Journal of Math-for-Industry, Vol. 3 (2011C-10), pp. 73–79

Math-for-industry
Education & Research Hub

Image segmentation using CUDA implementations of the Runge-Kutta-Merson and GMRES methods

Tomáš Oberhuber, Atsushi Suzuki, Jan Vacata and Vítězslav Žabka

Revised on October 7, 2011

Abstract. Modern GPUs are well suited for performing image processing tasks. We utilize their high computational performance and memory bandwidth for image segmentation purposes. We segment cardiac MRI data by means of numerical solution of an anisotropic partial differential equation of the Allen-Cahn type. We implement two different algorithms for solving the equation on the CUDA architecture. One of them is based on the Runge-Kutta-Merson method for the approximation of solutions of ordinary differential equations, the other uses the GMRES method for the numerical solution of systems of linear equations. In our experiments, the CUDA implementations of both algorithms are about 3–9 times faster than corresponding 12-threaded OpenMP implementations.

Keywords. CUDA, image segmentation, Allen-Cahn equation, Runge-Kutta-Merson, GMRES

1. Introduction

Graphics processing units (GPUs) were initially designed to accelerate computer graphics rendering. Since then, they have evolved into general purpose computing devices. Because of their parallel architecture, high computational power and memory bandwidth, GPUs are capable of outperforming CPUs in compute-intensive data-parallel computations. General-purpose computing on graphics processing units (GPGPU) has been successfully used in many applications, such as physical simulations [11, 27], biological simulations [13, 14] and digital signal processing [12, 20, 23], where significant speedups have been achieved.

This article describes an application of GPGPU in the area of the image processing; the GPU is used to perform 2D image segmentation. We segment greyscale images given by the intensity function $I^0 : \langle 0, L_1\rangle \times \langle 0, L_2\rangle \to \langle 0, 1\rangle$. The geodesic active contours approach to image segmentation consists of setting an initial curve inside the object of our interest and evolving it to find the object boundary [2]. The motion of the segmentation curve can be expressed by the following formula for its normal velocity v (see [15]):

$$v = g^0 k + \nabla g^0 \cdot \vec{N}, \tag{1}$$

where k is its curvature, $\vec{N}$ is its normal vector and $g^0 \equiv g\big(|\nabla G_\sigma * I^0|\big)$. The function g is a smooth edge-indicator function, e.g. $g(s) = \frac{1}{1+\lambda s^2}$ with the parameter $\lambda > 0$. The term $G_\sigma * I^0$ represents convolution of the smoothing kernel G_σ and the segmented image I^0. Due to the shape of the Perona-Malik function g, the evolution of the segmentation curve slows down near edges in the image.

Evolving curves can be treated in several ways — by direct approach based methods [7], level-set methods [2, 8] or phase-field methods [6]. We choose the phase-field approach originating from physical models of phase transitions [1]. The spatial domain is split into two parts — a liquid and a solid phase. Between them, there is a narrow interface. We consider a function u which is, for instance, zero at the solid part, one at the liquid part, and it continuously changes from zero to one at the interface. Then, the interface curve can be detected as the level set $u = \frac{1}{2}$. In [6], this model has been modified for the purpose of the image segmentation. The following segmentation equation has been proposed:

$$\xi \frac{\partial u}{\partial t} = \xi \nabla \cdot \left(g^0 \nabla u\right) + \left(\frac{1}{\xi} f_0(u) + \xi F |\nabla u|\right) g^0. \tag{2}$$

It is a non-linear anisotropic partial differential equation of the Allen-Cahn type for the function $u = u(t, x)$ which evolves on a rectangular two-dimensional domain $\Omega \subset \mathbb{R}^2$. The parameter $t \in (0, T)$ represents time, $x = [x_1, x_2] \in \Omega$ is the spatial variable, $g^0 = g^0(x)$ is the edge-indicator function, $\xi > 0$ is a parameter related to the thickness of the interface layer, $f_0(u) = u(1-u)(u-\frac{1}{2})$ is a polynomial derived from the double-well potential w_0 (see [5]) as $f_0 = -w_0'$ and $F = F(x)$ has the meaning of an a priori information about the expected location and shape of the segmented object. The segmentation curve shrinks when F is zero, and this motion accelerates for F negative and slows or even inverts, i.e. the curve stretches, for F positive.

