Kybernetika

Tomáš Oberhuber; Atsushi Suzuki; Vítězslav Žabka

The CUDA implementation of the method of lines for the curvature dependent flows

Kybernetika, Vol. 47 (2011), No. 2, 251--272

Persistent URL: `http://dml.cz/dmlcz/141571`

Terms of use:

© Institute of Information Theory and Automation AS CR, 2011

Institute of Mathematics of the Academy of Sciences of the Czech Republic provides access to digitized documents strictly for personal use. Each copy of any part of this document must contain these *Terms of use*.

This paper has been digitized, optimized for electronic delivery and stamped with digital signature within the project *DML-CZ: The Czech Digital Mathematics Library* `http://project.dml.cz`

KYBERNETIKA — VOLUME 47 (2011), NUMBER 2, PAGES 251–272

THE CUDA IMPLEMENTATION OF THE METHOD OF LINES FOR THE CURVATURE DEPENDENT FLOWS

Tomáš Oberhuber, Atsushi Suzuki and Vítězslav Žabka

We study the use of a GPU for the numerical approximation of the curvature dependent flows of graphs – the mean-curvature flow and the Willmore flow. Both problems are often applied in image processing where fast solvers are required. We approximate these problems using the complementary finite volume method combined with the method of lines. We obtain a system of ordinary differential equations which we solve by the Runge–Kutta–Merson solver. It is a robust solver with an automatic choice of the integration time step. We implement this solver on CPU but also on GPU using the CUDA toolkit. We demonstrate that the mean-curvature flow can be successfully approximated in single precision arithmetic with the speed-up almost 17 on the Nvidia GeForce GTX 280 card compared to Intel Core 2 Quad CPU. On the same card, we obtain the speed-up 7 in double precision arithmetic which is necessary for the fourth order problem – the Willmore flow of graphs. Both speed-ups were achieved without affecting the accuracy of the approximation. The article is structured in such way that the reader interested only in the implementation of the Runge–Kutta–Merson solver on the GPU can skip the sections containing the mathematical formulation of the problems.

Keywords: GPGPU, CUDA, parallel algorithms, high performance computing, differential geometry, mean-curvature flow, Willmore flow, Runge–Kutta method, method of lines, explicit scheme, complementary finite volume method

Classification: 68W10, 35K55, 35K52, 53A05, 53C44, 74S10, 74G15

1. INTRODUCTION

GPUs (Graphics processing units) are devices designed to accelerate visualization of 3D objects in computer graphics. Originally they were designed especially for the computer games. For this purpose, the GPU designers concentrated on the processing of detailed textures rather than on complex geometry with millions of polygons. Therefore the GPUs are equipped with memory chips optimized to read large sequential blocks of data which is significantly faster than random access. In order to implement special graphics effects like texture filtering, new programmable units were added to the GPU. Their capabilities were improving with each new generation. Their processing capability grew from only dozens of instructions into sophisticated one which allows to run the same code as usual CPU. They kept their advantage which is parallelism. It is usual to run thousands of threads concurrently

on one GPU. This is two orders higher in comparison with nowadays multicore CPU. While the peak performance of todays GPUs is estimated approximately to 1 TFlops, a four cores CPU peak performance is around 50 GFlops. As a result, we have a computing device better suited for numerical algorithms than the usual CPU. The main advantages of GPUs are

- higher memory bandwidth than common CPUs ($\approx 10\times$),
- higher level of parallelism than common CPUs ($\approx 100\times$).

It is no surprise that the researchers who need to compute complex simulations started to be interested in GPUs. The first attempts to implement general algorithms – not related to computer graphics – were based on the programming in OpenGL [38]. Soon, more advanced tools were released by the main vendors of GPUs – Nvidia and ATI/AMD. Mainly CUDA (Compute Unified Device Architecture) by Nvidia has attracted a lot of interest and became very popular in the community interested in GPGPU (General-Purpose Computation on GPU). Currently, a lot of algorithms are implemented in CUDA. There are several articles about sparse-matrix vector multiplication [1, 2] and about iterative linear solvers [9, 10]. The GPU implementation of the Gauss–Seidel solver can be found in [29] and the tridiagonal solver in [44]. A GPU acceleration of FEM is presented in a series of articles [18, 19, 20]. The implementation of the Runge–Kutta method has been discussed in [40]. In this article, we demonstrate the implementation of the Merson modification of the Runge–Kutta method [42]. The Merson algorithm allows for the adaptive choice of the integration time step. This leads to a robust solver for the method of lines for the parabolic partial differential equations. We use this solver for the numerical approximation of the mean-curvature flow and the Willmore flow of graphs by the complementary finite volume method and the method of lines. They are non-linear second and fourth order geometric partial differential equations.

Contributions. We present a detailed description of the implementation of the Runge–Kutta–Merson solver in CUDA and we discuss several optimization techniques for large kernels. We compare the accuracy and the efficiency of the solver running on both CPU and GPU. We also test the numerical convergence in the single and the double precision arithmetic.

Organization. The article is organized as follows. In Section 2 we briefly explain two curvature dependent flows which we solve. Section 3 describes the numerical approximation. From this section, the reader should see the amount of computations necessary in each iteration of the Runge–Kutta solver. Both sections can be skipped by readers interested only in the CUDA implementation of the Runge–Kutta–Merson solver which can be found in Section 4. Here, we first show the CPU code and then we transform it step by step to CUDA. We explain only the necessary minimum of the CUDA knowledge. The results we obtained are discussed in Section 5.

2. MATHEMATICAL FORMULATION

In this section, we show the numerical approximation of the mean-curvature flow and the Willmore flow of graphs. Both problems originate from differential geometry and they have applications in phase transitions [22], image processing [8, 33, 34], surface restoration [13] and physics of elasticity [25, 41]. The mean-curvature flow minimizes the surface area functional

$$\mathcal{A}(\Gamma) = \int_\Gamma 1 \mathrm{d}S, \tag{1}$$

where Γ is a hypersurface in $\mathbb{R}^n$. Let Γ_0 be the initial hypersurface. We may generate a class of hypersurfaces $\Gamma(t)$ depending on the parameter t (it has the meaning of artificial time) such that $\Gamma(0) = \Gamma_0$ and either $\Gamma(t_0)$ for $t_0 > 0$ or $\lim_{t\to\infty}\Gamma(t)$ minimizes (1). Since we are interested in the change of shape of $\Gamma(t)$, we will study the motion of the points $\mathbf{x}(t) \in \Gamma(t)$. Only the projection to the normal direction at each point changes the shape of $\Gamma(t)$ and therefore we may omit the tangential velocity. One can show that (1) is minimized if the normal velocity V reads as

$$V = H \text{ on } \Gamma(t), \tag{2}$$

where H is the mean curvature of $\Gamma(t)$. The Willmore flow minimizes the Willmore functional

$$\mathcal{W}(\Gamma) = \int_\Gamma H^2 \mathrm{d}S, \tag{3}$$

for which the normal velocity reads [28]

$$V = -\Delta_\Gamma H - \frac{1}{2}H^3 + 2KH \text{ on } \Gamma(t). \tag{4}$$

Here, Δ_Γ is the Laplace–Beltrami operator and K is the Gauss curvature. Let Ω be a domain in $\mathbb{R}^2$ and let $\Gamma(t)$ be given as a graph of a function $\varphi : \Omega \times (0,\infty) \to \mathbb{R}$ such that

$$\Gamma(t) \equiv \{[\mathbf{x}, \varphi(\mathbf{x}, t)] \mid \mathbf{x} \in \Omega\}. \tag{5}$$

Often, we solve more general problem with an additional forcing term $F : \Gamma(t) \to \mathbb{R}$. If we denote ν the outer unit normal of the boundary $\partial\Omega$, $Q = \sqrt{1 + |\nabla\varphi|^2}$ and $\mathbb{P} = \mathbb{I} - \frac{\nabla\varphi}{Q} \otimes \frac{\nabla\varphi}{Q}$ then (2) and (4) with the additional forcing term F read as follows:

