**Michał BORYSIAK, Zuzanna KRAWCZYK, Jacek STARZYŃSKI,
Robert SZMURŁO, Stanisław WINCENCIAK**

Warsaw University of Technology

CUDA accelerated finite element mesh morpher

Abstract. *We present a parallelized version of a simple 3D finite element mesh morpher. It is aimed at rapid creation of patient-specific finite element models of arms and legs. It implements a simple 3D algorithm of guided stretching. This approach needs only a few measurements of patients body. The program was implemented in C++ programming language and parallelized for CUDA API to utilize GPGPU (general computations with graphical processor units), what substantially speeds-up calculations comparing to a sequential code.*

Streszczenie. *W artykule przedstawiono zrównolegloną wersję prostego Algorytmu Sterowanego Naciągania Siatki przekształcającego trójwymiarowe siatki elementów skończonych w celu tworzenia zindywidualizowanych modeli kończyn do symulacjach stymulacji elektromagnetycznej. Utworzenie modelu dostosowanego do indywidualnych wymiarów pacjenta wymaga wykonania tylko kilku prostych pomiarów ciała. Program został zaimplementowany w C++ z wykorzystaniem środowiska CUDA SDK i wykorzystuje do obliczeń karty graficzne NVIDIA. Skutkuje to bardzo znaczącym przyspieszeniem obliczeń.* ***(Przyspieszenie programu do morfowania siatek trójwymiarowych przez wykorzystanie kart graficznych do obliczeń naukowych)***

Keywords: bioelectromagnetism, finite element method, realistic model of human body parallel computing
Słowa kluczowe: bioelektromagnetyzm, metoda elementów skończonych, realistyczne modele ciała, obliczenia równoległe

Introduction

The presented work is a part of the larger project aimed on the creation of software which will be used by medical staff in planning of electrical and magnetic therapeutic treatment. For such applications we need a tool which will quickly create a patient-similar model of body parts.

Patient specific, realistic, precise finite element models of a human body are nowadays becoming more and more popular [1, 2]. They allow more precise investigation of bioelectromagnetic phenomena.

In the classical approach to realistic model creation one starts with a fine quality cross-section images of human body. Based on the image database, a segmentation exhibiting the desired tissue distinction is created. Segmented images are then layered together to build a digital, voxelized model of the body. Finally the voxel grid is smoothed to obtain a finite element mesh. We can see, that creation of the realistic model is usually a time consuming process which needs exhaustive input data, and which can not be yet fully automated.

In our approach one starts withe a very fine, realistic model created with the classical approach on the basis of MRI scans of a sample body. This is the general, base model which will be later adapted to build a patient-specific one. Only a few simple measurements of external dimensions of the patients body form set of input data, which allow to morph (shrink and/or stretch) the base model, to fit it to the given patient. The base model is created only once using the time consuming classical approach, but morphing can be done very quickly.

Guided stretching algorithm

The guided stretching algorithm of mesh morphing was originally designed for 2D models and presented in [1]. Here we propose an extension aimed at effective creation of arms and legs models. The figures helping to understand the algorithm used in this paper were created for simplified meshes: cylinders morphed to bottle-like shapes. This solution was chosen to better emphasize the idea and present the way guided stretching works, but morphing real models of arms or legs works the same way.

In the 2D guided stretching algorithm the base model is stretched to the destination on the basis of a set of arbitrarily chosen “guiding vectors” (GV). The initial choice of GV location is made by a human on the foundation of initial experiments to provide the best fit for a given body part. The 3D version needs automatic choice of GV, because morphed objects are more complicated and relatively large number of GV is necessary to describe the morphing operator.

Luckily, the description of shape of an arm or a leg can be splitted into two sub-problems: we first match the overall length of the model (which can be done by a simple scaling) and then we perform morphing in a projected 2D space. Thus the 3D algorithm can be understood as a sort of projection of 2D algorithm into the third dimension.

