and Economics
H-1521 Budapest Hungary
Email: gyimothy,pavo@evt.bme.hu

***Abstract*—The utilization of Graphical Processing Units (GPUs) for the element-by-element (EbE) finite element method (FEM) is demonstrated. EbE FEM is a long known technique, by which a conjugate gradient (CG) type iterative solution scheme can be entirely decomposed into computations on the element level, i.e., without assembling the global system matrix. In our implementation, NVIDIA's parallel computing solution, the Compute Unified Device Architecture (CUDA), is used to perform the required element-wise computations in parallel. Since element matrices need not be stored, the memory requirement can be kept extremely low. It is shown that this low-storage but computation-intensive technique is better suited for GPUs than those requiring the massive manipulation of large data sets. This study of the proposed parallel model illustrates a highly improved locality and minimization of data movement, which could also significantly reduce energy consumption in other heterogeneous HPC architectures.**

***Index Terms*—CUDA Computing, EbE FEM, GPU Computing, parallel FEM**

I. Introduction

A. A Novel HPC Finite Element Environment

Our goal is to illustrate the efficiency of an application-centric parallel model for the solution of a wide range of partial differential equations (PDEs) discretized using regular and irregular meshes by the finite element (FE) method. Such a parallel model covers a significant application segment of scientific computing problems. Examples are numerical simulations of electromagnetics, structural mechanics, thermal, and related multiphysics and multi-scale problems. We demonstrate here the performance of the finite element parallel model through solving an ECG (electrocardiography) forward problem in the human torso[1].

The parallel model exhibits highly improved locality and minimization of data movement compared to other FE implementations on GPUs. The high locality and the increased intra-node parallelism could also significantly reduce energy consumption on other heterogeneous HPC architectures. Our acceleration technique based on an element-by-element (EbE) FE framework [2] provides a factor of one hundred speed improvement for the ECG forward problem compared to top-notch CPU implementations for a discretization with millions of tetrahedrons. Here we summarize the major elements of the parallel model and emphasize the high locality and the increased intra-node parallelism.

B. Element-by-element FEM method

Nowadays the finite element method is one of the most frequently used techniques for engineering analysis of complex, real-life applications of both linear and non-linear types [3]. Let us consider the linear equations system of the form

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1}$$

resulting from the FEM approximation of a partial differential equation. $\mathbf{A}$ is the system matrix, $\mathbf{x}$ is the vector of unknowns, and $\mathbf{b}$, the right hand side (RHS), represents the excitation. For large scale problems the solution of (1) is usually obtained by iterative solvers. To keep generality – not taking advantage of any special property of the system matrix $\mathbf{A}$, except sparsity – a preconditioned bi-conjugate gradient (BiCG) solver is used in this paper [4].

The element-by-element type finite element method (EbE FEM) was constructed originally for low memory computers [5]. The foundation of the method is based on the recognition that the assembling of element matrices to form the global system matrix is a linear operation. Therefore certain calculations with the system matrix –like e.g. a matrix-vector product– can be traced back to the level of finite elements, thus converted to calculations with the individual element matrices $\mathbf{A}_e$, which appear in the elementary equations having the form

$$\mathbf{A}_e\mathbf{x}_e = \mathbf{b}_e \tag{2}$$

Most iterative solvers can be decomposed into a sequence of matrix-vector products and inner products of vectors so they are suitable to be implemented in the EbE context. The idea comes naturally: do not assemble and store the element matrices into the system matrix as traditional methods do, rather recompute them in each iteration. This is possible because iterative solvers (in contrast to direct solvers) do not affect the system matrix during the solution.

The technique can be thought as one which transforms a highly memory dependent problem to a massively computational dependent one, which in turn can be efficiently parallelized [6]. Platforms having massively parallel computing capability (like today's GPUs) can take full advantage of this method.

C. Parallel FEM implementations on GPUs

Although several methods partially accelerating the FEM computations have already been implemented on GPUs [7], [8], [9], these usually suffer from a strict design limitation. Namely, these devices can operate only on data that is stored in their on-board memory. In other words, data must be transferred from the system memory to the device's memory prior to any computation.

