Journal of Computational and Applied Mathematics, Volume 293, February 2016, pp. 35-44
DOI: 10.1016/j.cam.2015.04.035

# A Parallel Generalized Relaxation Method for High–Performance Image Segmentation on GPUs☆

Pasqua D'Ambra[a,*], Salvatore Filippone[b]

[a]*Institute for High–Performance Computing and Networking, CNR, Via Pietro Castellino 111, I-80131 Naples, Italy*
[b]*Department of Civil and Computer Engineering, University of Rome "Tor Vergata", Via del Politecnico 1, I-00133, Rome, Italy*

**Abstract**

Fast and scalable software modules for image segmentation are needed for modern high–throughput screening platforms in Computational Biology. Indeed, accurate segmentation is one of the main steps to be applied in a basic software pipeline aimed to extract accurate measurements from a large amount of images. Image segmentation is often formulated through a variational principle, where the solution is the minimum of a suitable functional, as in the case of the Ambrosio–Tortorelli model. Euler–Lagrange equations associated with the above model are a system of two coupled elliptic partial differential equations whose finite–difference discretization can be efficiently solved by a generalized relaxation method, such as Jacobi or Gauss–Seidel, corresponding to a first–order alternating minimization scheme. In this work we present a parallel software module for image segmentation based on the Parallel Sparse Basic Linear Algebra Subprograms (PSBLAS), a general–purpose library for parallel sparse matrix computations, using its Graphics Processing Unit (GPU) extensions that allow us to exploit in a simple and transparent way the performance capabilities of both multi–core CPUs and of many–core GPUs. We discuss performance results in terms of execution times and speed–up of the segmentation module running on GPU as well

☆This work has been partially supported by *Public–Private Laboratory for the Development of Integrated Informatics Tools for Genomics, Transcriptomics and Proteomics (LAB GTP)*, funded by MIUR.

*Corresponding author.
*Email addresses:* pasqua.dambra@cnr.it (Pasqua D'Ambra), salvatore.filippone@uniroma2.it (Salvatore Filippone)

as on multi-core CPUs, in the analysis of 2D gray–scale images of mouse embryonic stem cells colonies coming from biological experiments.

*Keywords:* Image Segmentation, Variational Models, Relaxation Methods, GPU
*2000 MSC:* 65K10, 65N22, 65Y05, 68T45

---

## 1. Introduction

Accurate and fast image segmentation is a key issue when thousands of images have to be processed to extract relevant measurements from cell or tissues images arising in modern high–throughput screening and analysis [1].

Image segmentation is generally defined as the problem of identifying the image regions corresponding to the single objects. From a mathematical point of view, if $f$ is an image defined on a Lipschitz bounded domain $\Omega$, segmentation can be defined as the problem to find a suitable boundary set $\Gamma$ (*the edge set*) corresponding to a complete partition of $\Omega$, where each connected component $\Omega_i$ corresponds to a single physical object. This problem has a basic formulation in terms of the well–known Mumford–Shah (MS) functional, whose theory and approximation methods for practical solutions have been widely studied in the last 20 years [2, 3, 4]. In this paper we focus on the Ambrosio–Tortorelli approximation of the MS functional and discuss an efficient implementation of the solution algorithm introduced in [5] on modern General–Purpose Graphics Processing Units (GPGPUs) for high–performance computing.

Our implementation relies on Parallel Sparse Basic Linear Algebra (PSBLAS) software framework which implements basic Linear Algebra operators on sparse matrices for an efficient and transparent implementation on parallel architectures where MPI is available [6, 7][1]. PSBLAS is a library that allows easy and convenient implementation of iterative methods by providing sparse matrix operators such as matrix–vector product, sparse triangular system solution, splitting of a sparse matrix into upper and lower triangles, and more. These operators are complemented by an infrastructure that allows building and handling the necessary sparse matrix data structures in a simple way, even when dealing with sophisticated schemes for storing the coefficient

[1] The PSBLAS software is freely available from `http://www.ce.uniroma2.it/psblas`.

entries into the computer memory. The resulting expressiveness of the matrix operators allows implementation of iterative methods in a notation that is extremely similar to a blackboard description in pseudo code. The PSBLAS framework also forms the base for a package of algebraic multilevel preconditioners [8, 9].

Recently we introduced a plugin for PSBLAS that enables computations on GPGPUs; the software architecture allows us to have a code that is capable of running on parallel multi-core CPUs with MPI as well as on GPU with Cuda in a transparent way, as described in [10]. This capability has been exploited in this work to make the image segmentation GPU–enabled; to this end we have also had to extend some of the functionalities, as described in more details in Section 3.

The paper is organized as follows: in Section 2 we briefly describe the image segmentation model and its numerical solution algorithm, in Section 3 we discuss the main issues involved in an efficient implementation of the algorithm on GPU and describe the optimizations applied to critical phases of the computations. In Section 4 we discuss performance results and, finally, in Section 5 we give some remarks and future plans.