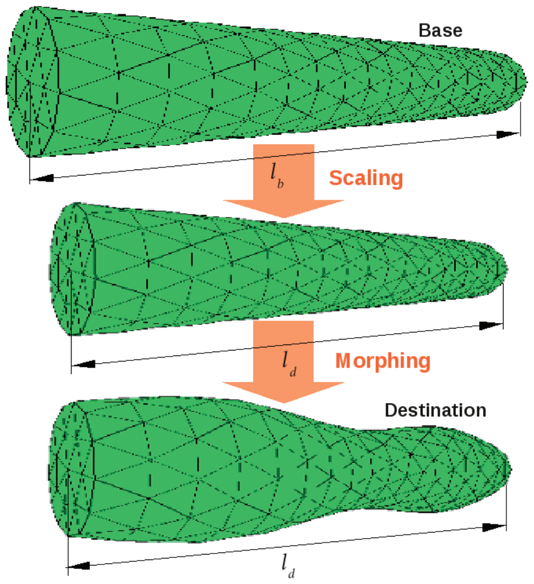

Fig. 1. Guided morphing – the idea of dimensioning and splitting

The algorithm of mesh scaling is quite obvious and can be described very briefly: knowing the length of the base model and length of the destination model we scale the base with respect of its geometrical center using the transformation matrix S:

$$S = \begin{bmatrix} l_d/l_b & 0 & 0 & 0 \\ 0 & l_d/l_b & 0 & 0 \\ 0 & 0 & l_d/l_b & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad (1)$$

where l_b is the length of the base model and l_d is the length of destination model (see Fig. 1).

The algorithm for automated choice of GV is based on the assumption that for any straight limb model we can select a straight central axis (along which l_b or l_d are measured).

Having this axis chosen, we create a Mapping Surface (MS) – a rough triangular, surface mesh wrapping the model. MS is created layer-by-layer along the axis of the model: moving with constant interval along the axis we compute the cross-section of the model and discretize the outer border of each cross-section into the same number of straight segments. The corresponding nodes of adjacent cross-sections are joined together (see Fig. 2).

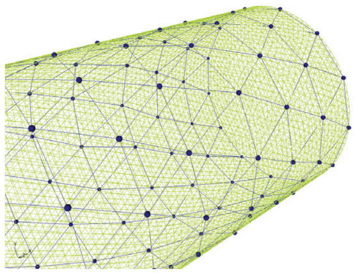

Fig. 2. Wrapping the base model (cyllinder, drawn with light green) with the layered surface mesh (layers are numbered, blue color); the axis of the model is in this simple case identical with the cylinder axis

To morph the mesh we need to create a Destination Surface (DS) first. To do it we measure the outer dimensions of the limb in the same numbers of intervals the MS has. Currently we approximate shape of each limb's cross-section with an ellipse (an thus we need to make 2 measurements of each cross-section), but this approximation can be made better in the future versions of the morpher. Discretizing each ellipse the same way the MS was created, we obtain the Destination Surface – a triangular mesh with exactly the same topology as MS, but wrapping the real limb of a patient.

Pairs of nodes (MS$\rightarrow$DS) form set of guiding vectors (see Fig. 3).

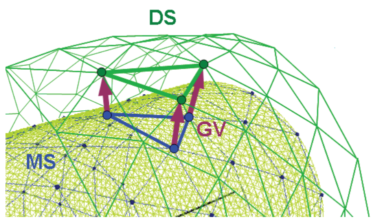

Fig. 3. Choosing GV for 3D algorithm: the base model (gray), wrapping mesh (red) and destination surface (green); guiding vectors for an example triangle is shown with purple color

To morph the base mesh into the destination we repeat the following Guided Stretching Algorithm for every vertex of the base mesh (see Fig. 4).

