Accelerating Anisotropic Mesh Adaptivity on nVIDIA's CUDA Using Texture Interpolation

Georgios Rokos[1], Gerard Gorman[2], and Paul H J Kelly[1]

[1]Software Performance Optimisation Group,
Department of Computing,
[2]Applied Modelling and Computation Group,
Department of Earth Science and Engineering,
Imperial College London,
South Kensington Campus, London SW7 2AZ, United Kingdom,
{georgios.rokos09,g.gorman,p.kelly}@imperial.ac.uk

Abstract. Anisotropic mesh smoothing is used to generate optimised meshes for Computational Fluid Dynamics (CFD). Adapting the size and shape of elements in an unstructured mesh to a specification encoded in a metric tensor field is done by relocating mesh vertices. This computationally intensive task can be accelerated by engaging nVIDIA's CUDA-enabled GPUs. This article describes the algorithmic background, the design choices and the implementation details that led to a mesh-smoothing application running in double-precision on a Tesla C2050 board. Engaging CUDA's texturing hardware to manipulate the metric tensor field accelerates execution by up to 6.2 times, leading to a total speedup of up to 148 times over the serial CPU code and up to 15 times over the 12-threaded OpenMP code.

Keywords: anisotropic mesh adaptivity, vertex smoothing, parallel execution, CUDA, metric tensor field, texturing hardware.

1 Introduction

Mesh adaptivity is an important numerical technology in Computational Fluid Dynamics (CFD). CFD problems are solved numerically using unstructured meshes, which essentially represent the discrete form of the problem. In order for this representation to be accurate and efficient, meshes have to be adapted according to some kind of error estimation. Furthermore, this error estimation may also encode information about a possible spatial orientation of the problem under consideration, in which case we say that the underlying dynamics is anisotropic and the error estimation is described using a metric tensor field.

One sophisticated adaptation technique, suitable for anisotropic problems, is Vertex Smoothing. Adapting a mesh to an error estimation involves an enormous amount of floating-point operations which can push even the most powerful processing units to their limits. The CUDA platform offers great computational power at relatively low cost. These properties make it a perfect candidate for

accelerating mesh adaptation. We wrote a new application framework which implements Pain's smoothing algorithm [7] along with the proposal by Freitag et al. [6] for its parallel execution, enabling mesh adaptation to be accelerated on CUDA GPUs. The main objectives achieved through this project can be summarised as follows:

- This is the first adaptive mesh algorithm implemented on CUDA as far as the authors are aware.
- It resulted in an application running in double-precision on a Fermi-based Tesla board up to 15 times faster than on a 12-core server.
- The key optimisation proved to be the use of texturing hardware to store and interpolate the metric tensor field, which offered speed-ups of up to 6.2 times over the simple CUDA code.

The rest of the article is organised as follows: Section 2 contains a comprehensive description of the algorithmic background and Section 3 describes how the application was designed and implemented. Section 4 presents performance graphs comparing the serial code against OpenMP and CUDA versions. We conclude this paper and discuss ideas for future work in Section 5.

2 Background

2.1 PDEs, meshes and mesh quality

The Finite Element Method (FEM) is a common numerical approach for the solution of PDEs, in which the problem space is discretised into smaller subregions, usually of triangular (in 2D) or tetrahedral (in 3D) shape. These subregions, referred to as elements, form a mesh. The equation is then discretised and solved inside each element. Common discretisation techniques often result in low quality meshes and this affects both convergence speed and solution accuracy [5]. A posteriori error estimations on the PDE solution help evaluate a quality functional [10] and determine the low-quality elements, which a mesh-improving algorithm tries to "adapt" towards the correct solution. Unstructured meshes, i.e. meshes in which a node can be connected to an arbitrary number of other nodes, offer greater numerical flexibility but their more complex representation is followed by higher computational cost [8].