Problem 2.1. The graph formulation of the mean-curvature flow with the forcing term F, the Dirichlet boundary conditions and the initial condition φ_{ini} is the second order parabolic problem given by

$$\partial_t\varphi = -Q\nabla \cdot \left(\frac{\nabla\varphi}{Q}\right) + F(t) \text{ on } \Omega \times (0,T], \tag{6}$$

$$\varphi|_{t=0} = \varphi_{\text{ini}} \text{ on } \Omega, \tag{7}$$

$$\varphi = g \text{ on } \partial\Omega. \tag{8}$$

The graph formulation of the mean-curvature flow with the forcing term F, the Neumann boundary conditions and the initial condition φ_{ini} is the second order parabolic problem given by (6)–(7) and

$$\partial_\nu \varphi = 0 \text{ on } \partial\Omega. \tag{9}$$

Problem 2.2. The graph formulation of the Willmore flow with the forcing term F, the Dirichlet boundary conditions and the initial condition φ_{ini} is a fourth order parabolic problem given by

$$\begin{aligned} \partial_t \varphi &= -Q\nabla \cdot \left(\frac{1}{Q}\mathbb{P}\nabla w - \frac{1}{2}\frac{w^2}{Q^3}\nabla\varphi \right) + F(t) \text{ on } \Omega \times (0,T], && (10)\\ w &= Q\nabla \cdot \left(\frac{\nabla\varphi}{Q} \right) \text{ on } \Omega \times [0,T], && (11)\\ \varphi\,|_{t=0} &= \varphi_{\text{ini}} \text{ on } \Omega, && (12)\\ \varphi &= g, w = 0 \text{ on } \partial\Omega. && (13) \end{aligned}$$

The graph formulation of the Willmore flow with the forcing term F, the Neumann boundary conditions and the initial condition φ_{ini} is the fourth order parabolic problem given by (10)–(12) and

$$\partial_\nu \varphi = 0, \partial_\nu w = 0 \text{ on } \partial\Omega. \tag{14}$$

For some theoretical results concerning (2) and (4), we refer to [11, 12, 14, 15, 16, 17, 26, 27, 30, 31, 32, 35, 39].

3. NUMERICAL APPROXIMATION

To approximate (2.1) and (2.2) numerically we firstly discretize the equations in space by the complementary finite volumes method. This method has been successfully used in [7, 23, 34, 36, 43]. In the second step we will proceed to the discretization in time by the method of lines. We assume that $\Omega \equiv (0, L_1) \times (0, L_2)$. Let h_1, h_2 be space steps such that $h_1 = \frac{L_1}{N_1}$ and $h_2 = \frac{L_2}{N_2}$ for some $N_1, N_2 \in \mathbb{N}^+$. We define the numerical grid, its closure and its boundary as

$$\begin{aligned} \omega_h &= \{(ih_1, jh_2) \mid i = 1 \dots N_1 - 1, j = 1 \dots N_2 - 1\}, && (15)\\ \overline{\omega}_h &= \{(ih_1, jh_2) \mid i = 0 \dots N_1, j = 0 \dots N_2\},\\ \partial\omega_h &= \overline{\omega_h} \setminus \omega_h. \end{aligned}$$

We define the projection operator $\mathcal{P}_h : C\left(\overline{\Omega}_h\right) \to \overline{\omega}$ as

$$\mathcal{P}_h\left(\varphi\right)_{ij} := \varphi^h_{ij} := \varphi\left(ih_1, jh_2\right). \tag{16}$$

The details of the space discretization (2.1) and (2.2) can be found in [36, 37]. W ith the discretization formulas from the appendix (7) we may write the numerical schemes.

Scheme 3.1. The complementary finite volume semi-discrete numerical scheme for the mean-curvature flow of graphs with the zero forcing term $F(t)$ and the Dirichlet boundary conditions takes the following form

$$\frac{\mathrm{d}}{\mathrm{d}t}\varphi^h_{ij} = Q^h_{ij}\left(\frac{\varphi^h_{i+1j}-\varphi^h_{ij}}{h_1^2 Q^h_{ij,i+1j}} + \frac{\varphi^h_{ij+1}-\varphi^h_{ij}}{h_2^2 Q^h_{ij,ij+1}} - \frac{\varphi^h_{ij}-\varphi^h_{i-1j}}{h_1^2 Q^h_{ij,i-1j}} - \frac{\varphi^h_{ij}-\varphi^h_{ij-1}}{h_2^2 Q^h_{ij,ij-1}}\right) \text{ on } \omega_h, \tag{17}$$

$$\varphi^h_{ij}\mid_{t=0} = \mathcal{P}_h\left(\varphi_{\mathrm{ini}}\right)_{ij} \text{ on } \overline{\omega}_h, \tag{18}$$

$$\varphi^h_{ij} = g_{ij} \text{ on } \partial\omega_h,$$

where Q^h_{ij} is given by (49) and $Q^h_{ij,i+1j}$, $Q^h_{ij,ij+1}$, $Q^h_{ij,i-1j}$ and $Q^h_{ij,ij-1}$ are given by (36)–(39). The complementary finite volume semi-discrete numerical scheme for the mean-curvature flow of graphs with the Neumann boundary conditions is given by (17)–(18) together with

$$\varphi^h_{0j} = \varphi^h_{1j} \quad \text{and} \quad \varphi^h_{N_1 j} = \varphi^h_{N_1-1j} \text{ for } j = 0,\ldots,N_2, \tag{19}$$

$$\varphi^h_{i0} = \varphi^h_{i1} \quad \text{and} \quad \varphi^h_{iN_2} = \varphi^h_{iN_2-1} \text{ for } i = 0,\ldots,N_1. \tag{20}$$

Scheme 3.2. The complementary finite volume semi-discrete numerical scheme for the Willmore flow of graphs with the zero forcing term $F(t)$ and the Dirichlet boundary conditions takes the following form

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}t}\varphi^h_{ij} = {} & Q^h_{ij}\left[\frac{1}{h_1}\left(\mathbb{E}^h_{11,ij,i+1j}\partial^h_{x_1} w^h_{ij,i+1j} + \mathbb{E}^h_{12,ij,i+1j}\partial^h_{x_2} w^h_{ij,i+1j}\right)\right.\\
& + \frac{1}{h_2}\left(\mathbb{E}^h_{21,ij,ij+1}\partial^h_{x_1} w^h_{ij,ij+1} + \mathbb{E}^h_{22,ij,ij+1}\partial^h_{x_2} w^h_{ij,ij+1}\right)\\
& - \frac{1}{h_1}\left(\mathbb{E}^h_{11,ij,i-1j}\partial^h_{x_1} w^h_{ij,i-1j} + \mathbb{E}^h_{12,ij,i-1j}\partial^h_{x_2} w^h_{ij,i-1j}\right)\\
& - \frac{1}{h_2}\left(\mathbb{E}^h_{21,ij,ij-1}\partial^h_{x_1} w^h_{ij,ij-1} + \mathbb{E}^h_{22,ij,ij-1}\partial^h_{x_2} w^h_{ij,ij-1}\right)\\
& - \frac{1}{h_1}\left(\frac{1}{2}\frac{\left(w^h_{ij,i+1j}\right)^2}{\left(Q^h_{ij,i+1j}\right)^3}\partial^h_{x_1}\varphi^h_{ij,i+1j} - \frac{1}{2}\frac{\left(w^h_{ij,i-1j}\right)^2}{\left(Q^h_{ij,i-1j}\right)^3}\partial^h_{x_1}\varphi_{ij,i-1j}\right)\\
& \left. - \frac{1}{h_2}\left(\frac{1}{2}\frac{\left(w^h_{ij,ij+1}\right)^2}{\left(Q^h_{ij,ij+1}\right)^3}\partial^h_{x_2}\varphi_{ij,ij+1} - \frac{1}{2}\frac{\left(w^h_{ij,ij-1}\right)^2}{\left(Q^h_{ij,ij-1}\right)^3}\partial^h_{x_2}\varphi_{ij,ij-1}\right)\right]
\end{aligned} \tag{21}$$

$$w^h_{ij} = Q^h_{ij}\left(\frac{\varphi^h_{i+1j}-\varphi^h_{ij}}{h_1^2 Q^h_{ij,i+1j}} + \frac{\varphi^h_{ij+1}-\varphi^h_{ij}}{h_2^2 Q^h_{ij,ij+1}} - \frac{\varphi^h_{ij}-\varphi^h_{i-1j}}{h_1^2 Q^h_{ij,i-1j}} - \frac{\varphi^h_{ij}-\varphi^h_{ij-1}}{h_2^2 Q^h_{ij,ij-1}}\right) \text{ on } \omega_h, \tag{22}$$

$$\varphi^h_{ij}\mid_{t=0} = \mathcal{P}_h\left(\varphi_{\mathrm{ini}}\right)_{ij} \text{ on } \overline{\omega}_h, \tag{23}$$

$$\varphi^h_{ij} = g_{ij} \text{ and } w^h_{ij} = 0 \text{ on } \partial\omega_h,$$

where Q^h_{ij} is given by (49) and $Q^h_{ij,i+1j}$, $Q^h_{ij,ij+1}$, $Q^h_{ij,i-1j}$ and $Q^h_{ij,ij-1}$ are given

by (36)–(39), $\mathbb{E}^h_{mn,ij,\bar{i}\bar{j}}$ for $m,n = 1,2$ is given by (53), $w^h_{ij,\bar{i}\bar{j}}$ by (51)–(52) and as (44)–(47). $\partial^h_{x_1}\varphi^h_{ij,\bar{i}\bar{j}}$ and $\partial^h_{x_2}\varphi^h_{ij,\bar{i}\bar{j}}$ are approximated by (40)–(43).