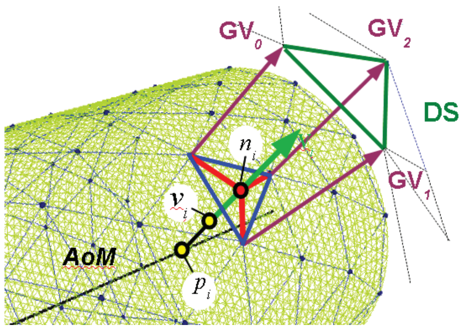

Fig. 4. The Guided Stretching Algorithm: node displacement

Guided Stretching Algorithm

1. Find a line defined by the vertex itself (v_i) and its projection onto the axis of the model (p_i).
2. Find intersection of this line and one of the triangles on the MS (n_i).
3. Calculate displacement $\mathbf{V}_i$ of the vertex as a weighted sum of three vectors which map the found triangle of MS to the DS of the limb model. The weights for each GV are proportional to areas of triangles built inside the triangle of the MS by the intersection (again refer to Fig. 4). The length of $\mathbf{V}_i$ is additionally scaled by the length ratio $|v_i - p_i|/|n_i - p_i|$.

After morphing we improve the quality of the stretched with Stellar – a public domain code for tetrahedral meshes improvement [4].

CUDA based parallel implementation

CUDA logical architecture (see Fig. 5) is well suited for data-parallel programming: we execute the same code on many data elements. These conditions are prefect for parallelization of the Guided Stretching Algorithm, since all morphing operations can be run in parallel for all vertices of the grid in the same time. We simply map grid of nodes into grid of CUDA threads.

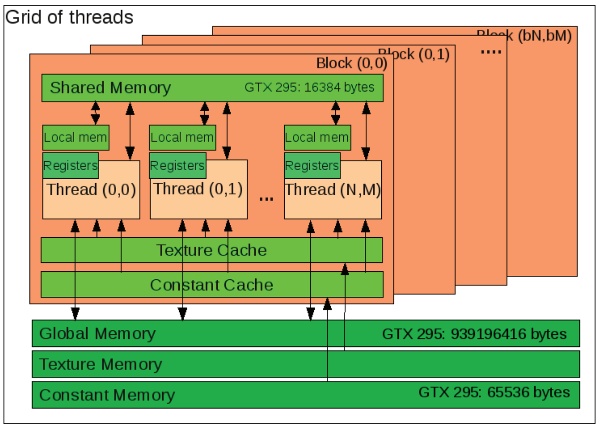

Fig. 5. CUDA Logical Architecture: each Block of Threads runs of separate multiprocessor, sharing some amount of fast memory. The bigger, but much slower global memory is accessible to all threads.

For sake of brevity we present here a slightly modified version of the code. It is not enough space here to go into details of finding procedure (steps 1 and 2 of the above presented Guided Stretching Algorithm), thus we have splitted parallel code into two *kernels*[1]: the first one (*findKernel*) represents steps 1 and 2, and the second (*morphKernel*) – the displacement described in step 3.

Table 1. The core implementation of Guided Stretching Algorithm

```
// prepare thread configuration                                                   1
int numOfThreads = numOfVertices;                                                 2
int threadsPerBlock = maxThreadsPerBlock; // device dependent                     3
int numOfBlocks = (numOfThreads % threadsPerBlock == 0)                           4
   ? numOfThreads / threadsPerBlock : numOfThreads / threadsPerBlock + 1;         5
                                                                                  6
// allocate memory, copy vertices to GPU                                          7
Vector *intersects; cudaMalloc((void **)&intersects, memSize);                    8
Vector *origins;    cudaMalloc((void **)&origins, memSize);                       9
int *found;         cudaMalloc((void **)&found, numOfVertices * sizeof(int));    10
cudaMemcpy(d_vertices, out, memSize, cudaMemcpyHostToDevice);                    11
                                                                                 12
// find Guiding Vectors                                                          13
findKernel <<<numOfBlocks, threadsPerBlock>>>                                    14
  (d_vertices, d_triangles, numOfVertices,  numOfTriangles,                      15
   numOfSlices, verticesPerSlice, origins, intersects, found);                   16
                                                                                 17
// do morphing                                                                   18
morphKernel<<<numOfBlocks, threadsPerBlock>>>                                    19
  (d_vertices, d_triangles, numOfVertices,  numOfTriangles,                      20
   origins, intersects, found);                                                  21
                                                                                 22
// finalize, retrieve results                                                    23
checkCUDAError("kernel invocation");                                             24
cudaMemcpy(out, d_vertices, memSize, cudaMemcpyDeviceToHost);                    25
```