2.2 Anisotropic PDEs

A problem is said to be "anisotropic" if its solution exhibits directional dependencies. In other words, an anisotropic mesh contains elements which have some (suitable) orientation, i.e. size and shape. The process of anisotropic mesh adaptation begins with a (usually automatically) triangulated mesh as input and results in a new mesh, the elements of which have been adapted according to some error estimation. This estimation is given in the form of a metric tensor field, i.e. a tensor which, for each point in the 2-D (or 3-D) space, represents

the desired length and orientation of an edge containing this point. As was the case with the PDE itself, the metric tensor is also discretised; more precisely, it is discretised node-wise. The value of the error at an in-between point can be taken by interpolating the error from nearby nodes. An example of adapting a mesh to the requirements of an anisotropic problem is shown in Figure 1.

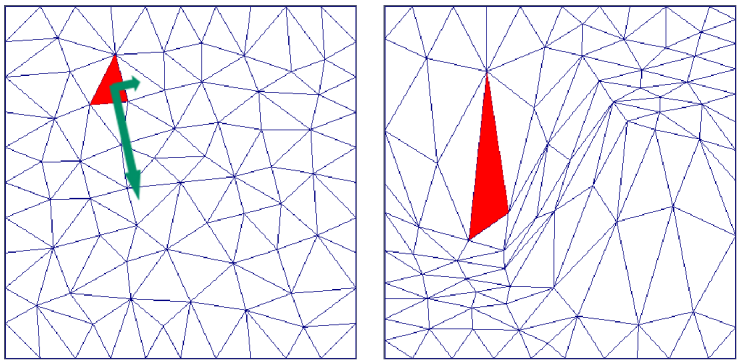

Fig. 1. Example of anisotropic mesh adaptation. The initial red triangle is stretched according to the metric tensor value (green arrow).

Adapting a mesh so that it distributes the error uniformly over the whole mesh is, in essence, equivalent to constructing a uniform mesh consisting of equilateral triangles with respect to the non-Euclidean metric $M(\mathbf{x})$. This concept can be more easily grasped if we give an analogous example with a distorted space like a piece of rubber that has been stretched (see Figure 2). In this example, our domain is the piece of rubber and we want to solve a PDE in this domain. According to the objective functional we used, all triangles in the distorted (stretched) piece of rubber should be equilateral with edges of unit length. When we release the rubber and let it come back to its original shape, the triangles will look compressed and elongated.

The metric tensor M can be decomposed as

$$M = Q\Lambda Q^T$$

where Λ is the diagonal matrix, the components of which are the eigenvalues of M and Q is an orthonormal matrix consisting of eigenvectors Q^i. Geometrically, Q represents a rotation of the axis system so that the base vectors show the direction to which the element has to be stretched and Λ represents the amount of distortion (stretching). Each eigenvalue λ^i represents the squared ideal length of an edge in the direction Q^i [8].

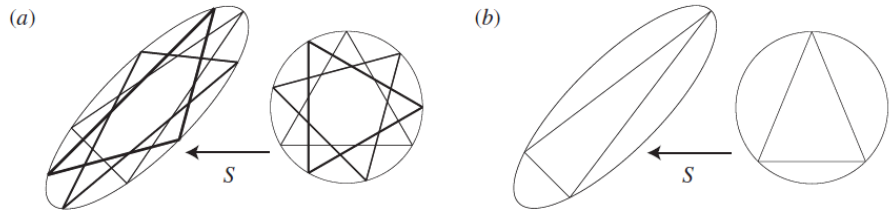

Fig. 2. Example of mapping of triangles between the standard Euclidean space (left shapes) and metric space (right shapes). In case (a), the elements in the physical space are of the desired size and shape, so they appear as equilateral triangles with edges of unit length in the metric space. In case (b), the triangle does not have the desired geometrical properties, so it does not map to an equilateral triangle in the metric space.(Figure from [8])