Large scale FEM problems need large storage capacity for the global system matrix. Consequently, efforts to accelerate only the calculation of the matrix-vector product may conflict with a substantial property of GPUs: the available memory capacity is limited (in a few GBs).

To overcome this problem one may consider different *domain decomposition* techniques. The decomposition can be conceptualized either in its traditional meaning [10] or one just decomposes the large system matrix to smaller parts that fit into the memory. These sub-matrices are then transferred to the GPU one after the other, and the partial multiplications are performed in parallel.

A major disadvantage of this technique is that although the partial products are performed significantly (by several order) faster this way, the necessary bus transfers will cause the overall acceleration to be much more moderate. This disadvantage becomes even more remarkable when multiple GPUs are present. In this case the computing capacity is drastically increased compared to the single GPU case but the relative throughput of the data transfers is bounded by the simultaneous needs from the devices.

To achieve better utilization, a remedy needs to be found to avoid costly data transfers. Such a technique would be one which acts entirely on the GPUs.

D. Aim of the work

As the gap between bus speed and computation density increases, codes which use the accelerator design (i.e. in which only the computation intensive parts of the program are executed on the GPU) will fall behind codes that take full advantage of it (i.e. perform all the necessary computations on the GPU) [11].

The aim of this paper is to show that modern high performance computing platforms (GPUs) offer considerable computing capacity that can be fully utilized only if the applied algorithm fits to their specific architecture. This property is in contrast to traditional (multi-) CPU based program design patterns where the efficiency of an algorithm is simply estimated using its computational complexity.

Relying on the fact that it is cheaper to recompute element matrices than continuously cache them between the device and the system memory, the EbE FEM technique is revisited here, and its implementation on CUDA architecture is presented. It is also demonstrated that EbE FEM highly extends the scale of problems that can be solved on devices having limited memory capacity but a massively parallel architecture.

II. Implementation of EbE FEM on CUDA

A. Disassembling matrix manipulations to the element level

The finite element assembling procedure relies on some functions by which the element matrix $\mathbf{A}_e$ and the RHS $\mathbf{b}_e$ of (2) are computed. These functions depend among others on the type of PDE to be solved as well as the applied shape functions. The computed element matrices and RHS vectors are then assembled to form the global system matrix $\mathbf{A}$ and RHS vector $\mathbf{b}$. Let this assembly step be represented by an operator $\mathcal{M}$, which is defined differently for matrices and vectors:

$$\mathbf{A} = \mathcal{M}(\mathbf{A}_e) = \sum_{e\in\mathcal{E}} \mathbf{C}_e^T \mathbf{A}_e \mathbf{C}_e \tag{3}$$

$$\mathbf{b} = \mathcal{M}(\mathbf{b}_e) = \sum_{e\in\mathcal{E}} \mathbf{C}_e \mathbf{b}_e \tag{4}$$

where $\mathcal{E}$ is the set of elements, and matrix $\mathbf{C_e}$ represents the transition between *local* and *global* numbering of the unknown variables for the e-th element. Contrary to the sparse global system matrix $\mathbf{A}$, the matrix $\mathbf{A_e}$ of size $n_e \times n_e$ (n_e being the local degrees-of-freedom) is usually dense.

Using the above concept, the matrix-vector product, which is the basis of iterative solvers, can be reformulated in terms of element-wise computations as

$$\mathbf{A}\mathbf{x} = \sum_{e\in\mathcal{E}} \mathbf{C}_e^T \mathbf{A}_e \mathbf{C}_e \mathbf{x} = \sum_{e\in\mathcal{E}} \mathbf{C}_e^T \mathbf{A}_e \mathbf{x}_e = \mathcal{M}(\mathbf{A}_e \mathbf{x}_e). \tag{5}$$

It follows that the product of an assembled global matrix and a vector is equivalent with the assembled vector of the elementary matrix-vector products. According to (4) the elementary contributions can be accumulated in a vector having the size of the global degrees-of-freedom (DoF). Hence only vectors have to be stored during the computations. Elementary matrix-vector products in (3) can be computed for each element separately, which enables parallel realization [6].

The inner product of two DoF-sized vectors is the other basic operation for iterative solvers. This operation is obviously independent of the mesh structure and connectivity, and its parallel execution is straightforward.