The initial-boundary-value problem for the equation (2)

reads as follows:

$$\begin{aligned}
&\xi\frac{\partial u}{\partial t} = \xi\nabla\cdot\left(g^0\nabla u\right) + \left(\frac{1}{\xi}f_0(u) + \xi F|\nabla u|\right)g^0 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{in } (0,T)\times\Omega, \quad (3)\\
&u|_{\partial\Omega} = 0 \quad \text{on } (0,T)\times\partial\Omega,\\
&u|_{t=0} = u_{\text{ini}} \quad \text{in } \overline{\Omega}.
\end{aligned}$$

The initial condition $u_{\text{ini}} = u_{\text{ini}}(x)$ can be given as a characteristic function of a domain covering the objects of interest or a domain inside the object of interest with the parameter F set appropriately.

1.1. Contribution

We present CUDA implementations of two algorithms to solve the problem (3) numerically. First, we apply the method of lines leading to a system of ordinary differential equations which we solve by the Runge-Kutta-Merson method. The other algorithm is based on the GMRES method which is used for the numerical solution of the system of linear equations obtained from the fully-discrete semi-implicit scheme for the problem (3).

1.2. Organization

The article is organized as follows. In Section 2, we show the numerical schemes for our problem and describe the Runge-Kutta-Merson method. In Section 3, we introduce the CUDA architecture. In Section 4, we present our implementations of the Runge-Kutta-Merson method and the GMRES method in the CUDA architecture. In Section 5, we compare the performance of our CUDA implementations with corresponding CPU implementations.

2. Numerical solution

Let $\Omega = (0,L_1)\times(0,L_2) \subset \mathbb{R}^2$ be the rectangular domain. We introduce the space steps $h = [h_1,h_2]$, the grid sizes $N_1 = \frac{L_1}{h_1}$, $N_2 = \frac{L_2}{h_2}$, the grid of internal nodes $\omega_h = \{[ih_1, jh_2] \mid i = 1,\dots,N_1-1;\ j = 1,\dots,N_2-1\}$, the grid of all nodes $\bar{\omega}_h = \{[ih_1, jh_2] \mid i = 0,\dots,N_1;\ j = 0,\dots,N_2\}$ and $\gamma_h = \bar{\omega}_h \setminus \omega_h$.

We consider the grid function $u^h : (0,T)\times\bar{\omega}_h \to \mathbb{R}$ which is defined by $u^h_{i,j}(t) = u(t, ih_1, jh_2)$. We denote backward and forward differences

$$u^h_{x_1,i,j} = \frac{u^h_{i+1,j} - u^h_{i,j}}{h_1}, \qquad u^h_{\bar{x}_1,i,j} = \frac{u^h_{i,j} - u^h_{i-1,j}}{h_1}, \quad (4)$$

$$u^h_{x_2,i,j} = \frac{u^h_{i,j+1} - u^h_{i,j}}{h_2}, \qquad u^h_{\bar{x}_2,i,j} = \frac{u^h_{i,j} - u^h_{i,j-1}}{h_2}, \quad (5)$$

and approximations of the gradient and the divergence $\left(\bar{\nabla}_h u^h\right)_{i,j} = \left[u^h_{\bar{x}_1,i,j}, u^h_{\bar{x}_2,i,j}\right]$, $\left(\nabla_h\cdot V^h\right)_{i,j} = V^1_{x_1,i,j} + V^2_{x_2,i,j}$ for $V^h = \left[V^1, V^2\right]$. Then the semi-discrete scheme for the problem (3) has the following form:

$$\begin{aligned}
&\frac{\mathrm{d}u^h}{\mathrm{d}t} = \nabla_h\cdot\left(g^0\bar{\nabla}_h u^h\right) + \left(\frac{1}{\xi^2}f_0\left(u^h\right) + F\left|\bar{\nabla}_h u^h\right|\right)g^0\\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{on } (0,T)\times\omega_h,\\
&u^h = 0 \quad \text{on } (0,T)\times\gamma_h,\\
&u^h(0) = u_{\text{ini}} \quad \text{on } \omega_h,
\end{aligned} \quad (6)$$

where g^0, F and u_{ini} stand for the grid values of the functions $g^0(x)$, $F(x)$ and $u_{\text{ini}}(x)$ respectively.