2.3 Vertex Smoothing and the algorithm by Pain et al.

Vertex smoothing is an adaptive algorithm which tries to improve mesh quality by relocating existing mesh vertices. Contrary to other techniques, which we discuss in Section 5, it leaves the mesh topology intact, i.e. connectivity between nodes does not change. All elements affected by the relocation of one vertex form an area called a cavity. A cavity is defined by its central, free vertex and all incident vertices. A vertex smoothing algorithm tries to equidistribute the quality among cavity elements by relocating the central vertex to a new position. Optimisation takes into account only elements belonging to the cavity, which means that only one vertex is considered for relocation at a time. An example of optimising a cavity is shown in Figure 3 [9].

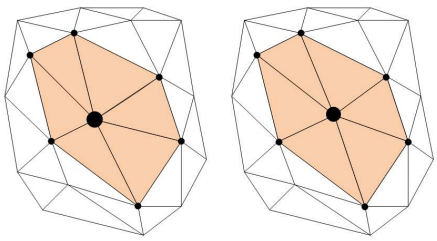

Fig. 3. Vertex smoothing example. The vertex under consideration is the one marked with a big black circle. The local problem area is the light-orange one. Left figure shows the cavity before smoothing. Right figure shows the result of local smoothing. (figure from [9])

The scope of optimisation is the cavity, therefore vertex smoothing is a local optimisation technique. The algorithm moves towards the global optimum through a series of local optimisations. The local nature of vertex smoothing leads to the need for optimising a cavity over and over again. After having smoothed a vertex, smoothing an incident vertex in the scope of its cavity may change the quality of the first cavity. Because of this property, the algorithm has to be applied a number of times in order to bring things to an equilibrium.

Running a smoothing algorithm in parallel can be done as dictated by the framework proposed by Freitag et al. [6]. In a parallel execution, we cannot smooth arbitrarily any vertices simultaneously. When a vertex is smoothed, all adjacent vertices have to be locked at their old positions. This means that no two adjacent vertices can be smoothed at the same time. In order to satisfy this requirement and ensure hazard-free parallel execution, mesh vertices have to be coloured so that no two adjacent vertices share the same colour. All vertices of the same colour form an independent set, which means that they are completely independent from each other and can be smoothed simultaneously. This is a classic graph colouring problem, with the graph being the mesh, graph nodes being mesh vertices and graph edges being mesh edges.

Pain et al. proposed a non-differential method to perform vertex smoothing [7]. A cavity C_i consists of a central vertex V_i and all adjacent vertices V_j. Let L_i be the set of all edges connecting the central vertex to all adjacent vertices. The aim is to equate the lengths of all edges $\in L_i$ (recall that the optimal cavity is the one in which all triangles are equilateral with edges of unit length with respect to some error metric). The length of an edge l in metric space is defined as $r_l = \left(u_l^T M_l u_l\right)$, where M_l is the value of the metric tensor field in the middle of the edge.

Let $\hat{p}^i$ be the initial position of the central vertex and p^i the new one. Then, the length of an edge in the standard Euclidean space is $u_l = p^i - y_l^i$, where y_l^i is the position of a non-central cavity vertex V_j. Also, it is important to use relaxation of p^i for consistency reasons, using $x^i = wp^i + (1-w)p^i, w \in (0,1]$. In this project, $w = 0.5$. We define $q^i = \sum_{l \in L_i} M_l y_l^i$ and $A^i = \sum_{l \in L_i} M_l$ and introduce a diagonal matrix D^i to ensure diagonal dominance and insensitivity to round-off error:

$$D_{jk}^i = \begin{cases} max A_{jj}^i, (1+\sigma) \sum_{m=1, m \neq j} \mid A_{jm}^i \mid, \; if \; j = k \\ 0, \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad if \; j \neq k \end{cases}$$

