

# GPU Acceleration of Equations Assembly in Finite Elements Method – Preliminary Results

Jiří Filipovič  
Masaryk University  
fila@ics.muni.cz

Igor Peterlík  
Masaryk University  
peterlik@ics.muni.cz

Jan Fousek  
Masaryk University  
izaak@mail.muni.cz

**Abstract**—The finite element method (FEM) is widely used for numerical solution of partial differential equations. Two computationally expensive tasks have to be performed in FEM – equations assembly and solution of the system of equations.

We present mapping of the equations assembly problem for StVenant-Kirchhoff material to GPU computation model and show results of its early implementations outperforming our single core CPU implementation by factor of 15. Moreover, presented method used for equations assembly problem solution is more general and can be considered as a technique to manage complex composition of medium-grained operations with different requirements of GPU resources.

## I. EQUATIONS ASSEMBLY

In this paper, CUDA-based acceleration of equations assembly in static deformation FE model is presented. The model coupling the applied loads  $f$  and resulting deformation  $u$  is based on the theory of elasticity. Beside the static equilibrium relating the internal *stress tensor*  $\sigma$  with the applied loads  $f$ , the model is given by the formulation of strain tensor. In our case, non-linear *Green strain tensor*  $\gamma$  is used:

$$\gamma_{ij} = \frac{1}{2}(\partial_j u_i + \partial_i u_j + \partial_i u_m \partial_j u_m) \quad (1)$$

where  $\partial_i$  denotes derivative of the argument w.r.t. the  $i$ -th component.

In our case, hyperelastic model is considered given by a *stored energy function*  $W$  such that

$$\sigma_{ij} = \frac{\partial W}{\partial \gamma_{ij}}. \quad (2)$$

For the StVenant-Kirchhoff material, the stored energy  $W$  is defined as

$$W = \frac{\lambda}{2} \gamma_{ii} \gamma_{jj} + \mu \gamma_{ij} \gamma_{ji} \quad (3)$$

where  $\lambda$  and  $\mu$  are material coefficients known as *Lamé constants*.

The resulting boundary value problem (BVP) is given by a governing equation which is built from the equations given above defined over a domain  $\Omega$  and boundary conditions. The BVP cannot be solved analytically due to the non-linearity of the governing equation and irregularity of the domain. Therefore, *finite element method* is used to reformulate the problem using weak formulation over the domain which is discretized into simple-shaped elements such as cubes and tetrahedra. More precise formulation of FEM is beyond the scope of this paper (see e.g. [1]). The final outcome of the

method is represented by a non-linear system  $\mathbf{K}(\mathbf{u}) = \mathbf{f}$  of algebraic equations, which is usually solved by some iterative method such as Newton-Raphson, where in each iteration, a linearized problem  $\mathbf{A}(\mathbf{u})\Delta\mathbf{u} = \mathbf{f} - \mathbf{K}(\mathbf{u})$  is solved using some actual estimation of  $\mathbf{u}$ . In this paper, the procedure assembling the matrices  $\mathbf{A}(\mathbf{u})$  and  $\mathbf{K}(\mathbf{u})$  is studied. It is supposed that the domain is discretized by linear tetrahedral elements having four vertices each. During the assembly procedure, small  $12 \times 12$  matrices  $\mathbf{A}^e(\mathbf{u})$  and  $\mathbf{K}^e(\mathbf{u})$  are assembled for each element  $e$ .

Denoting  $\phi$  the linear shape functions provided by the finite element method and  $\delta_{ij}$  the Kronecker  $\delta$ -symbol, the matrices  $\mathbf{A}^e(\mathbf{u})$  and  $\mathbf{K}^e(\mathbf{u})$  are defined by components using the tensor notation with summation convention as follows:

$$A_x^N = \int_{\Omega^e} \frac{\partial W}{\partial \gamma_{ij}} (\delta_{xi} + \partial_i u_x) \partial_N \phi_j dV - \int_{\Omega^e} f_x \phi_N dV - \int_{\partial\Omega^e} g_x \phi_N dS \quad (4)$$

$$K_{xy}^{NM} = \int_{\Omega^e} \left\{ \frac{\partial^2 W}{\partial \gamma_{ij} \partial \gamma_{kl}} [(\delta_{yi} + \partial_i u_y) \partial_j \phi_N] [(\delta_{xk} + \partial_k u_x) \partial_l \phi_M] + \delta_{xy} \frac{\partial W}{\partial \gamma_{ij}} \partial_i \phi_N \partial_j \phi_M \right\} dx. \quad (5)$$

where the indices  $M, N = 1 \dots 4$  run over the vertices of the element and  $x, y = 1 \dots 3$  are the components of the 3D problem. The integration over the volume  $\Omega$  or its boundary  $\partial\Omega$  is performed numerically using Gauss quadrature.

Once the system of equations is assembled, it has to be solved. We focus on equations assembly problem only, as solving of both sparse [2] and dense [3] systems of equations has been successfully accelerated using GPUs. The additional advantage of performing equations assembly in GPU is that the assembled system is already in GPU's memory eliminating the PCI-E overhead.