2.1. Explicit time discretization

The scheme (6) represents a system of ordinary differential equations which can be solved by the Runge-Kutta methods. We use the Runge-Kutta-Merson method [26] with adaptive choice of the integration step. Conveniently, the adaptive choice of the integration step guarantees the stability of the method. Now we describe the algorithm of the Runge-Kutta-Merson method for the scheme (6). We denote the right-hand side of the system by $f\left(t, u^h(t)\right)$ and include the boundary condition in the system. Then, the system can be rewritten in the form

$$\frac{\mathrm{d}u^h}{\mathrm{d}t} = f\left(t, u^h(t)\right), \quad (7)$$

where

$$f\left(t,u^h\right) = \begin{cases} \nabla_h\cdot\left(g^0\bar{\nabla}_h u^h\right) + \left(\frac{1}{\xi^2}f_0\left(u^h\right) + F\left|\bar{\nabla}_h u^h\right|\right)g^0 \\ \qquad\qquad\qquad\qquad \text{on } (0,T)\times\omega_h, \\ 0 \qquad \text{on } (0,T)\times\gamma_h. \end{cases} \quad (8)$$

The algorithm of the method reads as follows (see [26]):

1. Set $u^h_{i,j} := u_{\text{ini}}(ih_1, jh_2)$ for $i = 0,\dots,N_1$, $j = 0,\dots,N_2$ and $\tau := \tau_0$ for arbitrary $\tau_0 > 0$.
2. Compute grid functions $k^1,\dots,k^5$ as:

$$k^1_{i,j} := f\left(t,u^h\right)_{i,j}, \quad (9)$$

$$k^2_{i,j} := f\left(t+\frac{\tau}{3}, u^h + \frac{\tau}{3}k^1\right)_{i,j}, \quad (10)$$

$$k^3_{i,j} := f\left(t+\frac{\tau}{3}, u^h + \frac{\tau}{6}k^1 + \frac{\tau}{6}k^2\right)_{i,j}, \quad (11)$$

$$k^4_{i,j} := f\left(t+\frac{\tau}{2}, u^h + \frac{\tau}{8}k^1 + \frac{3\tau}{8}k^3\right)_{i,j}, \quad (12)$$

$$k^5_{i,j} := f\left(t+\tau, u^h + \frac{\tau}{2}k^1 - \frac{3\tau}{2}k^3 + 2\tau k^4\right)_{i,j} \quad (13)$$

   for $i = 0,\dots,N_1$, $j = 0,\dots,N_2$.
3. Evaluate the local truncation error

$$e := \tau\cdot\max_{\substack{i=0,\dots,N_1\\ j=0,\dots,N_2}}\left|\frac{1}{15}k^1_{i,j} - \frac{3}{10}k^3_{i,j} + \frac{4}{15}k^4_{i,j} - \frac{1}{30}k^5_{i,j}\right|. \quad (14)$$

4. If e is smaller than given tolerance ε, set $t = t + \tau$ and update the solution:

$$u^h_{i,j} := u^h_{i,j} + \frac{\tau}{6}\left(k^1_{i,j} + 4k^4_{i,j} + k^5_{i,j}\right) \qquad (15)$$

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

5. Update the integration step size as

$$\tau := \min\left\{0.8\,\tau\left(\frac{\varepsilon}{e}\right)^{0.2}, T - t\right\}. \qquad (16)$$

If $\tau > 0$, go to step 2.

2.2. Semi-implicit time discretization

In order to discretize the scheme (6) in time, we introduce a time step $\tau > 0$ and denote $u^{h,k} = u^h(k\tau)$. We use the fully-discrete semi-implicit scheme in the following form at the time level k:

$$\begin{aligned} \frac{u^{h,k} - u^{h,k-1}}{\tau} &= \nabla_h \cdot \left(g^0 \bar{\nabla}_h u^{h,k}\right) + \frac{g^0}{\xi^2} f_0\left(u^{h,k-1}\right) + \\ &\qquad g^0 F\left|\bar{\nabla}_h u^{h,k-1}\right| \quad \text{on } \omega_h, \\ u^{h,k} &= 0 \quad \text{on } \gamma_h, \\ u^{h,0} &= u_{\text{ini}} \quad \text{on } \omega_h. \end{aligned} \qquad (17)$$

The scheme can be written as a system of linear equations

$$\mathbf{A}\mathbf{u}^k = \mathbf{b}^k, \qquad (18)$$

where the vector $\mathbf{u}^k$ is composed of the values $u^{h,k}_{i,j}$ and the matrix $\mathbf{A}$ is sparse with one or five nonzero elements in each row. We solve the system by the GMRES method [21].