A further advantage of the EbE implementation is that no global numbering of unknowns and finite elements is required. Due to the lack of assembly, there is no need for an optimized global numbering to obtain a low bandwidth system matrix. If one uses some mesh refinement/reduction technique during the iterations, locality for the necessary modifications is also ensured [12].

B. Concurrency – global updates and coloring

On shared memory architectures (like the GPUs) an important question is how the partial products are summarized. The challenge during a global update is to ensure that different threads do not access the same memory space simultaneously.

This concurrent access is called *race-condition* and results in an indefinite outcome. Treatment of such cases is traditionally of two kinds. One solution is "atomic" updates, when the memory space is protected during I/O, causing other threads accessing the same memory place to wait until the operation is fully completed.

The other solution is "coloring" [13], [14]. In this case the mesh is considered as a graph, with the unknown variables (DoFs) being the nodes of the graph and the elements representing the connections between them. This graph is then colored in a way that any two elements having the same color do not share a common unknown. Different colors are then processed "serially" (one after the other), while elements with the same color are processed simultaneously in parallel.

C. Element-by-Element formulation of the BiCG solver

In a BiCG solver based on an EbE implementation the computations can be grouped into so-called "EbE" steps and "DoF" steps, respectively. The former refers to the matrix-vector product of (5), while the latter means vector-vector product or initialization of variables. The BiCG algorithm requires a few auxiliary vectors ($\mathbf{r}, \tilde{\mathbf{r}}, \mathbf{d}, \tilde{\mathbf{d}}, \mathbf{q}, \tilde{\mathbf{q}}$) and complex variables ($\delta, \tilde{\delta}, \alpha$) during the iterations. Function of these variables is identical to that outlined in [4, Chapter 2.3.5].

Elements of these vectors, similarly to elements of the solution vector, $\mathbf{u}$, are stored as global unknowns. A global unknown can therefore be thought as an object which contains all the (self) associated elements of the different working vectors.

The way the variables are stored gives the real modularity of the EbE method. Contrary to traditional FEM methods using global numbering, here a dynamic storage structure is used instead. The structure can be thought as an index array (pointers in the actual implementation) keeping the information how local unknowns correspond to global ones. (Such storage pattern is often referred as "spatial data structure" in the literature.) This is functionally equivalent to the role of $\mathbf{C}_e$ in (3) - (4).

Algorithm 1 shows the EbE based BiCG implementation, which is functionally equivalent to the one presented in [4, Chapter 2.3.5] and is implemented in terms of "EbE" and "DoF" iterations. Label "EbE iteration" indicates the computation of the element matrices. To avoid race conditions during global updates, the elements are colored. The iteration goes through all colors serially, and performs the computations on the elements having the actual color, $\mathcal{E}^{(c)}$, in parallel. Label "DoF iteration" indicates the computation of the vector-vector products. This iteration is performed simultaneously on all $\mathcal{U}$ global unknowns. Label "global update" means that the value of a global variable is affected. To avoid race conditions, atomic updates are used to access global variables.

D. Some drawbacks of the EbE implementation

The lack of assembling makes the method convenient for GPU parallel execution but it also comes with some difficulties. The first one is related to preconditioning, which traditionally assumes the system matrix to be in an assembled form. To overcome this problem one can use specific element-by-element preconditioners [14], [15].

In this paper a simple Jacobi preconditioner is used [4]