In this project, $\sigma = 0.01$. Then, x^i can be found by solving the linear system

$$(D^i + A^i)(x^i - \hat{p}^i) = w(q^i - A^i \hat{p}^i).$$

In the case of boundary vertices, i.e. vertices which are allowed to move only along a line (the mesh boundary), a modification of the above algorithm has to be used. The restriction that the vertex can only move along a line means that the new position x^i can be calculated using the equation

$$x^i = a_C^i u_l^i + \hat{p}^i,$$

where u_l^i is the unit vector tangential to the boundary line and a_C^i is the displacement along this line measured from the initial position $\hat{p}^i$ of the vertex. a_c^i can be calculated from the equation

$$(D^i + \hat{M}^i)a_c^i = wg^i,$$

where $\hat{M}^i = {u_l^i}^T \sum_{l \in L_i} M_l u_l^i$ and $g^i = \sum_{l \in L_i} {u_l^i}^T M_l (x^i - \hat{p}^i)$.

2.4 CUDA's Texturing Hardware

Texturing hardware is an important heritage left by the graphics-processing roots of CUDA. Reading data from texture memory can have a lot of performance benefits, compared to global memory accesses. Texture memory is cached in a separate texture cache (optimised for 2D spatial locality), leaving more room in shared memory/L1 cache. If memory accesses do not follow the patterns required to get good performance (as is the case with unstructured problems), higher bandwidth can be achieved provided there is some locality on texture fetches. Additionally, addressing calculations are executed automatically by dedicated hardware outside processing elements, so that CUDA cores are not occupied by this task and the programmer does not have to care about addressing [3, 4].

The most important texturing feature is interpolation. Textures are discretised data from a (theoretically) continuous domain. In graphics processing, a texture value may be needed at a coordinate which falls between discretisation points, in which case some kind of texture data filtering has to be performed. Interpolating values from the four nearest discretisation points is the most common type of texture filtering, called linear filtering. In two dimensions, the result $tex(x, y)$ of linear filtering is:

$$\begin{aligned} tex(x,y) = (1-\alpha)(1-\beta)T[i,j] + \alpha(1-\beta)T[i+1,j] + \\ + (1-\alpha)\beta T[i,j+1] + \alpha\beta T[i+1,j+1], \end{aligned}$$

where α is the horizontal distance of point (x, y) from the nearest texture sample (discretisation point) $T[i, j]$ and β is the vertical distance. The key point is that this calculation can be automatically performed by dedicated texturing hardware outside multiprocessors. Interpolation performed by this specialised hardware is done faster than performing it in software. Apart from freeing CUDA's multiprocessors to perform other tasks, it also decreases the size of the computational kernel by occupying fewer registers, which is quite important for the maximum achievable warp occupancy.

3 Design and Implementation

The application we developed targets nVIDIA's Fermi architecture (compute capability 2.0). Double-precision arithmetic was preferred over single-precision in order to make the algorithm more robust to the order in which arithmetic operations take place (a quite common problem in numerical analysis) and reduce

Listing 1.1. Setting up texture memory.

```
#if defined(USE_TEXTURE_MEMORY)
  texture<float4, 2, cudaReadModeElementType> metricTex;

  cudaChannelFormatDesc channelDesc =
    cudaCreateChannelDesc(32, 32, 32, 32, cudaChannelFormatKindFloat);
  cudaMallocArray(&cudaMetricField, &channelDesc, textDim, textDim);
  cudaMemcpyToArray(cudaMetricField, 0, 0, hostMetricField,
    textDim * textDim * sizeof(float4), cudaMemcpyHostToDevice);

  metricTex.normalized = true;
  metricTex.filterMode = cudaFilterModeLinear;
  metricTex.addressMode[0] = cudaAddressModeClamp;
  metricTex.addressMode[1] = cudaAddressModeClamp;

  cudaBindTextureToArray(metricTex, cudaMetricField, channelDesc);
#else
  cudaMalloc((void **) &cudaMetricField,
    metricDim * metricDim * 4 * sizeof(float));
  cudaMemcpy(cudaMetricValues, hostMetricValues,
    metricDim * metricDim * 4 * sizeof(float), cudaMemcpyHostToDevice);
#endif
```