The complementary finite volume semi-discrete numerical scheme for the Willmore flow of graphs with the Neumann boundary conditions is given by (21)–(23), (19)–(20) together with

$$w^h_{1,j} = w^h_{0,j} \quad \text{and} \quad w^h_{N_1,j} = w^h_{N_1-1,j} \text{ for } j = 0,\dots,N_2, \tag{24}$$

$$w^h_{i,1} = w^h_{i,0} \quad \text{and} \quad w^h_{i,N_2} = w^h_{i,N_2-1} \text{ for } i = 0,\dots,N_1. \tag{25}$$

Remark 3.3. After the discretization in space we have a system of ordinary differential equations of the form

$$\frac{\mathrm{d}u^h_{ij}}{\mathrm{dt}} = f\left(t, u^h\right)_{ij}, \tag{26}$$

where $f\left(t, u^h\right)_{ij}$ is given by the right-hand sides of (17) and (21). We solve the system by the Runge–Kutta method. It is the explicit time discretization known as the method of lines.

4. THE CUDA IMPLEMENTATION OF THE RUNGE–KUTTA–MERSON SOLVER

4.1. The Runge–Kutta–Merson solver for the Method of Lines

The advantage of explicit schemes is their high accuracy and easier implementation in comparison with semi-implicit or fully-implicit schemes involving solvers of linear or non-linear systems. The disadvantage is that they require significantly smaller time steps. It means that the solver must perform more iterations. In each iteration, the right-hand side f is evaluated. As a result, the explicit solvers can be computationally more intensive. This makes them good candidates for the implementation on the GPU.

The fourth order Runge–Kutta solvers were successfully used in many articles [3, 4, 5, 6]. The Merson solver [42] belongs to this class of solvers. Moreover, it offers the automatic choice of time step which makes the solver more robust. We will solve the system of ordinary differential equations (26). The Runge–Kutta–Merson solver consists of the following steps:

Algorithm 4.1. The explicit Runge–Kutta–Merson solver (Vitásek [42]) consists of the following steps:

1. Compute the grid functions k^1_{ij}, k^2_{ij}, k^3_{ij}, k^4_{ij}, k^5_{ij} as:

$$\begin{aligned}
k^1_{ij} &:= \tau f\left(t, u^h\right)_{ij} \\
k^2_{ij} &:= \tau f\left(t + \frac{1}{3}\tau, u^h + \frac{1}{3}k^1\right)_{ij} \\
k^3_{ij} &:= \tau f\left(t + \frac{1}{3}\tau, u^h + \frac{1}{6}k^1 + \frac{1}{6}k^2\right)_{ij} \\
k^4_{ij} &:= \tau f\left(t + \frac{1}{2}\tau, u^h + \frac{1}{8}k^1 + \frac{3}{8}k^3\right)_{ij} \\
k^5_{ij} &:= \tau f\left(t + \tau, u^h + \frac{1}{2}k^1 - \frac{3}{2}k^3 + 2k^4\right)_{ij},
\end{aligned}$$

   for $i = 0, \dots N_1$ and $j = 0, \dots, N_2$.

2. Evaluate the approximation error for the current time step τ as

$$e := \max_{\substack{i=0,\dots,N_1 \\ j=0,\dots,N_2}} \frac{1}{3}\left|\frac{1}{5}k^1_{ij} - \frac{9}{10}k^3_{ij} + \frac{4}{5}k^4_{ij} - \frac{1}{10}k^5_{ij}\right|. \tag{27}$$

3. If this error is smaller than given tolerance ϵ, update u^h as:

$$u^h_{ij} := u^h_{ij} + \frac{1}{6}\left(k^1_{ij} + 4k^4_{ij} + k^5_{ij}\right), \tag{28}$$

   for $i = 0, \dots N_1$, $j = 0, \dots, N_2$ and set

$$t := t + \tau.$$

4. Independently on the previous condition update τ as:

$$\tau := \min\left\{\tau \cdot \frac{4}{5}\left(\frac{\epsilon}{e}\right)^{\frac{1}{5}}, T - t\right\}. \tag{29}$$

5. Repeat the whole process with the new τ i.e. go to step 1.

4.2. Implementation of the Runge–Kutta–Merson solver on CPU

The implementation of the algorithm (4.1) in the C language on the CPU reads as follows:

```
1  void RungeKuttaMersonCPU( double *u_ini, double final_time )
2  {
3     // compute the degrees of freedom
4     const int N = N1 * N2;
5
6     // allocate the numerical grids
7     double *u, *k1, *k2, *k3, *k4, *k5, *k;
```

```
8     u = ( double* ) malloc( sizeof( double ) * N );
9     k1 = ( double* ) malloc( sizeof( double ) * N );
10    ...
11
12    // copy the initial condition to u
13    memcpy( u_ini, u, sizeof( double ) * N );
14
15    double t = 0;
16    double tau = tau_0;
17
18
19
20
21
22    while( t<final_time )
23    {
24       // compute the grid functions k1, k2, k3, k4, k5
25       EvaluateRHS( t, u, k1 );
26       for( int i = 0; i<N; i++ )
27          k[i] = u[i] + tau/3.0 * k1[i];
28       EvaluateRHS( t + tau/3.0, k, k2 );
29       ...
30
31       // compute the error with given tau
32       double e = 0.0;
33       for( int i = 0; i<N; i++ )
34          e = Max(e, tau/3.0 *
35                   (0.2*k1[i] - 0.9*k3[i] + 0.8*k4[i] - 0.1*k5[i]));
36
37       // if e is small enough proceed to the next time level
38       if( e<epsilon )
39       {
40          for( i = 0; i<N; i++ )
41             u[i] = u[i]+tau/6.0*( k1[i] + 4.0*k4[i] + k5[i] );
42          t=t+tau;
43       }
44
45       // recompute the new time step
46       tau=Min( 4.0/5.0*tau*pow( epsilon/e, 0.2 ), T-t );
47    }
48
49    // copy the result to the grid with the initial data
50    memcpy( u, u_ini, sizeof( double ) * N );
51
52    // free the allocated memory
53    memfree( u );
54    memfree( k1 );
55    ...
56 }
```

4.3. Implementation of the Runge–Kutta–Merson method in CUDA

Here is what we need for the implementation of our algorithm on the GPU in CUDA:

1. allocate the numerical grids on the CUDA device

2. copy the initial data from the host system (CPU memory) to the CUDA device

3. evaluate the weighted sum of the grid functions $u^h + \frac{1}{3}k^1$, $u^h + \frac{1}{6}k^1 + \frac{1}{6}k^2$, $u^h + \frac{1}{8}k^1 + \frac{3}{8}k^3$ and $u^h + \frac{1}{2}k^1 - \frac{3}{2}k^3 + 2k^4$ in the first step and $u^h_{ij} := u^h_{ij} + \frac{1}{6}\left(k^1_{ij} + 4k^4_{ij} + k^5_{ij}\right)$ in the third step of the Algorithm 4.1,

4. evaluate the right-hand side $f\left(t, u^h\right)_{ij}$ in the first step of the Algorithm 4.1

5. evaluate the maximum $e := \max_{\substack{i=0,\ldots,N_1\\ j=0,\ldots,N_2}} \frac{1}{3}\left|\frac{1}{5}k^1_{ij} - \frac{9}{10}k^3_{ij} + \frac{4}{5}k^4_{ij} - \frac{1}{10}k^5_{ij}\right|$.