The core implementation, presented in Table 1 can be logically divided into 4 parts: thread configuration (lines 1–5), memory allocation and copying data to the card (lines 7–11), running kernels which do the calculations (lines 13–21) and finally retrieving results from the card (line 25). The whole

[1]According to CUDA terminology a *kernel* is a function executed on the GP – simultaneously by many threads in parallel.

computational task is splitted into number of threads equal to the total number of grid vertices (nodes) or slightly greater. All threads are grouped so the maximal allowed number of threads runs on the same multiprocessor, what in future will allow us to use shared memory in the best way.

Table 2 presents the implementation of the node displacement procedure. The kernel performs only the check, if the calculated thread index corresponds to a node number (as you can see, we may generate some redundant threads – see lines 4 and 5 of the code in Table 1). The real hard work – calculation of weights and making of displacement vector $\mathbf{V}_i$ is done by the *displace* function.

Table 2. Node displacement

```
// morphig Kernel — routine to move single vertex
__global__ void morphKernel(Vector *verts, int numOfVerts, Triangle *mapsrf,
                            Vector *p_is, Vector *n_is, int *n_i2ms) {
  int vertIndex = blockIdx.x * blockDim.x + threadIdx.x;
  if( vertIndex < numOfVerts ) {
    Vector morphed_v_i;
    displace( verts[vertIndex], n_is[vertIndex], p_is[vertIndex],
              mapsrf[n_i2ms[vertIndex]], &morphed_v_i);
    copy(morphed_v_i, &verts[vertIndex]);
  }
}

// vertex displacement — the real work: morphs v_i making m_i
__device__ void displace(const Vector &v_i, const Vector &n_i,
                         const Vector &p_i, const Triangle &ms_trngl,
                         Vector *m_i) {
  Vector edge01; create(ms_trngl.vertx[0], ms_trngl.vertx[1], &edge01);
  Vector edge02; create(ms_trngl.vertx[0], ms_trngl.vertx[2], &edge02);
  float area= crossProdLgt(edge01, edge02);
  Vector n_i1; create(n_i, ms_trngl.vertx[0], &n_i1);
  Vector n_i2; create(n_i, ms_trngl.vertx[1], &n_i2);
  Vector n_i3; create(n_i, ms_trngl.vertx[2], &n_i3);
  float areaWeight1 = crossProdLgt(n_i2, n_i3) / area;
  float areaWeight2 = crossProdLgt(n_i3, n_i1) / area;
  float areaWeight3 = crossProdLgt(n_i1, n_i2) / area;
  float distWeight = length(v_i — p_i) / length(n_i — org);
  Vector displ1; mul(ms_trngl.GV[0], areaWeight1 * distWeight, &displ1);
  Vector displ2; mul(ms_trngl.GV[1], areaWeight2 * distWeight, &displ2);
  Vector displ3; mul(ms_trngl.GV[2], areaWeight3 * distWeight, &displ3);
  add(displ1, displ2, displ3, v_i, m_i);
}
```

Parallel speed-up

The speed-up of the parallel code was calculated using the example shown in Fig. 6. As it was mentioned before, a simple-shaped model (a cylinder with two parallel cylindrical rods inside) allows us to present the GSA results with readable figures. The destination surface is a bottle-like shape with ellipsoidal cross-sections. We can see, that the interior structure of the base grid (two internal cylinders) are morphed harmoniously with the outer shape.