round-off errors. The application adapts 2D meshes using the vertex smoothing scheme proposed by Pain et al. [7]. By performing vertex smoothing, node connectivity remains constant and there is no need to re-colour the mesh after every iteration. Graph colouring was implemented using a single-threaded and greedy colouring algorithm, called First Fit Colouring [1], which runs adequately fast and colours the mesh with satisfactorily few colours (7-8 on average).

The mesh is represented using two arrays: an array V of vertices (a vertex is simply a pair of coordinates) and an array C of cavities. A cavity in the i-th position of C is the cavity defined by vertex V[i], i.e. the cavity in which V[i] is the central vertex, and (this cavity) is in turn an array containing the indices in array V of all vertices which are adjacent to V[i]. E.g. if vertex V[0] is connected to vertices V[3], V[5], V[10] and V[12], then C[0] is the cavity in which V[0] is the free vertex and C[0] = 3, 5, 10, 12. There is also a simple representation of independent sets, each one being an array containing the indices of all vertices belonging to that set.

As was described in Section 2, the metric tensor field is discretised vertex-wise, so it could be represented by extending the definition of a vertex to include the metric tensor value associated with that vertex, in addition to the vertex's coordinates. However, looking at the smoothing algorithm, it becomes apparent that the middle of an edge (where the metric value is needed) will most probably not coincide with a discretisation point, so the metric value has to be found by interpolating the values from nearby discretisation points. If the field is stored as a texture, interpolation can be done automatically by CUDA's texturing units. Linear filtering is quite suitable and yields good interpolation results, even when the metric tensor field has a lot of discontinuities.

More insight into the role of the metric tensor field reveals that it is just an estimation or indication about the desired orientation of an element and we

Listing 1.2. Accessing a metric tensor field value.

```
#if defined(USE_TEXTURE_MEMORY)
  float4 floatMetric = tex2D(metricTex, iCoord, jCoord);
#else
  double iIndex = jCoord * metricDim, jIndex = iCoord * metricDim;
  int i = floor(((metricDim - 1) / metricDim) * iIndex);
  int j = floor(((metricDim - 1) / metricDim) * jIndex);
  iIndex -= i; jIndex -= j;

  if(i == metricDim - 1)       // top or bottom boundary
    metric = cudaMetricValues[metricDim*(metricDim-1) + j] * (1-jIndex) +
             cudaMetricValues[metricDim*(metricDim-1) + (j+1)] * jIndex;
  else if(j == metricDim - 1) // left or right boundary
    metric = cudaMetricValues[(i+1)*metricDim - 1] * (1-iIndex) +
             cudaMetricValues[(i+2)*metricDim - 1] * iIndex;
  else
    metric = cudaMetricValues[i*metricDim + j] * (1-iIndex)*(1-jIndex) +
             cudaMetricValues[i*metricDim + (j+1)] * (1-iIndex)*jIndex +
             cudaMetricValues[(i+1)*metricDim + j] * iIndex*(1-jIndex) +
             cudaMetricValues[(i+1)*metricDim + (j+1)] * iIndex*jIndex;
#endif
```

have observed that it does not have to be as accurate as possible. For this reason, single-precision representation (double-precision is not supported for textures) is more than enough and the data structure used to store it can be a 2D array, organised using the GPU's blocked texture storage layout. In order to convert the unstructured representation to an array, we super-sample the initial field with adequate resolution and store these samples in an array. The initial, auto-generated mesh is anyway quite uniform, i.e. elements tend to be equilateral triangles and vertices are equally spaced from each other, a state which is not very different from a 2D-array representation.