3. CUDA architecture

NVIDIA CUDA (Compute Unified Device Architecture) is a general-purpose computing architecture designed to take advantage of GPUs' computational power. It supports various programming languages or application programming interfaces such as CUDA C, which is a simple extension of the C programming language, CUDA Fortran, OpenCL and DirectCompute. Currently, CUDA applications work only on NVIDIA GPUs.

GPU, denoted as *device*, serves as a highly parallel programmable coprocessor to the CPU, denoted as *host*. It contains a scalable array of SIMT (single instruction, multiple thread) multiprocessors. The SIMT architecture is analogous to SIMD (single instruction, multiple data). Unlike SIMD, it enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads [19]. Device functions executed in parallel by different CUDA threads are called *kernels*.

Each multiprocessor executes hundreds of threads in groups of 32 threads called *warps*. Threads on a multiprocessor are organized in *thread blocks*, which may contain up to 1024 threads on current GPUs. Threads within a block can synchronize their execution and share data by fast *shared memory*. Cooperation of threads in different blocks is limited.

All threads have access to the same *global memory*. Global memory resides in device memory and is more plentiful but slower than shared memory. A very important performance consideration is coalescing global memory accesses. Global memory should be viewed in terms of segments of thirty-two 32-bit or 64-bit words aligned to 32 times the size of the word. If the k^{th} thread in a warp accesses the k^{th} word in one segment, the accesses are coalesced into one transaction. Otherwise, more transactions are issued, and the memory throughput decreases. However, the requirements to achieve coalescing are much more relaxed for newer devices. Reading from cached *texture memory*, which also resides in device memory, might lead to better performance than uncoalesced reading from global memory [18].

There are many significant hardware differences between CPU hosts and GPU devices. It is recommended to partition applications so that both systems carry out computations they are suitable for. Devices are especially well suited for computations that can be run on numerous data elements in parallel, e.g. arithmetic operations on vectors and matrices. In order to get maximum performance, it is necessary to maximize the amount of code which can be parallelized.

4. Implementation in CUDA

4.1. Runge-Kutta-Merson method

We implement the Runge-Kutta-Merson algorithm described in Section 2. The grid functions $u^h, k^1, \dots, k^5$ are allocated as arrays `U`, `K1`, ..., `K5` of length $(N_1+1)(N_2+1)$ in device memory and operations on them are performed for all their elements in parallel. At the beginning of the algorithm, the values $u_{\text{ini}}(ih_1, jh_2)$ are copied to array `U`, then they are processed by the device, and, at the end, the array `U` contains the result of the algorithm.

Implementation of three basic operations in CUDA is necessary: the linear combination of arrays (steps 2, 3 and 4 of the algorithm), finding the maximum magnitude element of an array (step 3) and evaluation of the right-hand side function (step 2).

The implementation of the linear combination of arrays stored in device memory is straightforward. In Listing 1 we show the CUDA kernel for the operation `y = a1*x1 + a2*x2`, where `y`, `x1`, `x2` are arrays and `a1`, `a2` given constants. Linear combinations of more than two arrays are implemented similarly.

In the CUDA architecture, the maximum magnitude element of an array is typically obtained by a parallel reduction [10]. We use its reference implementation from the CUBLAS library — the `cublasIdamax()` function.

The right-hand side function f given by (8) is evaluated in `f_kernel()` (see Listing 2). Each element of the re-

```
__global__ void
lincomb_kernel(double *y, int length,
               double a1, double *x1,
               double a2, double *x2)
{
  int tid = blockIdx.x*blockDim.x + threadIdx.x;
  if (tid < length)
    y[tid] = a1*x1[tid] + a2*x2[tid];
}
```