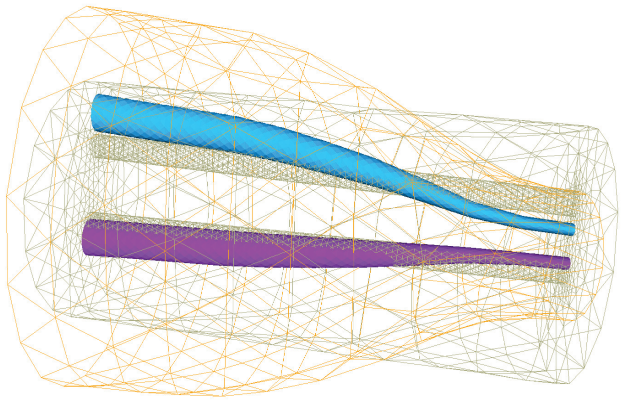

Fig. 6. Testing example: the base grid (gray) and the final model (orange, purple and blue)

The real speed-up of the morpher for three different hardware configurations is compared in Table 3. As we can see, the achieved results are impressing even if we take into account the whole time of program run, including disk read-write operations. Comparison of time of calculations only (summarized in Table 4) reveals even bigger speed-up due the GPU utilization.

The reader should notice, that presented here parallel implementation of GSA is relatively simple and that it does not fully utilize the capabilities of GPU. The last three rows of Table 4 show not only times used for all CUDA base operations (lines 8–25 of the code shown in Table 1) but also (in parentheses) time used by the kernels only (lines 14–21 of the same listing). We can see, that memory operations consume quite substantial part of time. Use of GPU's shared memory can greatly reduce this time and speed-up kernels operations as well [5]. This will be the next move in improvement of the presented code.

Table 3. Program speed-up on three different computers

	Hardware	Speed-UP
oer	Intel Core2 Quad CPU Q6600 @2.4GHz + GeForce 9800 GT (14×8=112 cores)	~**19**×
ham	Intel Core2 Quad CPU Q8200 @2.4GHz + GeForce GTX 260 (27×8=216 cores)	~**41**×
len	Intel Core2 Quad CPU E8400 @3.0GHz + GeForce GTX 295 (30×8=240 cores)	~**43**×

Table 4. Speed-up of the morphing procedure

Mesh:	small (20 869 nodes, 117 572 elements)	big (116 111 nodes, 668 611 elements)
	CPU only	
oer	36.443s	211.355s
ham	27.353s	151.537s
len	21.197s	119.561s
	CPU+GPU	
oer	1.33496s (1.28343s)	6.97691s (6.92625s)
ham	0.15700s (0.09934s)	0.43444s (0.37967s)
len	0.11103s (0.07458s)	0.37769s (0.33991s)

Conclusions

Even quite straightforward implementation with CUDA allowed substantial speed-up of the GSA code. The scaling of the parallel code is excellent for our problem. Better fit to GPU architecture should bring further reduction of computation time.

In this case CUDA offers impressing ratio of benefit compared to cost of hardware and amount of work needed to change the code.

Acknowledgment: this work was partially supported by Polish Ministry of Science and Higher Education (research grant no. N N510 148838).

REFERENCES

[1] R. Szmurło, J. Starzyński: Specimen-specific finite element models of human head obtained with mesh-morphing, *Przeglad Elektrotechniczny (Electrical Review)*, R. 85 NR 4/2009, pp. 47-49

[2] I.A. Sigal, M.R. Hardisty, C.M. Whyne: Mesh-morphing algorithms for specimen-specific finite element modeling, *J Biomech.*, 2008, pp. 1381-1389

[3] M. Borysiak, Z. Krawczyk, J. Starzyński: Creating Patient-Specific Finite Element Models with a Simple Mesh Morpher, *Proceedings of Computer Applications in Electrical Engineering, ZkwE'2010*, Poznań, Poland, Apr. 19-21, 2010

[4] Bryan Matthew Klingner and Jonathan Richard Shewchuk: Aggressive Tetrahedral Mesh Improvement, *Proceedings of the 16th International Meshing Roundtable (Seattle, Washington)*, pages 3–23, October 2007.

[5] NVIDIA: *CUDA, Programming Guide*, ver. 3.0, 2010, downloaded from http://developer.nvidia.com

Authors: Eng. Michał Borysiak, Eng. Zuzanna Krawczyk, Prof. Jacek Starzyński, Ph.D. Robert Szmurło, Prof. S. Wincenciak, IETiSIP, Warsaw University of Technology, ul. Koszykowa 75, 00-662 Warszawa, Poland, email: jstar@iem.pw.edu.pl