In 2D, the metric tensor field is a 2×2 matrix, so it can be represented as a 4-element vector of single-precision floating-point values. Copying data from host to device as textures is demonstrated in Listing 1.1. Retrieving the value of the metric tensor field at any point in the mesh is just a texture fetch, as can be seen in Listing 1.2, which also contrasts the addressing and interpolation overhead we avoid.

Subsequent adaptation attempts will have to use the unstructured representation of the field. After adapting the mesh, we re-solve the PDE and make new error estimations, which lead to a new metric tensor field, discretised at the nodes of an anisotropic, non-uniform mesh. In this case, different resolution will be needed in different areas of the mesh and we have to follow the unstructured approach. This does not reduce the significance of using texturing hardware. The first adaptation attempt is the one which really needs to be sped-up, as it needs $\Theta(number\_of\_vertices)$ iterations to converge, inducing the most extensive changes to the mesh. After that, the mesh will have, more or less, acquired its final "shape", so subsequent attempts will only need a few iterations to improve it.

In devices of CUDA's compute capability 2.0 and above, the on-chip memory is used both as shared memory and L1 cache. The unstructured nature of

Listing 1.3. OpenMP execution

```
for(int indSetNo = 0; indSetNo < numberOfSets; indSetNo++) {
  vertexID iSet[] = indSets[indSetNo];

#pragma omp parallel for private(setIterator)
  for(int setIterator = 0; setIterator < verticesInSet; setIterator++) {
    cavityID cavity = iSet[setIterator];
    if(!meshCavities[cavity].isOnBoundary())
      newCoords = relocateInnerVertex(...);
    else
      newCoords = relocateOuterVertex(...);
  }
}
```

Listing 1.4. CUDA execution

```
for(int indSetNo = 0; indSetNo < numberOfSets; indSetNo++) {
  dim3 numBlocks(ceil((double) verticesInSet / threadsPerBlock));
  kernel<<<numBlocks, threadsPerBlock>>>(indSets[indSetNo]);
  cudaThreadSynchronize();
}

__device__ void kernel(IndependentSet iSet) {
  int vertex = blockIdx.x * blockDim.x + threadIdx.x;
  if(vertex < verticesInSet) {
    cavityID cavity = iSet[vertex];
    if(!meshCavities[cavity].isOnBoundary())
      newCoords = relocateInnerVertex(...);
    else
      newCoords = relocateOuterVertex(...);
  }
}
```

anisotropic mesh adaptivity has not allowed us to use shared memory explicitly. On the other hand, a hardware-managed L1 cache exploits data locality more conveniently. Configuring the on-chip memory as 16KB of shared memory with 48KB of L1 cache can be done [3] by preceding the kernel invocation with a statement like:

$cudaFuncSetCacheConfig(optimizationKernel, cudaFuncCachePreferL1);$.

Parallel execution is based on the independent sets. The way cavities are assigned to OpenMP resp. CUDA threads can be seen in Listing 1.3 resp. Listing 1.4. Recall from the description of the vertex smoothing algorithm that boundary vertices are smoothed using a variation of the main algorithm. In order to avoid thread divergence, which is problematic for a CUDA kernel, boundary vertices are put into dedicated independent sets, so that a dedicated set contains vertices of the same "kind".

4 Experimental Evaluation

All experiments were run on node CX1 of Imperial College's HPC supercomputer, hosting two Intel "Gulftown" six-core Xeon X5650 CPUs (2.8GHz), 24GB

RAM and a nVIDIA Tesla C2050 graphics board. The operating system was Red Hat Enterprise Linux Client release 5.5 running Linux kernel 2.6.18. CPU code was compiled with GCC version 4.1.2 giving the -O3 flag, whereas for GPU code we used CUDA SDK 3.1 and CUDA compilation tools, release 3.1, V0.2.1221, with the -O2 flag. Experiments were done using the nVIDIA Forceware driver, version 260.19.29.