Listing 1: CUDA kernel for the linear combination of two arrays of given length in global device memory.

```
texture<int2, 1, cudaReadModeElementType> texU;
texture<int2, 1, cudaReadModeElementType> texG;

__device__ double
fetch(texture<int2, 1, cudaReadModeElementType> tex,
      int index)
{
  int2 v = tex1Dfetch(tex, index);
  return __hiloint2double(v.y, v.x);
}

__global__ void
f_kernel(double *fu, int N1, int N2, double *F,
         double h1, double h2, double ksi)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int j = blockIdx.y*blockDim.y + threadIdx.y;
  if (i <= N1 && j <= N2) {
    int index = j*(N1 + 1) + i;

    if (i == 0 || i == N1 || j == 0 || j == N2)
      fu[index] = 0.0;
    else {
      double u = fetch(texU, index);
      double g = fetch(texG, index);

      double bdx = (u - fetch(texU, index - 1     ))/h1;
      double bdy = (u - fetch(texU, index - 1 - N1))/h2;
      double fdx = (fetch(texU, index + 1     ) - u)/h1;
      double fdy = (fetch(texU, index + 1 + N1) - u)/h2;

      fu[index] =
          (fetch(texG, index + 1     )*fdx - g*bdx)/h1 +
          (fetch(texG, index + 1 + N1)*fdy - g*bdy)/h2 +
          g*(u*(1.0 - u)*(u - 0.5)/(ksi*ksi) +
             F[index]*sqrt(bdx*bdx + bdy*bdy));
    }
  }
}
```

Listing 2: CUDA kernel for evaluation of the right-hand side function f given by (8).

sulting array is computed by a single CUDA thread; the threads are organized in two-dimensional blocks, and they do not cooperate. According to (8), five values $u^h_{i-1,j}$, $u^h_{i,j-1}$, $u^h_{i,j}$, $u^h_{i,j+1}$, $u^h_{i+1,j}$ are needed to compute $f(t,u^h)_{i,j}$ on ω_h, and the same holds for the values of the function g^0. We use texture memory and read these values from textures `texU` and `texG`.

4.2. GMRES method

A detailed description of the GMRES method can be found in [22]. Our CUDA implementation comprises four operations with matrices and vectors: the linear combination of vectors, the Euclidean norm of a vector, the dot product of two vectors and the sparse matrix-vector product. In order to avoid unnecessary data transfers between the device and the host we also assemble the right-hand side vectors $\mathbf{b}^k$ on the device using a kernel similar to `f_kernel()` presented in Listing 2.

We already discussed the implementation of the linear combination of vectors in Section 4.1. To compute the Euclidean norm of vectors and the dot product we use functions `cublasDnrm2()` and `cublasDdot()` from the CUBLAS library.

Effective implementation of the sparse matrix-vector product is crucial for many numerical algorithms and has been deeply investigated (see [3, 4, 16, 17, 25]). We use the approach presented in [24], which is based on a special storage format for sparse matrices called Row-Grouped CSR (RgCSR). It is a modification of the CSR matrix storage format, designed so that all global memory accesses during matrix-vector multiplication are coalesced.

According to the RgCSR format, the matrix is divided into *groups*. A group of size k consists of k consecutive matrix rows. Rows 1 to k form the first group, rows $k+1$ to $2k$ the second group etc. The overall number of groups of a $n \times n$ matrix is $\lceil \frac{n}{k} \rceil$. The matrix is stored in the following four arrays (see Fig. 1):

- `values[]` — array of nonzero elements of the matrix. First k elements of the array represent the first nonzero matrix elements in rows of the first group, next k elements represent second nonzero matrix elements in rows of the first group etc. If there are not enough nonzero elements in some row, their places in the array remain unused. When all nonzero elements of the first group are stored, the same procedure is applied to other groups. Let $r_1, \ldots, r_n$ be the numbers of nonzero elements in the rows of the matrix and
$$m_i = \max\{r_{(i-1)k+1}, r_{(i-1)k+2}, \ldots, r_{ik}\} \qquad (19)$$
for $i = 1, \ldots, \lceil \frac{n}{k} \rceil$. Then the i^{th} group occupies km_i elements in the array and the overall length of the array is
$$M = \sum_{i=1}^{\lceil \frac{n}{k} \rceil} m_i \,. \qquad (20)$$
- `columns[]` — array of column indices. Its elements are the column indices of corresponding elements of the `values[]` array. Both arrays have the same length.
- `rowSizes[]` — array of the row sizes $r_1, \ldots, r_n$.
- `groupPtrs[]` — array of length $\lceil \frac{n}{k} \rceil$ containing indices of first elements of groups in the `values[]` array, i.e. numbers $0, km_1, k(m_1+m_2), \ldots, k\sum_{i=1}^{p} m_i$ where $p = \lceil \frac{n}{k} \rceil - 1$.