## II. PROBLEM MAPPING TO GPU COMPUTATION MODEL

In this chapter, the mapping of equations assembly problem to GPU computation model is discussed. It is out of the scope of this paper to describe CUDA-enabled GPU programming and memory model, therefore we refer to NVIDIA CUDA Programming Guide [4].

The system of the equations solved in FEM is composed from results of equations assembly for all elements of the mesh. The amount of computations needed for the equations assembly for one element is insufficient to utilize GPU well,

thus we perform one operation for multiple elements in parallel and reach sufficient GPU utilization for FEM models consisting of thousands elements or more.

The equations assembly is too complex problem to be implemented in one block of code. Moreover, any changes in a monolithic implementation can be difficult. Thus, we decompose equations assembly problem into kernels solving particular algebraic operations and develop each operation separately<sup>1</sup>. After that, we can compose operations to solve specific problem, such as equations assembly for St.Venant material.

The majority of tasks that has been successfully accelerated by GPUs so far are usually decomposed to threads in two ways:

- coarse-grained parts are solved by thread blocks, where each block is decomposed to fine-grained operations solved by particular threads (e.g. blocking in matrix multiplication [3])
- fine-grained operations are solved per thread without any significant role of thread blocks (e.g. Coulomb potential map computation [5]).

In the equations assembly problem, many medium-grained operations are performed. These operations do not fit into any of the models previously mentioned – solving each operation by thread block yields insufficient warp utilization (operations are too fine-grained to be decomposed to sufficient number of threads), on the other hand operations are too coarse-grained to be solved each by one thread yielding insufficient multiprocessor utilization because of their memory needs. Thus, the two-level parallelism within the thread block is utilized – each operation is performed by a few threads and a few operations are performed in parallel within one thread blocks. This approach yields good GPU utilization, because enough threads fit on particular multiprocessors.

### III. ALGEBRAIC OPERATIONS IMPLEMENTATION

All operations are implemented to allow both launch from host code using data in global memory and launch from kernel using data in shared memory, allowing its further composition. Because of poor flop-to-world ratio of used operations, some of them are bounded by global memory bandwidth even without optimal usage of shared memory. We tried to develop all operations to reach good performance using data in both global memory and shared memory.

Many operations used in the equations assembly are medium-grained. Let's consider the multiplication of  $3 \times 3 \times 3$  tensor by  $3 \times 3$  matrix with contraction in one dimension (resulting is  $3 \times 3 \times 3$  tensor). This operation reads 36 floats as input and produces 27 floats as a result. Keeping all of them in shared memory, only 65 operations can be performed in multiprocessor. Performing each operation by one thread yields poor multiprocessor occupancy. The maximal reasonable number of threads solving one operation is 27 (where

<sup>1</sup>In the following text, we use the term *algebraic operation* or *operation* denoting the high level algebraic operation on data structures such as tensors, matrices, vectors and scalars, which are used in equation 5.

each thread produces one scalar number of the result tensor), thus solving one operation in one block yields insufficient warp utilization. Moreover, when data are readed from global memory, the bandwidth is reduced due to noncoalesced access in the case of small blocks. The 2-level parallelism within the block can be used to overcome this limitation, e.g. the tensor matrix multiplication mentioned above can be solved by 27 or 9 threads and one thread block can solve a few multiplications.

In some cases, the managing of shared memory is difficult or some bank conflicts are unavoidable because of small size of data processed by the operations, e.g. threads solving one operation can synchronously read the same value, which cannot be broadcasted, because another threads within the same warp solving another operation synchronously reads another value. In some cases a nonstandard data representation (e.g. permuted dimensions or interleaved storage of multiple objects) can be used to avoid memory bank conflicts, but this optimization affects multiple operations (both producers and consumers of data) and therefore reduces general versatility of implemented operations.

### IV. THE COMPOSITION OF ALGEBRAIC OPERATIONS

Basically there are two ways how to compose operations. They can be executed in serial from host code, each operation reading and writing data from/to global memory. The implementation of this model is simpler, but its performance can be limited by global memory bandwidth. On the other hand, we can compose multiple operations into one kernel allowing data exchange through fast shared memory. Especially when operation with poor flop-to-world ratio are used, this model yields significantly better results, but the problem of composing algebraic operations is more complex.

There are two kinds of problems in transferring data via shared memory. First, the grouping of operations communicating via shared memory has to be optimized. Second, the optimal shared memory usage has to be found minimizing memory requirements by rearranging independent operations and efficient shared memory reuse by data structures. This problems are addressed in the following text.

The number of possibilities how to compose algebraic operations with allowed data transfer via shared memory is large. When data and thread requirements of operations differ, it is not clear when all operations can be composed into one block transferring data via shared memory between all operations and when the split of operations to two or more compositions communicating via global memory is more efficient. The reason why splitting operations into more compositions can yield better performance is underutilizing GPU when operations employing fewer threads are performed within thread blocks that have size for operations using more threads. Moreover, some operations can be performed in the loop of fixed size and can be run in sequential or parallel. If they are executed in parallel within thread block, the parallelism of sequential parts is reduced (because of thread count reduction by larger memory requirements of parallelized operations in the loop), but operations in the loop can utilize multiprocessor better.