Algorithm 1: Element-by-Element BiCG algorithm

```
INIT  foreach u ∈ 𝒰 do                                   // DoF iteration
          r^(u) ← 0
          D^(u) ← 0
      for c = 1 to |colors| do                           // serial loop
          foreach e ∈ ℰ^(c) do                           // EbE iteration
              r_e ← r_e + b_e − A_e u_e
              D_e ← D_e + diag(A_e)
      foreach u ∈ 𝒰 do                                   // DoF iteration
          D^(u) ← 1/D^(u)
          r̃^(u) ← r^(u)
          d^(u) ← D^(u) · r^(u)
          d̃^(u) ← D^(u) · r̃^(u)
          q^(u) ← 0
          q̃^(u) ← 0
      δ ← 0                                              // global update
      foreach u ∈ 𝒰 do                                   // DoF iteration
          δ ← δ + r^(u) d^(u)                            // global update
LOOP  while δ > prescribed accuracy do                   // host loop
          for c = 1 to |colors| do                       // serial loop
              foreach e ∈ ℰ^(c) do                       // EbE iteration
                  q_e ← q_e + A_e d_e
                  q̃_e ← q̃_e + A_e^T d̃_e
          α ← 0                                          // global update
          foreach u ∈ 𝒰 do                               // DoF iteration
              α ← α + d̃^(u) · q^(u)                      // global update
          foreach u ∈ 𝒰 do                               // DoF iteration
              x^(u) ← x^(u) + δ/α · d^(u)
              r^(u) ← r^(u) − δ/α · q^(u)
              r̃^(u) ← r̃^(u) − δ/α · q̃^(u)
          δ̃ ← δ; δ ← 0                                   // global update
          foreach u ∈ 𝒰 do                               // DoF iteration
              δ ← δ + r^(u) D^(u) r̃^(u)                  // global update
          α ← δ/δ̃                                        // global update
          foreach u ∈ 𝒰 do                               // DoF iteration
              d^(u) ← D^(u) · r^(u) + α · d^(u)
              d̃^(u) ← D^(u) · r̃^(u) + α · d̃^(u)
              q^(u) ← 0
              q̃^(u) ← 0
```

because it can be represented by a diagonal matrix ($\mathbf{D}$), which can be stored the same way as the DoF-sized auxiliary vectors. The Jacobi preconditioner is implemented as a "DoF" step (see Algorithm 1, line 10-11, 31 and 34-35).

The second problem is related to the required "extra" computations: since element matrices are not stored, they must be recomputed in each iteration, which is obviously redundant when dealing with linear problems. However, this extra computation becomes necessary for non-linear or coupled problems [12].

E. Details of CUDA implementation

The EbE BiCG method is implemented to run exclusively on the GPUs. After the mesh is read in and its appropriate "coloring" is determined, point positions and triangulation information is moved to the GPU memory. Both data are stored in special, aligned data types to ensure coalesced access during the operations [16]. All computations are carried out using double precision floating point representation.

Each EbE and DoF iteration is performed as a separate kernel call including global updates. When only one GPU is used, there is no need for synchronization during the kernel calls. In an EbE iteration, a kernel call has as many allocated threads as the number of elements having the current color. Element sets of different colors are handled one after the other by calling the same kernel, embedded in a host side iteration through all the colors. Since in a DoF iteration all computations can be carried out simultaneously, kernels have as many threads as global unknowns. Global updates in DoF iterations are carried out using kernels with internal block summations.

Following the initialization part, the BiCG loops (c.f. "INIT" and "LOOP" labels in Algorithm 1) are computed as numerous kernel calls embedded in a host loop. At the end of this loop, the global variable δ is transferred to the host's memory, and the termination of the loop is decided on the host side. Since no other host side computation is carried out during looping, the CPU load of the algorithm is negligible.

III. Results

To study the accuracy and speed of the proposed method, the "Utah Torso" model was investigated by solving an ECG forward problem [1] (c.f. Fig. 1). This test problem is a static conduction problem with inhomogeneous conductivity. The equation to be solved is therefore the Laplace equation with spatially varying conductivity

$$\nabla \cdot \sigma \nabla \varphi = 0. \tag{6}$$

The domain is discretized by tetrahedral elements and linear nodal shape functions are used. The element matrices are computed using analytical expressions [17]. The global unknowns (DoFs) referred in Section II-C are the potential values φ at the nodes of the mesh.

The run time statistics obtained for several different mesh sizes are shown in Table I . (The target residual error is set to 10^{-8} for both the GPU and CPU implementations.)