Our CUDA kernel for the sparse matrix-vector product is presented in Listing 3. Each CUDA thread computes one component of the resulting vector y. The kernel uses shared memory as a user-managed cache for elements of the `groupPtrs[]` array. It needs `blockDim.x/k*sizeof(int)` bytes of shared memory per CUDA block to execute. If the group size k is divisible by the warp size, all global device memory accesses are fully coalesced.

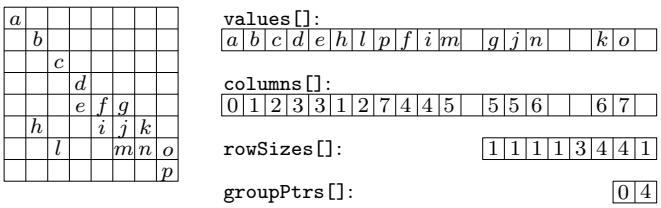

Figure 1: Example of RgCSR sparse matrix storage format for group size $k = 4$. The matrix has 8 rows, so the number of groups is 2. The first group occupies 4 elements of the `values[]` array.

```
texture<int2, 1, cudaReadModeElementType> texX;

__global__ void
spmv_kernel(double *Ax, int n, int k, double *values,
            int *columns, int *rowSizes, int *groupPtrs)
{
  extern __shared__ int groupPtrsCache[];

  int offset, groupIndex;
  int rowIndex = blockIdx.x*blockDim.x + threadIdx.x;
  if (rowIndex < n) {
    groupIndex = threadIdx.x/k;
    offset = threadIdx.x%k;
    if (offset == 0)
      groupPtrsCache[groupIndex] = groupPtrs[rowIndex/k];
  }

  __syncthreads();

  if (rowIndex < n) {
    double value = 0.0;
    int index = groupPtrsCache[groupIndex] + offset;
    for (int i = 0; i < rowSizes[rowIndex]; i++) {
      value += values[index]*fetch(texX, columns[index]);
      index += k;
    }

    Ax[rowIndex] = value;
  }
}
```

Listing 3: CUDA kernel for RgCSR sparse matrix-vector product.

5. Results

We tested the presented CUDA implementations of the image segmentation algorithms on magnetic resonance images of the human heart (see Fig. 2). The segmentation can be used to determine the volume of cardiac ventricles and help in diagnosis of heart diseases [8, 9]. We performed all computations using double precision floating point arithmetics. We compared running times of the CUDA implementations on NVIDIA GeForce GTX 480 (480 cores, 1.4 GHz) with running times of the corresponding multithreaded OpenMP implementations on AMD Opteron 6172 (12 cores, 2.1 GHz). The results are shown in Tables 1 and 2. The CUDA applications were significantly faster than the 12-threaded CPU application, up to about 9 times for the higher image resolution.