We have compared the running time between a single-threaded execution, a 12-threaded OpenMP execution and CUDA execution with and without engaging texturing hardware. Timing results include the time it takes to copy data between host and device, but no measurement includes the time it takes to read in the unstructured grid from the disk, construct the mesh, colour it or write back the results to the disk, because these tasks are always performed in a single-threaded fashion on the host side. On the other hand, the time to copy data between host and device is trivial because these transfers take place only twice during an execution (copying the initial mesh to the device at the beginning and copying the adapted mesh back to the host at the end) and when there are thousands of iterations this time is amortised.

The optimisation kernel occupies 59 registers in the non-textured version and 51 registers in the textured one. Using the Occupancy Calculator [2] and experimental measurements, it was found that the best CUDA execution configuration is 32 threads per block, which gives an occupancy of 33.3%. In both cases, occupancy is very low, suggesting the future optimisation of breaking down the kernel into smaller parts.

Table 1 presents the amount of time each version of the code needs to perform $1,000$ iterations over various meshes. Figure 4 shows the relative speedup between these versions. The 12-threaded OpenMP version is steadily ~ 10 times faster than the serial code. The non-textured CUDA version runs on average 24 times faster than the serial code (peaking at 42 times) and 2.5 times faster than the OpenMP version (peaking at 4.7 times). Enabling texturing support, the CUDA code runs on average twice as fast as its non-textured counterpart (peaking at 6.2 times). Compared to the host side, it runs on average 60 times faster than the serial CPU version (peaking at 148 times) and 6 times faster than the OpenMP version (peaking at 15 times).

The high performance divergence and the unpredictable (to some extent) behaviour of a CUDA implementation come as a consequence of the highly unstructured nature of the problem. We expect substantial differences from one mesh to another in terms of achievable data locality, partitioning of data in global memory and degree of coalescence of memory accesses. This uncertainty could be mitigated by implementing a two-level mesh partitioning scheme: one topological partitioning of the mesh into mini-partitions, so that the whole mini-partition fits in the on-chip memory, and a second logical partitioning (graph colouring) inside each mini-partition for the purpose of correct parallel execution.

Number of mesh vertices	CPU 1 Thread	CPU 12 Threads	CUDA (no texturing)	CUDA (texturing)
25,472	20.76	2.156	1.243	0.610
56,878	46.82	4.676	2.281	1.163
157,673	144.3	15.27	5.902	3.172
402,849	510.3	52.36	15.03	8.249
1,002,001	777.6	75.34	28.17	5.664
4,004,001	3,203	318.0	134.2	21.64
5,654,659	8,921	983.9	210.0	114.2

Table 1. Comparison of the execution time in seconds between the serial, the 12-threaded OpenMP, the non-textured CUDA and the textured CUDA versions, performing 1,000 iterations over meshes of variable size.

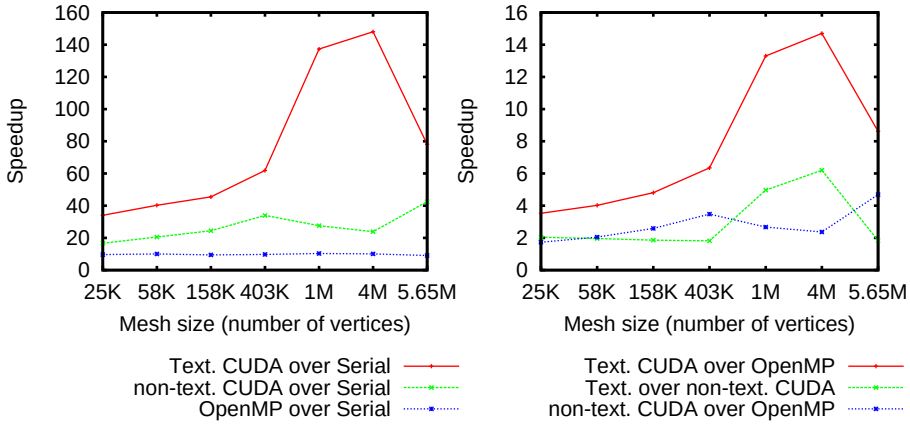

Fig. 4. Relative speedup between the serial, the OpenMP, the non-textured CUDA and the textured CUDA versions.