The algorithm reads as follows:

```
1  void RungeKuttaMersonCUDA( double *u_ini, double final_time )
2  {
3     // compute the degrees of freedom
4     const int N = N1 * N2;
```

The allocation of memory on the GPU device in CUDA is done using the function `cudaMemalloc` [45]:

```
6     // allocate the numerical grids on CUDA device
7     double *u, *k1, *k2, *k3, *k4, *k5, *k;
8     cudaMemalloc( ( void** ) &u, sizeof( double ) * N );
9     cudaMemalloc( ( void** ) &k1, sizeof( double ) * N );
10    ...
```

To copy the initial data from the CPU memory to the global memory of the GPU we use the function `cudaMemcpy` [45]:

```
12    // copy the initial condition to u
13    cudaMemcpy( u, u_ini, sizeof( double ) * N, cudaMemcpyHostToDevice );
```

We set the necessary parameters:

```
15    double t = 0;
16    double tau = tau_0;
```

To evaluate the right-hand side of (26) and the weighted sums of the numerical functions $k^1, \ldots, k^5$, we start one CUDA thread for each node (i, j) of the given mesh function. Totally we run $N = (N_1 + 1)(N_2 + 1)$ threads concurrently. In CUDA, the threads are grouped into *blocks* and the blocks are grouped into *grids*. All threads running in the same block share the fast shared memory through which they can pass data to each other. The threads belonging to one block can be synchronized by an explicit command. Since there can be at most 512 threads in one block, we usually need more than one block of threads. The number of blocks, which is the smallest integer not less than `N / desBlockSize`, is given on the line 19.

```
18    const int desBlockSize = 128;
19    const int gridSize = N / desBlockSize + ( N % desBlockSize != 0 );
20    dim3 gridDim( gridSize ), blockDim( desBlockSize );
```

We may start now the main loop

```
21    while( t<final_time )
22    {
23       // compute the grid functions k1, k2, k3, k4, k5
24       EvaluateRHS<<< gridDim, blockDim >>>( t, u, k1 );
```

On the line 24, we call a *CUDA kernel*, a function running on the GPU processed by N threads concurrently. Its implementation for the right-hand side of (17) and (21) will be discussed later. We demonstrate the concept on the next kernel

```
26       // Compute k[i] = u[i] + tau/3.0 * k1[i] for i=0..N-1
27       EvaluateK2Argument<<< gridDim, blockDim >>>( N, tau, u, k1, k );
28       EvaluateRHS( t + tau/3.0, k, k2 );
```

The code for `EvaluateK2Argument` reads as follows:

```
101 __global__ void EvaluateK2Argument( const int N, const double tau,
102                                     const double* u, const double* k1,
103                                     double* k )
104 {
105    int i = blockIdx.x * blockDim.x + threadIdx.x;
106    if( i < N )
107       k[ i ] = u[ i ] + tau * ( 1.0 / 3.0 * k1[ i ] );
108 }
```

The word `__global__` indicates that the function `EvaluateK2Argument` is a CUDA kernel. The parameters `const int N, const double tau` must be passed as a value not as a reference. It is because they reside in the host memory (CPU) and they must be copied to the device memory (GPU). The pointers `const double* u, const double* k1, double* k` point to the mesh functions u, k^1 and k already allocated on the GPU. On the line 105, we compute the ID i of the current thread. Since there are generally more then N threads running (if $N = 129$ and the block size is 128 we must run 2 blocks and we have 256 threads) we check whether $i < N$ on the line 106. On the line 107, we perform the main computation. The next kernel will not start until the current one has finished. This is an important synchronization in our algorithm. The main loop of the solver continues as follows:

```
31       // compute the error with given tau
32
33
34       double e = ComputeE(tau, k1, k3, k4, k5 );
```

The function `ComputeE` computes the local error e_{ij} and it performs the parallel reduction [24]. We run 128 threads per block to reduce at most 2048 elements. It means that there are $N/2048$ blocks in the grid. Each thread sequentially reduces 16 elements and then the parallel reduction with logarithmic complexity is performed. The result is stored in the global memory in an array having the same number of elements as the number of blocks in the grid (each block stores one number – the result of the reduction). In the next step, we run the same kernel again with $N/2048$ elements. After the last step we have only one number stored on the GPU in the variable `device_e` from which it is then copied to the host variable `e`. The rest of the code is straightforward.

```
37        // if e is small enough proceed to the next time level
38        if( e<epsilon )
39           ComputeNewU<<< gridDim, blockDim >>>( tau, u, k1, k4, k5 );
40
41           t=t+tau;
42
43
44
45        // recompute the new time step
46        tau=Min( 4.0/5.0*tau*pow( epsilon/e, 0.2 ), T-t );
47     }
48
49     // copy the result to the grid with the initial data
50     cudaMemcpy( u, u_ini, sizeof( double ) * N, cudaMemcpyDeviceToHost );
51
52     // free the allocated memory
53     cudaFree( u );
54     cudaFree( k1 );
55     ...
56  }
```

We would like to comment on the evaluation of the right-hand side (17) or (21). The node (i, j) is mapped to global memory through bijection $I(i, j) = iN_2 + j$, where $i = 0, \dots, N_1$ and $j = 0, \dots, N_2$. The same bijection maps nodes and threads. We employ N threads splitted into one dimensional blocks. The node coordinates (i, j) are extracted using the following code

```
201 const int ij = blockIdx.x * blockDim.x + threadIdx.x;
202 const int i = ij / N2;
203 const int j = ij % N2;
```

To get (17) in one interior node, we start a new kernel and we fetch φ_{ij} and its 8 neighbors to the shared memory. This grid function is binded to a texture and so the reading is cached. Then, we compute (36)–(47) and (49). It takes 36 additions, 21 multiplications (we precompute the values $1/h_1$ and $1/h_2$), 8 absolute values and 4 square roots, i. e. 69 FLOPs (FLoating-point OPerations). Finally, we evaluate (48) which requires 4 multiplications (h_1^2 and h_2^2 are also precomputed), 4 divisions and 7 additions. At the end, we store the value $Q_{ij}H_{ij}$, i. e. 1 multiplication. In total, we have 85 FLOPs per 10 (coalesced) global memory accesses – 9 readings and 1 writing. The arithmetic intensity, which is defined as the ratio of operations to memory access, is 8.5.

To evaluate (21), we first compute w_{ij} using the kernel for (17). Then, we start a new kernel and read φ_{ij}, its 8 neighbors and w_{ij} with its 8 neighbors (W_{ij} is also binded to a texture for cached reading). We recompute $Q_{ij,i+1,j}$, $Q_{ij,ij+1}$,$Q_{ij,i-1,j}$ and $Q_{ij,ij-1}$. It is faster than storing them in the global memory in the first kernel and rereading now. It takes 28 additions, 20 multiplications, 4 square roots and 8 absolute values. Then, we compute (40)–(47) and (51)–(53). It takes 54 additions, 76 multiplications, 16 divisions and 8 absolute values and 4 roots of square. Totally, it is 82 additions, 96 multiplications, 16 divisions, 16 absolute values and 8 square roots. This yields 218 FLOPs per 18 global memory readings and 1 writing. The arithmetic intensity is 11.5.

Table 1. This table show the multiprocessor occupancy and the global memory throughput. In the first line, there are kernels computing the grid functions $k_1, \ldots, k_5$ and the kernel for updating u. In the second line, there is a reduction of the integration error e. The last two lines show kernels evaluating the right hand side f of (26) and the forcing terms F_{WC} and F_W.

Kernel	Occupancy	Global memory throughput
$k_1 \ldots k_5$, u	1	100-105 GB/s
e	1	100 GB/s
f mean-curvature flow	0.33	85 GB/s
f Willmore flow	0.33	55 GB/s
F_{WC}, F_W	0.2	14 GB/s

Another important indicator is a multiprocessors occupancy. It is percentage of time spent by the computing. The occupancy depends on the number of registers used by one thread, shared memory allocated for one block and the number of threads in one block. The results obtained on the numerical grid with 512×512 nodes are summarized in the Table 1. This table also shows the global memory throughput. All kernels used for the Runge–Kutta-Merson solver achieve occupancy 100% and the global memory throughput over 100 GB/s. However, most of the time, 97 %, is spent in kernels evaluating (6), (11) and (10). These kernels attain only 33 % resp. 20 % occupancy and 14 to 85 GB/s memory throughput. The limiting factor is the number of registers [1]. Moreover, for large kernels some variables may not fit into limited shared memory on the multiprocessor (it is 16 kB on Nvidia GeForce GTX 280). The CUDA PTX compiler may decide to store some variables in *the local memory* of the thread which resides in the global memory of the device. Accessing these variables is very slow. Passing a parameter `--ptxas-options=` $-v$ to `nvcc` compiler evokes printing information about the memory allocation. One should try to reduce bytes denoted by `lmem`. It can be achieved by reducing the number of the kernel variables by avoiding unnecessary variables or variables which are used only once and which just fetch data from the global memory. Note that the kernel code is also stored in the shared memory. Reducing the code size can therefore help. By this techniques we optimized our algorithm to run 25 % faster.