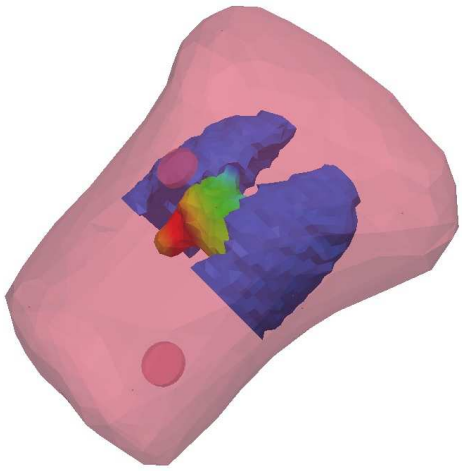

Figure 1. Utah torso model with major organs and electrodes

The computations have been carried out on a HP-XW8600 workstation, having 32 GB memory, an NVIDIA GTX 590 GPU card (with two GPU processors on a single board) and a quad-core Intel Xeon X3440 CPU. The fully parallelized CPU implementation of the preconditioned BiCG method is based on the Intel® Math Kernel Library (MKL). Figure 2 shows the runtime comparison for the different implementations.

Table I
RESULTS, UTAH TORSO PROBLEM

Mesh	**#1**	**#2**	**#3**	**#4**
No. of tetrahedra	$560K$	$6,559K$	$18,884K$	$29,772K$
No. of unknowns	$91K$	$1,339K$	$3,836K$	$6,064K$
GPU implementation				
No. of iterations	322	671	978	1111
No. of colors	41	45	53	56
Memory [MByte]	12	213	613	968
Runtime (1 GPU) [s]	2.8	40.3	171.3	303.8
Runtime (2 GPUs) [s]	1.7	23.6	93.4	167.6
CPU implementation				
No. of iterations	79	248	338	391
Memory [MByte]	1,564	5,251	13,645	19,371
Runtime [s]	5.9	403.4	1,166.6	1,694.6

As also outlined in [9], the performance of the GPU accelerated matrix-vector multiplication (MxV) highly depends on the structure of the matrix, i.e., the distribution and number of non-zero elements. Unlike methods using GPUs only for accelerating the computation of the MxV, the proposed method does not rely on the system matrix, hence no such degradation may occur. This results in a uniform performance irrespectively of the matrix structure.

Another advantage is the memory efficiency. Since sparse matrix storage inherently requires some extra storage overhead (for the row and column information of non-zero elements), the efficiency of memory occupancy is limited. On the contrary, the proposed EbE FEM only needs to store the meshing information and several DoF sized auxiliary vectors required for the CG iterations (see Section II-C).

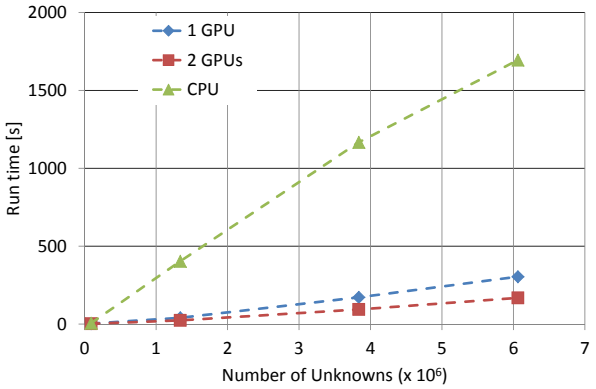

Figure 2. Runtime comparison of GPU and CPU implementations

IV. Conclusion

We re-formulated the EbE FEM method to fit to GPU architectures. The method has extremely low memory consumption and can take full advantage of the massively parallel execution environment due to the applied EbE-FEM formulation.

The main cause of the outstanding performance of such a method is that it is very well suited for the special multi-core GPU environment where cores are independent and don't easily communicate with each other. Since the EbE-FEM requires no communication on the level of processing cores (that is, it is highly localized) the hardware utilization can be maximized. Another important property of the method is that it does not rely on the traditional "assemble-and-solve" computational structure because the entire system matrix is never assembled. This results in an excellent memory utilization pattern.

Not only the algorithm outperforms traditional CUDA accelerated FEM methods [7], [8], [9] but it is also competitive with today's CPU based algorithms. Table I shows some run time data for the same meshes generated by a FEM conduction solver using a preconditioned BiCG implementation based on the Intel® Math Kernel Library. Another advantage over the popular GPU FEM implementations is that the treatable problem size is limited by the amount of meshing information rather than by the number of non-zero elements in the system matrix.

Although this paper only outlined the application of the method on a single GPU, multi-GPU computations were also performed and preliminary results have been already acquired for a simple 2 GPU system. (See in Table I.) These preliminary multi-GPU results indicate good scalability of the method, which could easily be extended to GPU clusters. Utilization of further processing capacity revealed that the expected linear scaling could be achieved. This topic will be covered in a forthcoming paper.