5 Conclusions and Future Work

The aim of this project was to determine the extent to which a Fermi-based GPU can still be efficient when it has to deal with unstructured problems. The experimental results show that the capabilities of this architecture extend well beyond the borders of structured applications, which are the norm in evaluating and demonstrating processing hardware. A single Tesla C2050 board outperformed 12 Nehalem cores by many times and there is still room for improvement, as is suggested by the low warp occupancy and the scope for improved data locality.

When it comes to texturing hardware, it was shown that it offers substantial amounts of computational power and can more than double performance in problems with appropriate characteristics, like the metric tensor field of anisotropic mesh adaptivity. The assistance of texturing hardware in the 3D version of the problem (a 3D implementation is planned for future work) is expected to be even

more valuable. In a 3D metric tensor field we have to interpolate the values from the 8 nearest points and doing so requires (compared to 2D problems) double the data volume to be fetched from memory and three times more arithmetic.

Apart from the aforementioned optimisations, this project leaves many other topics open to further study. Perhaps, the most interesting direction is the implementation of a much heavier and of higher quality smoothing algorithm, based on differential methods, which is called optimisation-based smoothing [5]. Additionally, vertex smoothing is usually combined with other adaptive methods, such as regular refinement, edge flipping and edge collapsing [6, 7]. The development of a mesh improving application which manipulates all these techniques and the assessment of its performance on CUDA (and possibly other high-performance architectures through OpenCL) is in progress.

References

1. Al-Omari, H., Sabri, K.E.: New graph coloring algorithms. Journal of Mathematics and Statistics (2006)
2. nVIDIA Corporation: CUDA GPU Occupancy Calculator, `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`
3. nVIDIA Corporation: nVIDIA CUDA Programming Guide, Version 3.1. Tech. rep. (2010)
4. nVIDIA Corporation: nVIDIA CUDA Reference Manual, Version 3.1. Tech. rep. (2010)
5. Freitag, L., Jones, M., Plassmann, P.: An Efficient Parallel Algorithm for Mesh Smoothing. In: INTERNATIONAL MESHING ROUNDTABLE. pp. 47–58 (1995)
6. Freitag, L.F., Jones, M.T., Plassmann, P.E.: The Scalability Of Mesh Improvement Algorithms. In: IMA VOLUMES IN MATHEMATICS AND ITS APPLICATIONS. pp. 185–212. Springer-Verlag (1998)
7. Pain, C.C., Umpleby, A.P., de Oliveira, C.R.E., Goddard, A.J.H.: Tetrahedral mesh optimisation and adaptivity for steady-state and transient finite element calculations. Computer Methods in Applied Mechanics and Engineering 190(29-30), 3771 – 3796 (2001), `http://www.sciencedirect.com/science/article/B6V29-42RMN8G-5/2/8c901884db5fd4b67d19b63fb9691284`
8. Piggott, M.D., Farrell, P.E., Wilson, C.R., Gorman, G.J., Pain, C.C.: Anisotropic mesh adaptivity for multi-scale ocean modelling. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 367(1907), 4591–4611 (2009), `http://rsta.royalsocietypublishing.org/content/367/1907/4591.abstract`
9. Rokos, G.: ISO Thesis: Study of Anisotropic Mesh Adaptivity and its Parallel Execution. Imperial College London (2010)
10. Vasilevskii, Y., Lipnikov, K.: An adaptive algorithm for quasioptimal mesh generation. Computational Mathematics and Mathematical Physics 39(9), 1468–1486 (1999)