The disadvantage of the CUDA implementation of the Runge–Kutta–Merson solver is that it does not work as a black box as the matrix solvers do. The user must write his own CUDA kernel. The efficiency of the solver then strongly depends on the efficiency of this kernel. We now summarize several rules for writing the kernel.

- The coalesced accesses to the global memory are essential in reducing the latency of the global memory of GPU [45]. It is fulfilled easily using the natural mapping between numerical grid nodes and CUDA threads as we discussed above.

[1] The kernels were optimized using the CUDA Occupancy Calculator.

- It is better to recompute some quantities than to store them in the global memory.
- Elimination of unnecessary variables in large kernels may reduce the use of the local memory and registers of the multiprocessor. This can improve the occupancy.

5. COMPUTATIONAL RESULTS

To measure the speed-up of the Runge–Kutta–Merson solver implemented in CUDA and to compare its accuracy on both the GPU and the CPU, we will evaluate *the experimental order of convergence* (EOC). It shows how the approximation error depends on the space step of the numerical grid ω_h. To do this we take an analytical solution of (2.1) and (2.2) and compare it with the numerical approximation. We set force terms F_{MC} (resp. F_W) to match the solution

$$\zeta\left(t,\mathbf{x}\right) := \cos\left(\pi t\right)\frac{1}{r^{2n}}\left(x^n - r^n\right)\left(y^n - r^n\right)\exp\left(-\sigma\left(x^2+y^2\right)\right) \text{ on } \Omega\times[0,T]. \quad (30)$$

The forcing terms F_{MC} and F_W are evaluated exactly (see [37]). For given T, we evaluate the errors of the numerical approximation in the norms of the spaces $L_1\left(\Omega;[0,T]\right)$, $L_2\left(\Omega;[0,T]\right)$ and $L_\infty\left(\Omega;[0,T]\right)$ resp. their approximations

$$\left\|\varphi^h - \mathcal{P}_h\left(\zeta\right)\right\|^{h,\theta}_{L_1(\omega_h;[0,T])} := \sum_{k=0}^{M}\theta\sum_{i=0,j=0}^{N_1,N_2}\left|\varphi^h_{ij}\left(k\theta\right) - \mathcal{P}_h\left(\zeta\right)_{ij}\left(k\theta\right)\right| h_1h_2, \quad (31)$$

$$\left\|\varphi^h - \mathcal{P}_h\left(\zeta\right)\right\|^{h,\theta}_{L_2(\omega_h;[0,T])} := \left(\sum_{k=0}^{M}\theta\sum_{i=0,j=0}^{N_1,N_2}\left(\varphi^h_{ij}\left(k\theta\right) - \mathcal{P}_h\left(\zeta\right)_{ij}\left(k\theta\right)\right)^2 h_1h_2\right)^{\frac{1}{2}}, \quad (32)$$

$$\left\|\varphi^h - \mathcal{P}_h\left(\zeta\right)\right\|^{h,\theta}_{L_\infty(\omega_h;[0,T])} := \max_{k=0,\dots,M}\max_{\substack{i=0,\dots,N_1\\ j=0,\dots,N_2}}\left|\varphi^h_{ij}\left(k\theta\right) - \mathcal{P}_h\left(\zeta\right)_{ij}\left(k\theta\right)\right|, \quad (33)$$

for $\theta = T/M$. For two numerical solutions φ^{h_1} and φ^{h_2} obtained by the discretization with the space steps h_1 and h_2, we compute the approximation errors Err_{h_1} and Err_{h_2} in one of the norms (31)–(33). Then, the experimental order of convergence is defined as

$$EOC\left(Err_{h_1}, Err_{h_2}\right) := \frac{\log\left(Err_{h_1}/Err_{h_2}\right)}{\log\left(h_1/h_2\right)}. \quad (34)$$

All computations were done on the Intel Core 2 Quad CPU with 4 cores, 4 MB cache memory running at 2.66 GHz and Nvidia Geforce GTX 280 with CUDA 2.3 and GNU/gcc 4.3 installed on the 64-bit GNU/Linux Ubuntu 10.04. The CPU computations were single threaded but also parallelized by OpenMP standard. The CPU code was not explicitly optimized to use SSE instructions. We set the domain $\Omega \equiv [-4,4]^2$ and the stop time $T = 0.1$. The error as well as the experimental order of convergence for both the CPU and the GPU are the same. The experimental

Table 2. The experimental order of convergence (EOC) for the **mean-curvature flow of graphs** on the CPU and the GPU in the **single** precision. N denotes meshes of the numerical grid.

Meshes	$\|\cdot\|^{h,\tau}_{L_1(\omega_h;[0,T])}$		$\|\cdot\|^{h,\tau}_{L_2(\omega_h;[0,T])}$		$\|\cdot\|^{h,\tau}_{L_\infty(\omega_h;[0,T])}$	
	Error	EOC	Error	EOC	Error	EOC
16^2	7.1e-03		9.0e-03		4.3e-02	
		2		**2.1**		**2.1**
32^2	1.5e-03		1.9e-03		8.5e-03	
		2		**1.9**		**1.8**
64^2	3.9e-04		4.9e-04		2.4e-03	
		2		**2**		**2**
128^2	9.8e-05		1.2e-04		6.1e-04	
		2		**2**		**2**
256^2	2.5e-05		3.1e-05		1.5e-04	
		2		**2**		**2**
512^2	6.2e-06		7.8e-06		3.9e-05	
		1.9		**1.9**		**1.8**
1024^2	1.7e-06		2.1e-06		1.1e-05	

order of convergence equals 2. The Table 2 shows results for the single precision arithmetic while the Tables 3 and 4 in the double precision. We see that the numerical approximation of the mean-curvature flow of graphs in the single and the double precision exhibits approximately the same accuracy.

The Tables 5–7 show the sequential CPU time, the parallel CPU time and the GPU time together with the number of gigaflops and the speed-up. The last columns show the speed-up of the GPU relative to the sequential code resp. to the parallel OpenMP code. The power of the GPU is evident especially on large meshes. GPU profits from the single precision arithmetic for which it is equipped with more computing units. The speed-up is almost 17. Applications in image processing can profit from it. The speed-up in the double precision is up to 7. With the CPU, we have achieved 3 GFLOPS (Giga FLoating-point Operations Per Second) performance. Note, that our algorithm contains functions like `pow`, `exp`, `sin` or `sqrt` which we take as one FLOP. On GPU we get almost 50 GFLOPS in the single precision and 17 GFLOPS in the double precision.

The Figures 1 and 2 show the evolution of the initial surface given as a graph of the following function

$$\varphi\,|_{t=0} = \sin\left(3\pi\sqrt{x^2+y^2}\right) \quad \text{on } \Omega, \tag{35}$$

where $\Omega \equiv (-2,2)^2$. We set the Neumann boundary conditions (9) resp. (14) and the space step $h = 0.03125$, i. e., 128^2 meshes. The CPU and the GPU times together with the speed-up for the single and the double precision are in the Table 8. The approximation of the Willmore flow in the single precision was omitted since it does not give reasonable results.

Table 3. The experimental order of convergence (EOC) for the **mean-curvature flow of graphs** on the CPU and the GPU in the **double** precision. N denotes meshes of the numerical grid.

Meshes	$\|\cdot\|^{h,\tau}_{L_1(\omega_h;[0,T])}$		$\|\cdot\|^{h,\tau}_{L_2(\omega_h;[0,T])}$		$\|\cdot\|^{h,\tau}_{L_\infty(\omega_h;[0,T])}$	
	Error	EOC	Error	EOC	Error	EOC
16^2	7.1e-03		9.0e-03		4.3e-02	
32^2	1.5e-03	**2**	1.9e-03	**2.1**	8.5e-03	**2.1**
64^2	3.9e-04	**2**	4.9e-04	**1.9**	2.4e-03	**1.8**
128^2	9.8e-05	**2**	1.2e-04	**2**	6.1e-04	**2**
256^2	2.5e-05	**2**	3.1e-05	**2**	1.5e-04	**2**
512^2	6.2e-06	**2**	7.7e-06	**2**	3.8e-05	**2**
1024^2	1.6e-06	**1.9**	1.9e-06	**2**	9.6e-06	**2**

Table 4. The experimental order of convergence (EOC) for the **Willmore flow of graphs** on the CPU and the GPU in the **double** precision. N denotes meshes of the numerical grid.