Although the presented solution of an ECG problem was notably accelerated, the utilization of EbE FEM can be appreciated even more when treating non-linear problems or when the mesh structure changes during the computation (that is, when adaptive mesh refinement is applied). In non-linear FE problems the system matrix must be recomputed in every iteration step. Thus, due to the fact that the EbE method recomputes the local element matrix in each iteration step, the achievable speed-up could be even more significant. In adaptive mesh generation the system matrix as well as the unknowns are changing during the solution steps. Since the EbE technique stores the variables locally it is straightforward that adaptive mesh refinement is easier to perform in this environment.

Finally, it should be noted that all these benefits also make the element-by-element method very favorable for other heterogeneous HPC environments where high locality is a key concern.

References

[1] R. MacLeod, C. Johnson, and P. Ershler, "Construction of an inhomogeneous model of the human torso for use in computational ECG," in *IEEE Medicine and Biology Society*. IEEE Press, 1991, pp. 688–689.

[2] I. Kiss, S. Gyimóthy, Z. Badics, and J. Pávó, "Parallel realization of the element-by-element FEM technique by CUDA," *IEEE Trans. on Magnetics*, vol. 48(2), pp. 507–510, 2012.

[3] P. P. Silvester and R. L. Ferrari, *Finite Elements for Electrical Engineers*. Cambridge University Press, 1990.

[4] R. Barrett, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Ed.* Philadelphia, PA: SIAM, 1994.

[5] G. F. Carey, E. Barragy, R. McLay, and M. Sharma, "Element-by-element vector and parallel computations," *Commun. Appl. Numer. Methods*, vol. 4, no. 3, pp. 299–307, 1988.

[6] G. F. Carey and B.-N. Jiang, "Element-by-element linear and nonlinear solution schemes," *Appl. Num. Meth.*, vol. 2 (2), pp. 145–153, 1986.

[7] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, pp. 917–924, July 2003.

[8] C. Cecka, A. Lew, and E. Darve, "Introduction to assembly of finite element methods on graphics processors," *IOP Conference Series: Materials Science and Engineering*, vol. 10, no. 1, p. 012009, 2010.

[9] A. Cevahir, A. Nukada, and S. Matsuoka, "Fast conjugate gradients with multiple GPUs," in *ICCS 2009*, G. G. van Albada, J. Dongarra, and P. Sloot, Eds., 2009, vol. 5544, pp. 893–903.

[10] R. K. W. Hackbusch, B. N. Khoromskij, "Direct schur complement method by domain decomposition based on H-matrix approximation," *Computing and Visualization in Science*, vol. 8, pp. 179–188, Dec 2005.

[11] I. Kiss, S. Gyimóthy, and J. Pávó, "Acceleration of moment method using CUDA," *The International Journal for Computation and Mathematics in Electrical Engineering (COMPEL)*, vol. 31(6), IN PRESS.

[12] S. Gyimóthy and I. Sebestyén, "Symbolic description of field calculation problems," *IEEE Tran. Mag.*, vol. 34, no. 5, pp. 3427 –3430, 1998.

[13] C. Farhat and L. Crivelli, "A general approach to nonlinear FE computations on shared-memory multiprocessors," *Computer Methods in Applied Mechanics and Engineering*, vol. 72, no. 2, pp. 153–171, Feb. 1989.

[14] A. J. Wathen, "An analysis of some element-by-element techniques," *Computer Methods in Applied Mechanics and Engineering*, vol. 74, no. 3, pp. 271–287, Sep. 1989.

[15] G. Golub and Q. Ye, "Inexact preconditioned conjugate gradient method with inner-outer iterations," *SIAM J. on Scientific Computing*, vol. 21(4), pp. 1305–1320, 2000.

[16] D. Kirk and W.-M. Hwu, *Programming Massively Parallel Processors, A Hands-on Approach*. Morgan Kaufmann, 2010.

[17] A. Nentchev, "Numerical analysis and simulation in microelectronics by vector finite elements," Ph.D. dissertation, Tech. Univ. Wien, 2008.