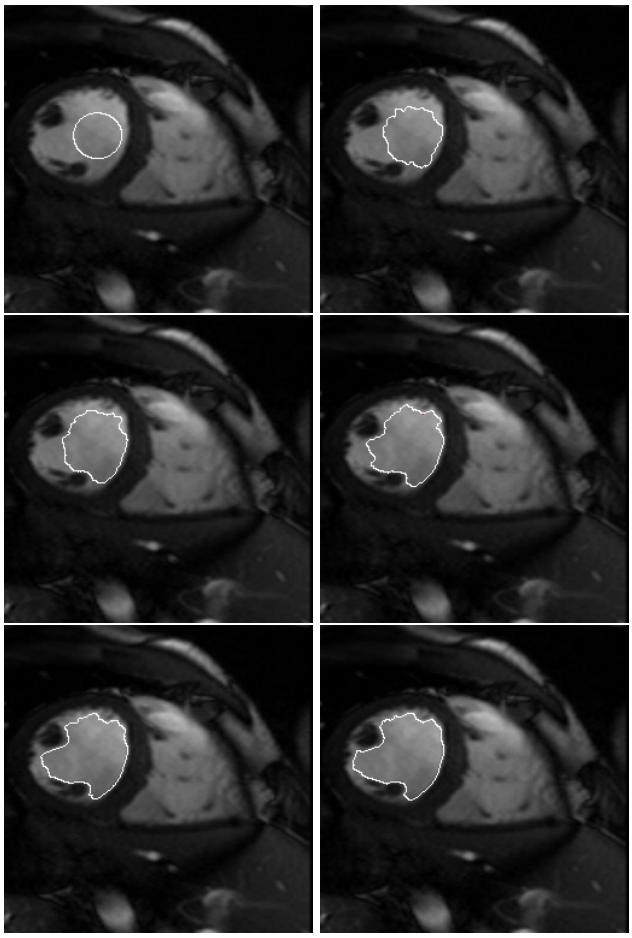

Figure 2: Image segmentation of the right heart ventricle. The white line represents the segmentation curve. The initial condition is shown in the top left corner. Parameters: $L_1 = L_2 = 1$, $N_1 = N_2 = 255$, $\xi = h_1 + h_2$, $g(s) = \frac{1}{1+s^2}$, G_σ is the Gaussian kernel, $\sigma = 0.005$, $F \equiv 100$.

6. Conclusion

We implemented the Runge-Kutta-Merson method and the GMRES method in the CUDA architecture and applied them to the numerical solution of problem (3) concerning the image segmentation. We compared the CUDA implementations with corresponding multithreaded CPU implementations. We observed that the CUDA implementations were about 3–9 times faster than the 12-threaded CPU implementations.

Acknowledgements

This work was partially supported by the Research Direction Project "Applied Mathematics in Technical and Physical Sciences" of the Ministry of Education of the Czech Republic No. MSM6840770010 and by the project "Advanced Supercomputing Methods for Implementation of Mathematical Models" of the Student Grant Agency of the Czech Technical University in Prague No. SGS11/161/OHK4/3T/14. MRI images were provided by

Resolution	Run on	Time	RT
512×512	GPU	9.8 s	1.0
	CPU (1 thread)	448 s	45.7
	CPU (2 threads)	220 s	22.4
	CPU (4 threads)	108 s	11.0
	CPU (12 threads)	50 s	5.1
1024×1024	GPU	143 s	1.0
	CPU (1 thread)	7780 s	54.4
	CPU (2 threads)	4100 s	28.7
	CPU (4 threads)	2300 s	16.1
	CPU (12 threads)	1410 s	9.9

Table 1: Performance comparison of CPU and GPU image segmentation using the Runge-Kutta-Merson method. The last column shows segmentation duration relative to the GPU implementation.

Resolution	Run on	Time	RT
512×512	GPU	28 s	1.0
	CPU (1 thread)	435 s	15.5
	CPU (2 threads)	217 s	7.8
	CPU (4 threads)	115 s	4.1
	CPU (12 threads)	80 s	2.9
1024×1024	GPU	182 s	1.0
	CPU (1 thread)	7410 s	40.7
	CPU (2 threads)	3895 s	21.4
	CPU (4 threads)	1905 s	10.5
	CPU (12 threads)	1485 s	8.2

Table 2: Performance comparison of CPU and GPU image segmentation using the GMRES method. The last column shows segmentation duration relative to the GPU implementation.

Institute for Clinical and Experimental Medicine, Prague.

References

[1] Allen, S. and Cahn, J.: A microscopic theory for antiphase boundary motion and its application to antiphase domain coarsening, *Acta Metallurgica*, **27**:1085–1095, 1979.

[2] Aubert, G. and Kornprobst, P.: Mathematical problems in image processing: Partial differential equations and the calculus of variations (second edition), Volume 147 of Applied Mathematical Sciences, Springer Verlag, 2006.

[3] Baskaran, M. and Bordawekar, R.: Optimizing sparse matrix-vector multiplication on GPUs, Research Report RC24704, IBM TJ Watson Research Center, 2008.

[4] Bell, N. and Garland, M.: Efficient sparse matrix-vector multiplication on CUDA, Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[5] Beneš, M.: Mathematical and computational aspects of solidification of pure substances, *Acta Math. Univ. Comenian.*, **70**(1):123–152, 2001.

[6] Beneš, M. and Chalupecký, V. and Mikula, K.: Geometrical image segmentation by the Allen-Cahn equation, *Applied Numerical Mathematics*, **51**(2–3):187–205, 2004.