N	$\|\cdot\|^{h,\tau}_{L_1(\omega_h;[0,T])}$		$\|\cdot\|^{h,\tau}_{L_2(\omega_h;[0,T])}$		$\|\cdot\|^{h,\tau}_{L_\infty(\omega_h;[0,T])}$	
	Error	EOC	Error	EOC	Error	EOC
16^2	1.4e-01		2.5e-01		1.3	
32^2	8.1e-02	**0.8**	1.5e-01	**0.7**	7.0e-01	**0.8**
64^2	5.8e-03	**3.8**	1.2e-02	**3.6**	7.1e-02	**3.3**
128^2	1.8e-03	**1.7**	3.8e-03	**1.7**	2.3e-02	**1.6**
256^2	4.5e-04	**2**	9.9e-04	**2**	6.1e-03	**1.9**

Table 5. The efficiency of the CUDA implementation demonstrated on the **mean-curvature flow of graphs** on the CPU and the GPU in the **single** precision. N denotes meshes of the numerical grid. The CPU and the GPU times are presented in seconds. The last column shows speed-up of the GPU implementation compared to sequential resp. parallel CPU code.

	1 core CPU		4 cores CPU			Nvidia GTX 280		
N	Time	GFlops	Time	GFlops	Speed-up	Time	GFlops	Speed-up
16^2	0.18	0.86	0.18	0.86	1	0.04	3.9	**4.5/4.5**
32^2	0.51	0.9	0.42	1.08	1.2	0.05	9.2	**10.2/8.5**
64^2	2	1.02	1.53	1.32	1.3	0.1	20.4	**20/15.4**
128^2	12	0.9	5.01	2.16	2.4	0.32	33.7	**37.5/15.6**
256^2	173	0.96	55.8	2.97	3.1	3.5	47.4	**49.4/15.9**
512^2	2537	0.91	746	3.1	3.4	49	47.1	**51.8/15.2**
1024^2	40869	0.9	12771	2.9	3.2	754	48.8	**54.2/16.9**

Table 6. The efficiency of the CUDA implementation demonstrated on the **mean-curvature flow of graphs** on the CPU and the GPU in the **double** precision. N denotes meshes of the numerical grid. The CPU and the GPU times are presented in seconds. The last column shows speed-up of the GPU implementation compared to sequential resp. parallel CPU code.

	1 core CPU		4 cores CPU			Nvidia GTX 280		
N	Time	GFlops	Time	GFlops	Speed-up	Time	GFlops	Speed-up
16^2	0.13	0.85	0.12	0.93	1.1	0.05	2.21	**2.6/2.4**
32^2	0.53	0.87	0.35	1.31	1.5	0.06	7.74	**8.9/5.9**
64^2	2.1	0.86	1.23	1.46	1.7	0.17	10.5	**12.3/7.2**
128^2	13	0.83	5.41	1.99	2.4	0.66	16.3	**19.7/8.2**
256^2	200	0.8	64.5	2.48	3.1	9.0	17.7	**22.2/7.16**
512^2	3272	0.76	934	2.66	3.5	138	17.4	**23/6.57**
1024^2	54187	0.72	16420	2.37	3.3	2213	17.2	**24/7.27**

Table 7. The efficiency of the CUDA implementation demonstrated on the **Willmore flow of graphs** on the CPU and the GPU in the **double** precision. N denotes meshes of the numerical grid. The CPU and the GPU times are presented in seconds. The last column shows speed-up of the GPU implementation compared to sequential resp. parallel CPU code.

	1 core CPU		4 cores CPU			Nvidia GTX 280		
N	Time	GFlops	Time	GFlops	Speed-up	Time	GFlops	Speed-up
16^2	0.28	0.9	0.12	2	2.2	0.19	1.26	**1.4/0.7**
32^2	7.4	0.87	2.38	2.7	3.1	1.44	4.6	**5.3/1.7**
64^2	492	0.8	136	2.9	3.6	38	9.6	**13/3.6**
128^2	30494	0.83	8241	3.1	3.7	1494	10.8	**20/5.4**
256^2	197424	0.81	51953	3.1	3.8	93031	17	**21/5.5**

Table 8. Comparison of the CPU and the GPU time for the evolution of the surface on the Figures 1 and 2 in the single and the double precision with 128^2 meshes. The CPU and the GPU times are presented in seconds. The last column shows speed-up of the GPU implementation compared to sequential resp. parallel CPU code.

Precision	CPU time	4 cores CPU time	Speed-up	GPU time	Speed-up
	Mean-curvature flow				
Single	14.3	4.08	3.5	1	**14.3/4.1**
Double	23.2	6.62	3.5	2	**11.6/3.3**
	Willmore flow				
Single	–	–	–	–	–
Double	152717	44354	3.4	11818	**12.9/3.8**

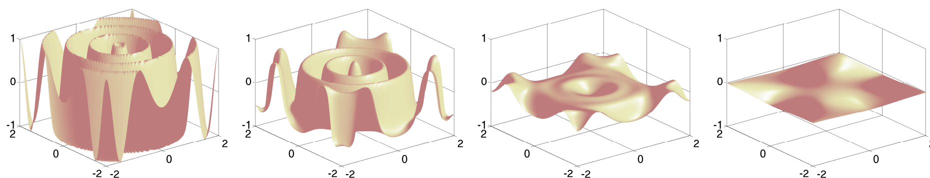

Fig. 1. Evolution of the surface given as a graph of (35) driven by the mean-curvature flow of graphs at times $t = 0$, $t = 0.025$, $t = 0.1$ and $t = 0.5$. The single and the double precision give the same results.

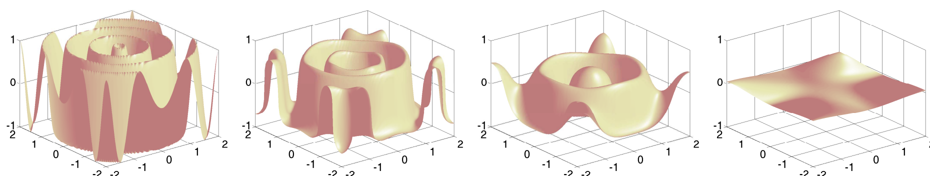

Fig. 2. Evolution of the surface given as a graph of (35) driven by the Willmore flow of graphs at times $t = 0$, $t = 0.01$, $t = 0.025$ and $t = 0.1$ in the double precision.

6. CONCLUSION

We have presented the CUDA implementation of the Runge–Kutta–Merson solver and use it for the numerical approximation of the curvature dependent flows by the method of lines. We obtained speed-up 17 in the single precision arithmetic and 7 in the double precision arithmetic. Advantages of this algorithm are automatic choice of the integration time step and relatively simple implementation. It is computationally more intensive in comparison with linear solvers and thus it profits more from the great performance of modern GPUs. Unfortunately, the number of the multiprocessor registers is a limiting factor for larger kernels. It is the reason why our kernels exploit only 33% of the GPU performance. We have also mentioned a few optimization techniques which might be useful for writing larger CUDA kernels with many variables. The source code for the CPU and the GPU implementation of the Runge–Kutta–Merson solver is freely available as a part of the Template Numerical Library (TNL) at http://geraldine.fjfi.cvut.cz/˜oberhuber/doku-wiki-tnl.