These aspects should be balanced very carefully to keep the overall GPU utilization as high as possible.

The size of shared memory becomes limiting for composed operations easily because of using multiple data structures in multiple operations. The data structures in shared memory has the lifetime of the thread block, thus its deallocation is not possible. To utilize shared memory well, it is necessary to manage its reusing explicitly. The problem here is how to arrange independent operations to minimize overall shared memory requirements by optimal reusing shared memory space.

When algebraic operations are developed and optimized, their composition lies only in data loading/storing from/to global memory and proper execution. Although the optimization of the shared memory reusing is analytical, the solution of the operations composition problem is rather experimental (multiple versions of the code must be written and benchmarked). We believe both algebraic operations composition and shared memory utilization can be performed automatically.

## V. PRELIMINARY RESULTS

In our previous work, the GetFEM library has been used for equations assembly and MUMPS for solving the system of equations [6]. Although better results have been reached later by tuning GetFEM settings, the time needed to perform equations assembly remained significant limitation. We have developed C++ implementation for CPU and CUDA implementation applying the programming model described above.

We have implemented two methods of equations assembly on GPU – the first one uses the global memory transfers between all operations, the second one has a fraction of frequently executed operations composed into one kernel performing communication via shared memory.

Our implementations have been benchmarked using machine equipped with Core2 Quad Q9550 (2.83 GHz), 8 GB RAM and GeForce GTX 280 connected via PCI-E 2.0 8x. Only assembling the equations of elements without initial data copy to GPU is benchmarked, because our target is to copy initial data to the GPU once and iteratively assemble the equations and solve them, both in GPU. We note here the shared memory composition is rather experimental and better results can be reached further.

Our CPU implementation is written to solve specific problem and thus it is faster than more general GetFEM implementation, outperforming GetFEM 9×. The CUDA implementation using only global memory for intermediate data transfers outperforms our CPU implementation running by one CPU core 8.3×. The part of GPU operations that communicate via shared memory in second mentioned GPU implementation runs 3.3× faster yielding 1.8× overall speedup. In Figure 1, the comparison of CPU implementation using one core, GPU implementation using only global memory and GPU implementation with fraction of operations communicating via shared memory is depicted.

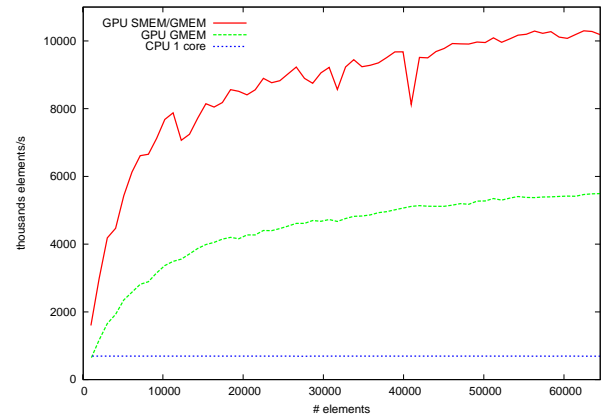


Fig. 1. Comparison of 3 different implementations of equations assembly for St.Venant-Kirchhoff material.

## VI. CONCLUSION AND FUTURE WORK

We have been concerned in acceleration of the equations assembly problem for St.Venant-Kirchhoff material in FEM. This problem is special because of two reasons:

- the high parallelism of GPU requires single operation multiple data approach, but the granularity of operation is too coarse to be performed by thread and too fine to be performed by block
- operations should exchange data via shared memory, but its data and parallelism requirements vary significantly yielding underutilization of GPU when all operations are composed into one kernel

We have implemented the equations assembly on GPU addressing these issues obtaining quite promising results. Moreover, the discussed problem is more general and the same approach can be applied to wide class of similar problems.

The future work will be focused on two areas. First, we try to increase the portion of computations transferring data via shared memory and optimize some algebraic operations code in current St.Venant-Kirchhoff implementation. Moreover, we plan to implement the equations assembly for another material to prove the general usability of our concept. Second, we will target the automatic searching for optimal composition of algebraic operations communicating via shared memory.

## REFERENCES

- [1] J.T.Oden. Finite Elements of Nonlinear Continua. McGraw Hill, 1972
- [2] Dominik Göddeke and Robert Strzodka. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations (Part 2: Double Precision GPUs). *Ergebnisberichte des Instituts für Angewandte Mathematik, Nr. 370, TU Dortmund*, 2008
- [3] Vasily Volkov and James Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. *Technical Report No. UCB/EECS-2008-49, EECS Department, University of California, Berkeley*, 2008.
- [4] NVIDIA CUDA Programming guide version 2.1. nVidia, 2008.
- [5] John E. Stone, James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry, Volume 28 Issue 16*, 2008.
- [6] Jiří Filipovič, Igor Peterlík, Luděk Matyska. On-Line Precomputation Algorithm for Real-Time Haptic Interaction with Non-Linear Deformable Bodies. In *Third Joint EuroHaptics Conference and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems*, 2009.