[7] Beneš, M. and Kimura, M. and Pauš, P. and Ševčovič, D. and Tsujikawa, T. and Yazaki, S.: Application of a curvature adjusted method in image segmentation, *Bulletin of the Institute of Mathematics*, Academia Sinica (New Series), **3**(4):509–523, 2008.

[8] Beneš, M. and Máca, R. and Tintěra, J.: Degenerate diffusion methods in computer image processing and application, to appear in Journal of Math-for-Industry.

[9] Bogaert, J. and Dymarkowski, S. and Taylor, A. M.: Clinical cardiac MRI, Springer Verlag, 2005.

[10] Harris, M.: Optimizing parallel reduction in CUDA, NVIDIA CUDA SDK, 2010.

[11] Ji, X. and Cheng, T. and Wang, Q.: A simulation of large-scale groundwater flow on CUDA-enabled GPUs, SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing, pp. 2402–2403, 2010.

[12] Kang, S. G. and Sun, P. J. and Kim, C. H. and Kim, J.-M.: Power analysis for decoding of the digital audio encoding format MP3: Decoding the central processing unit and the graphics processing unit, *The Journal of the Acoustical Society of America*, **127**(3):2037–2037, 2010.

[13] Liu, W. and Schmidt, B. and Voss, G. and Muller-Wittig, W.: Accelerating molecular dynamics simulations using graphics processing units with CUDA, *Computer Physics Communications*, **179**(9):634–641, 2008.

[14] Manavski, S. and Valle, G.: CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment, *BMC Bioinformatics*, **9**(2): S10, 2008.

[15] Mikula, K. and Sarti, A.: Parallel co-volume subjective surface method for 3D medical image segmentation, In: Parametric and Geometric Deformable Models: An application in Biomaterials and Medical Imagery, Volume-II, Springer Publishers, (Eds. Jasjit S. Suri and Aly Farag), pp. 123–160, 2007.

[16] Monakov, A. and Avetisyan, A.: Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs, In SAMOS '09, pp. 289–297, 2009.

[17] Monakov, A. and Lokhmotov, A. and Avetisyan, A.: Automatically tuning sparse matrix-vector multiplication for GPU architectures, High Performance Embedded Architectures and Compilers, 5th International Conference, pp. 111–125, 2010.

[18] NVIDIA Corporation, CUDA C best practices guide, Version 3.2, 2010.

[19] NVIDIA Corporation, NVIDIA CUDA programming guide, Version 3.2, 2010.

[20] Rofouei, M. and Moazeni, M. and Sarrafzadeh, M.: Fast GPU-based space-time correlation for activity recognition in video sequences, IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia, pp. 33–38, 2008.

[21] Saad, Y. and Schultz, M.: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM Journal on Scientific and Statistical Computing*, **7**(3):856–869, 1986.

[22] Saad, Y.: Iterative methods for sparse linear systems, Society for Industrial and Applied Mathematics, 2003.

[23] Trebien, F. and Oliveira, M.: Realistic real-time sound re-synthesis and processing for interactive virtual worlds, *The Visual Computer*, **25**(5):469–477, 2009.

[24] Vacata, J.: GPGPU: General purpose computation on GPUs, Master's thesis, FNSPE CTU in Prague, 2008.

[25] Vázquez, F. and Garzón, E. and Martínez, J. and Fernández, J.: The sparse matrix vector product on GPUs, Technical Report, University of Almería, 2009.

[26] Vitásek, E.: Numerické metody, SNTL, Prague, 1987.

[27] Zhu, J. and Liu, Y. and Bao, K. and Chang, Y. and Wu, E.: Realtime simulation of burning solids on GPU with CUDA, 10th IEEE International Conference on Computer and Information Technology, pp. 1219–1224, 2010.

Tomáš Oberhuber, Jan Vacata and Vítězslav Žabka
Department of Mathematics, Faculty of Nuclear Sciences and Physical Engineering, Czech Technical University in Prague
E-mail: tomas.oberhuber(at)fjfi.cvut.cz
jan.vacata(at)seznam.cz
zabkavit(at)fjfi.cvut.cz

Atsushi Suzuki
CERMICS ENPC, France
E-mail: atsushi.suzuki(at)ann.jussieu.fr