7. APPENDIX

In this appendix we show the discretization formulas. We set

$$Q^h_{ij,i+1j} = \sqrt{1+\left(\partial^h_{x_1}\varphi^h_{ij,i+1j}\right)^2+\left(\partial^h_{x_2}\varphi^h_{ij,i+1j}\right)^2}, \tag{36}$$

$$Q^h_{ij,ij+1} = \sqrt{1+\left(\partial^h_{x_1}\varphi^h_{ij,ij+1}\right)^2+\left(\partial^h_{x_2}\varphi^h_{ij,ij+1}\right)^2}, \tag{37}$$

$$Q^h_{ij,i-1j} = \sqrt{1+\left(\partial^h_{x_1}\varphi^h_{ij,i-1j}\right)^2+\left(\partial^h_{x_2}\varphi^h_{ij,i-1j}\right)^2}, \tag{38}$$

$$Q^h_{ij,ij-1} = \sqrt{1+\left(\partial^h_{x_1}\varphi^h_{ij,ij-1}\right)^2+\left(\partial^h_{x_2}\varphi^h_{ij,ij-1}\right)^2}, \tag{39}$$

for

$$\partial^h_{x_1}\varphi^h_{ij,i+1j} = \frac{\varphi^h_{i+1j}-\varphi^h_{ij}}{h_1}\quad,\quad \partial^h_{x_1}\varphi^h_{ij,i-1j} = \frac{\varphi^h_{ij}-\varphi^h_{i-1j}}{h_1}, \tag{40}$$

$$\partial^h_{x_2}\varphi^h_{ij,ij+1} = \frac{\varphi^h_{ij+1}-\varphi^h_{ij}}{h_2}\quad,\quad \partial^h_{x_2}\varphi^h_{ij,ij-1} = \frac{\varphi^h_{ij}-\varphi^h_{ij-1}}{h_2}, \tag{41}$$

and

$$\partial^h_{x_2}\varphi^h_{ij,i+1j} = \frac{\varphi^h_{ij,i+1j+1}-\varphi^h_{ij,i+1j-1}}{h_2}\ ,\ \partial^h_{x_2}\varphi^h_{ij,i-1j} = \frac{\varphi^h_{ij,i-1j+1}-\varphi^h_{ij,i-1j-1}}{h_2}, \tag{42}$$

$$\partial^h_{x_1}\varphi^h_{ij,ij+1} = \frac{\varphi^h_{ij,i+1j+1}-\varphi^h_{ij,i-1j+1}}{h_1}\ ,\ \partial^h_{x_1}\varphi_{ij,ij-1} = \frac{\varphi_{ij,i+1j-1}-\varphi_{ij,i-1j-1}}{h_1}, \tag{43}$$

where we denote

$$\varphi^h_{ij,i+1j+1} = \frac{1}{4}\left(\varphi^h_{ij}+\varphi^h_{i+1j}+\varphi^h_{ij+1}+\varphi^h_{i+1j+1}\right), \tag{44}$$

$$\varphi^h_{ij,i+1j-1} = \frac{1}{4}\left(\varphi^h_{ij}+\varphi^h_{i+1j}+\varphi^h_{ij-1}+\varphi^h_{i+1j-1}\right), \tag{45}$$

$$\varphi^h_{ij,i-1j+1} = \frac{1}{4}\left(\varphi^h_{ij}+\varphi^h_{i-1j}+\varphi^h_{ij+1}+\varphi^h_{i-1j+1}\right), \tag{46}$$

$$\varphi^h_{ij,i-1j-1} = \frac{1}{4}\left(\varphi^h_{ij}+\varphi^h_{i-1j}+\varphi^h_{ij-1}+\varphi^h_{i-1j-1}\right). \tag{47}$$

We approximate the mean curvature H as

$$H^h_{ij} \approx \left(\frac{\varphi^h_{i+1j}-\varphi^h_{ij}}{h_1^2 Q^h_{ij,i+1j}}+\frac{\varphi^h_{ij+1}-\varphi^h_{ij}}{h_2^2 Q^h_{ij,ij+1}}-\frac{\varphi^h_{ij}-\varphi^h_{i-1j}}{h_1^2 Q^h_{ij,i-1j}}-\frac{\varphi^h_{ij}-\varphi^h_{ij-1}}{h_2^2 Q^h_{ij,ij-1}}\right). \tag{48}$$

Setting

$$Q^h_{ij} = \frac{1}{4}\left(Q^h_{ij,i+1j}+Q^h_{ij,ij+1}+Q^h_{ij,i-1j}+Q^h_{ij,ij-1}\right), \tag{49}$$

$$w^h_{ij} = Q^h_{ij}H^h_{ij}, \tag{50}$$

$$w^h_{ij,i+1j} = \frac{1}{2}\left(w^h_{ij} + w^h_{i+1j}\right) \quad , \quad w^h_{ij,ij+1} = \frac{1}{2}\left(w^h_{ij} + w^h_{ij+1}\right), \tag{51}$$

$$w^h_{ij,i-1j} = \frac{1}{2}\left(w^h_{ij} + w^h_{i-1j}\right) \quad , \quad w^h_{ij,ij-1} = \frac{1}{2}\left(w^h_{ij} + w^h_{ij-1}\right), \tag{52}$$

$$\mathbb{E}^h_{ij,i+1j} = \frac{1}{Q_{ij,i+1j}} \begin{pmatrix} 1-\left(\partial_{x_1}\varphi^h_{ij,i+1j}\right)^2 & -\partial_{x_1}\varphi^h_{ij,i+1j}\partial_{x_2}\varphi^h_{ij,i+1j} \\ -\partial_{x_1}\varphi^h_{ij,i+1j}\partial_{x_2}\varphi^h_{ij,i+1j} & 1-\left(\partial_{x_2}\varphi^h_{ij,i+1j}\right)^2 \end{pmatrix},$$

$$\mathbb{E}^h_{ij,ij+1} = \frac{1}{Q_{ij,ij+1}} \begin{pmatrix} 1-\left(\partial_{x_1}\varphi^h_{ij,ij+1}\right)^2 & -\partial_{x_1}\varphi^h_{ij,ij+1}\partial_{x_2}\varphi^h_{ij,ij+1} \\ -\partial_{x_1}\varphi^h_{ij,ij+1}\partial_{x_2}\varphi^h_{ij,ij+1} & 1-\left(\partial_{x_2}\varphi^h_{ij,ij+1}\right)^2 \end{pmatrix},$$

$$\mathbb{E}^h_{ij,i-1j} = \frac{1}{Q_{ij,i-1j}} \begin{pmatrix} 1-\left(\partial_{x_1}\varphi^h_{ij,i-1j}\right)^2 & -\partial_{x_1}\varphi^h_{ij,i-1j}\partial_{x_2}\varphi^h_{ij,i-1j} \\ -\partial_{x_1}\varphi^h_{ij,i-1j}\partial_{x_2}\varphi^h_{ij,i-1j} & 1-\left(\partial_{x_2}\varphi^h_{ij,i-1j}\right)^2 \end{pmatrix},$$

$$\mathbb{E}^h_{ij,ij-1} = \frac{1}{Q_{ij,ij-1}} \begin{pmatrix} 1-\left(\partial_{x_1}\varphi^h_{ij,ij-1}\right)^2 & -\partial_{x_1}\varphi^h_{ij,ij-1}\partial_{x_2}\varphi^h_{ij,ij-1} \\ -\partial_{x_1}\varphi^h_{ij,ij-1}\partial_{x_2}\varphi^h_{ij,ij-1} & 1-\left(\partial_{x_2}\varphi^h_{ij,ij-1}\right)^2 \end{pmatrix},$$

$$\mathbb{E}^h_{ij,\bar{i}\bar{j}} = \begin{pmatrix} \mathbb{E}^h_{11,ij,\bar{i}\bar{j}} & \mathbb{E}^h_{12,ij,\bar{i}\bar{j}} \\ \mathbb{E}^h_{21,ij,\bar{i}\bar{j}} & \mathbb{E}^h_{22,ij,\bar{i}\bar{j}} \end{pmatrix}. \tag{53}$$

Approximating $\partial_{x_1} w^h_{ij,\bar{i}\bar{j}}$ and $\partial_{x_2} w^h_{ij,\bar{i}\bar{j}}$ in the same way as $\partial_{x_1}\varphi^h_{ij,\bar{i}\bar{j}}$ and $\partial_{x_2}\varphi^h_{ij,\bar{i}\bar{j}}$ by (40)–(43).

ACKNOWLEDGMENT

This work was partially supported by the Jindřich Nečas Center for Mathematical Modelling, Research center of the Ministry of Education of the Czech Republic LC06052, Research Direction Project of the Ministry of Education of the Czech Republic No. MSM6840770010 and Supercomputing Methods in Mathematical Modelling of Problems in Engineering and Natural Sciences, project of the Student Grant Agency of the Czech Technical University in Prague No. 283 OHK4-009/10 P3913.

(Received July 20, 2010)

REFERENCES

[1] M. M. Baskaran and R. Bordaweker: Optimizing Sparse-Vector Matrix Multiplication on Gpus. IBM Research Report RC24704, IBM 2009.

[2] N. Bell and M. Garland: Implementing sparse matrix-vector multiplication on throughput oriented processors. In Supercomputing'09, Nov. 2009.

[3] M. Beneš: Mathematical and computational aspects of solidification of pure substances. Acta Math. Univ. Comenian. *70* (2000), 123–151.

[4] M. Beneš: Mathematical analysis of phase-field equations with numerically efficient coupling terms. Interfaces and Free Boundaries *3* (2001), 201–221.

[5] M. Beneš: Diffuse-interface treatment of the anisotropic mean-curvature flow. Appl. Math. *80* (2003), 6, 437–453.

[6] M. Beneš: Phase Field Model of Microstructure Growth in Solidification of Pure Substances. PhD. Dissertation, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University, Prague 1997.

[7] M. Beneš, K. Mikula, T. Oberhuber, and D. Ševčovič: Comparison study for level set and direct Lagrangian methods for computing Willmore flow of closed plannar curves. Computing and Visualization in Science *12* (2009), 307–317.

[8] M. Bertalmio, V. Caselles, G. Haro, and G. Sapiro: Handbook of Mathematical Models in Computer Vision. PDE-Based Image and Surface Inpainting, Springer 2006, pp. 33–61.

[9] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder: Sparse matrix solvers on the gpu: Conjugate gradients and multigrid. ACM Trans. Graphics *22* (2003), 3, 917–924.

[10] L. Buatois, G. Caumon, and B. Levy: Concurrent number cruncher: a gpu implementation of a general sparse linear solver. Internat. J. Parallel Emerg. Distrib. Syst. *24* (2009), 3, 205–223.

[11] Y.-G. Chen, Y. Giga, and S. Goto: Uniqueness and existence of viscosity solutions of generalized mean curvature flow equations. J. Differential Geom. *33* (1991), 3, 749–786.

[12] U. Clarenz: The Wulff shape minimizes an anisotropic Willmore functional. Interfaces and Free Boundaries *6* (2004), 351–360.

[13] U. Clarenz, U. Diewald, G. Dziuk, M. Rumpf, and R. Rusu: A finite element method for surface restoration with smooth boundary conditions. Computer Aided Geometric Design *21* (2004), 5, 427–445.

[14] K. Deckelnick and G. Dziuk: Mathematical aspects of evolving interfaces. Lecture Notes in Math. *1812*, Numerical Approximation of Mean Curvature Flow of Graphs and Level Sets, Springer-Verlag, Berlin–Heidelberg 2003, pp. 53–87.

[15] G. Dziuk, E. Kuwert, and R. Schätzle: Evolution of elastic curves in $\mathbb{R}^n$: Existence and computation. SIAM J. Math. Anal.*41* (2003), 6, 2161–2179.

[16] L. C. Evans and J. Spruck: Motion of level sets by mean curvature II. Trans. Amer. Math. Soc. *330* (1993), 1, 321–332.

[17] Y. Giga: Surface evolution equations: A level set approach. Birkhauser Verlag 2006.

[18] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. H. Buijssen, M. Grajewski, and S. Turek: Exploring weak scalability for fem calculations on a gpu-enhanced cluster. Parallel Computing, Special issue: High-performance computing using accelerators *33* (2007), 685–699.

[19] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, and S. Turek: Using gpus to improve multigrid solver performance on a cluster. Internat. J. Comput. Sci. Engrg. *4* (2008), 1, 36–55.

[20] D. Göddeke, H. Wobker, R. Strzodka, J. Mohd-Yusof, P. McCormick, and S. Turek: Co-processor acceleration of an unmodified parallel solid mechanics code with feastgpu. Internat. J. Comput. Sci. Engrg. *4* (2009), 4, 254–269.

[21] A. Grama, A. Gupta, G. Karypis, and V. Kumar: Introduction to Parallel Computing. Pearson, Addison Wesley 2003.

[22] M. E. Gurtin: On the two-phase stefan problem with interfacial energy and entropy. Arch. Rational Mech. Anal. *96* (1986), 200–240.

[23] A. Handlovičová, K. Mikula, and F. Sgallari: Semi-implicit complementary volume scheme for solving level set like equations in image processing and curve evolution. Numer. Math. *93* (2003), 675–695.

[24] M. Harris: Optimizing parallel reduction in cuda. NVIDIA CUDA SDK 2007.

[25] W. Helfrich: Elastic properties of lipid bilayers: theory and possible experiments. Zeitschrift für Naturforschung *28* (1973), 693–703.

[26] G. Huisken: Flow by mean curvature of convex surfaces into spheres. J. Differential Geometry *20* (1984), 237–266.

[27] G. Huisken: Non-parametric mean curvature evolution with boudary conditions. J. Differential Geom. *77* (1988), 369–379.

[28] M. Kimura: Topics in mathematical modeling. Jindřich Nečas Center for Mathematical Modelling *4*, Lecture Notes, Geometry of Hypersurfaces and Moving Hypersurfaces in R^m for the Study of Moving Boundary Problems, Matfyzpress, Publishing House of Mathematics and Physics, Charles University in Prague 2008, pp. 39–94.

[29] J. Kruger and R. Westermann: Linear algebra operators for gpu implementation of numerical algorithms. ACM Trans. Graphics *22* (2003), 3, 908–916.

[30] E. Kuwert and R. Schätzle: Gradient flow for the Willmore functional. Comm. Anal. Geom. *10* (2003), 2, 307–340.

[31] E. Kuwert and R. Schätzle: The Willmore flow with small initial energy. J. Differ. Geom. *57* (2001), 409–441.

[32] U. F. Mayer and G. Simonett: Evolution equations: Applications to physics, industry, life scienses economics. Self-intersections for Willmore flow, Progress in nonlinear differential equations and their applications, Birkhäuser Verlag, Basel 2003, pp. 341–348.

[33] K. Mikula: Image processing with partial differential equations. In: Modern Methods in Scientific Computing and Applications (A. Bourlioux and M. Gander, eds.), NATO Science Ser. II *75*, Kluwer Academic Publishers, Dodrecht 2002, pp. 283–322.

[34] K. Mikula and A. Sarti: Parametric and geometric deformable models: An application in biomaterials and medical imagery. In: Parallel co-volume subjective surface method for 3D medical image segmentation *2*, 2007, pp. 123–160.

[35] J. C. C. Nitsche: On new results in the theory of minimal surfaces. Bull. Amer. Math. Soc. *71* (1965), 195–270.

[36] T. Oberhuber: Complementary finite volume scheme for the anisotropic surface diffusion flow. In: Proc. Algoritmy 2009 (A. Handlovičová, P. Frolkovič, K. Mikula, and D. Ševčovič, eds.), pp. 153–164.

[37] T. Oberhuber: Numerical Solution of Willmore Flow. PhD. Thesis, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, 2009.

[38] M. Pharr, ed.: GPU Gems 2: Programming Techniques for High-Performance Graphics and General–Purpose Computation. Addison-Wesley, 2005.

[39] G. Simonett: The Willmore flow near spheres. Differential and Integral Equations *14* (2001), 8, 1005–1014.

[40] V. Šimek, R. Dvořák, F. Zbořil, and J. Kunovský: Towards accelerated computation of atmospheric equations using CUDA. In: 11th Internat. Conf. on Computer Modelling and Simulation, pp. 449–454, 2009.

[41] S. Svetina and B. Žekš: Membrane bending energy and shape determination of phospholipid vesicles and red blood cells. Eur. Biophys. J. *17* (1989), 101–111.

[42] E. Vitásek: Numerické metody (In Czech). SNTL, Nakladatelství technické literatury, 1987.

[43] N. J. Walkington: Algorithms for computing motion by mean curvature. SIAM J. Numer. Anal. *33* (1996), 6, 2215–2238.

[44] Y. Zhang, J. Cohen, and J. D. Owens: Fast tridiagonal solvers on the gpu. In: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2010, p. 10.

[45] Nvidia, http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf. NVIDIA CUDA Programming Guide 3.0, 2010.

Tomáš Oberhuber, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, Trojanova 13, 120 00 Praha 2. Czech Republic.
e-mail: tomas.oberhuber@fjfi.cvut.cz

Atsushi Suzuki, CERMICS ENPC, 6 et 8 avenue Blaise Pascal, Cité Descartes – Champs sur Marne, 77455 Marne la Vallée. France.
e-mail: Atsushi.Suzuki@cermics.enpc.fr

Vítězslav Žabka, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague, Trojanova 13, 120 00 Praha 2. Czech Republic.
e-mail: vitezslav.zabka@fjfi.cvut.cz