## 2. Mathematical Model and Numerical Algorithm

In this work we focus on the MS model, which is the prototype of all the variational models for image segmentation. In particular, we are interested in the phase–field approximation of the MS model, introduced in [11] by Ambrosio and Tortorelli, where the original grayscale image function $f$ is approximated by a piecewise smooth function $u$, which can be discontinuous across a closed set $K \subset \Omega$ represented by a suitable function $z$.

Let $\Omega \subset \mathbb{R}^2$ be a Lipschitz bounded open set and $f \in L^\infty(\Omega)$ the observed grayscale image, the segmentation problem can be described in terms of the minimization of the following functional:

$$E_\epsilon(u,z) = \int_\Omega (u-f)^2 dxdy + \beta \int_\Omega z^2 |\nabla u|^2 dxdy + \alpha \int_\Omega \left( \epsilon |\nabla z|^2 + \frac{(z-1)^2}{4\epsilon} \right) dxdy, \quad (1)$$

where $u \in C^1(\Omega \setminus K)$, and $z$, $0 \le z \le 1$, is a function which controls $|\nabla u|$ and gives an approximate representation of the set $K$ ($z(x,y) \approx 0$ if $(x,y) \in K$

and $z(x,y) \approx 1$ if $(x,y)$ is part of the smooth regions), $\alpha$ and $\beta$ are positive coefficients and $\epsilon$ is a positive sufficiently small parameter. Coefficients $\alpha$ and $\beta$ are suitable weights which control the so–called penalizing terms in (1); in particular, parameter $\beta$ controls the smoothness of the function $u$ outside of the edge set $K$ (for increasing $\beta$ we look for very small gradients outside of $K$), while parameter $\alpha$ controls the length of the edge set, so that large values for $\alpha$ require less jumps in the recovered image function.

Parameter $\epsilon$ was introduced by the phase–field approximation to obtain a sequence of elliptic functionals which, for $\epsilon \to 0$, are $\Gamma-$convergent to the original MS functional [12]. The need to choose a good $\epsilon$ is one of the main drawbacks, together with non–convexity, in numerical approximations of (1); generally a good choice for $\epsilon$ is such that $h/\epsilon < 1$, where $h$ is the discretization mesh–size. Further details on this issue may be found in [5] and the references therein.

The interest for model (1) is related to the the fact that its minimization is achievable by well–known techniques from Calculus of Variations, i.e., by writing the corresponding Euler–Lagrange equations. Our numerical algorithm for solving (1) is based on a second–order finite–difference discretization of the following Euler–Lagrange equations:

$$\begin{cases} 2(u-f) - 2\beta\nabla\cdot(z^2\nabla u) = 0 \\ 2\beta z|\nabla u|^2 - 2\alpha\epsilon\nabla^2 z + \frac{\alpha}{2\epsilon}(z-1) = 0 \end{cases} \qquad (x,y) \in \Omega, \tag{2}$$

coupled with natural boundary conditions. Equations (2) are a system of coupled elliptic partial differential equations which, in discrete form, when the image domain $\Omega$ has been discretized by fixing a mesh–size $h > 0$ (usually related to the original image size) and a so–called unknown–based discretization of the equations has been applied [13], can be written in the following block form:

$$\begin{bmatrix} A(\mathbf{z}) & \mathbf{0} \\ \mathbf{0} & B(\mathbf{u}) \end{bmatrix} \begin{pmatrix} \mathbf{u} \\ \mathbf{z} \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \end{pmatrix}. \tag{3}$$

Details on the discretization scheme used for system (2) can be found in [5], here we note that matrices $A(\mathbf{z})$ and $B(\mathbf{u})$ have the same symmetric sparsity pattern and are both diagonally dominant M–matrices, per each finite vector $(\mathbf{u},\mathbf{z})$.

In [5] we propose to solve system (3) by using a generalized relaxation method, such as the Gauss–Seidel method, accelerated by inner linear iterations. We proved that this approach is more efficient and reliable when

compared both with standard non–linear Gauss–Seidel iterations and with other first–order alternating minimization schemes, such as gradient descent. The solution algorithm is described in Algorithm 1.

**Algorithm 1:** Gauss–Seidel method coupled with inner linear iterations

$k = 0$, $\mathbf{z}^0 = \mathbf{1}$ (all–ones vector) and $\mathbf{u}^0 = f_h$ (original image);
build r.h.s. $\mathbf{f}_1$ and $\mathbf{f}_2$;
**repeat**
  build matrix $A(\mathbf{z}^k)$;
  compute $\mathbf{u}^{k+1}$ by *iterative solution of system* $A(\mathbf{z}^k)\mathbf{u} = \mathbf{f}_1$, starting from $\mathbf{u}^k$;
  build matrix $B(\mathbf{u}^{k+1})$ ;
  compute $\mathbf{z}^{k+1}$ by *iterative solution of system* $B(\mathbf{u}^{k+1})\mathbf{z} = \mathbf{f}_2$, starting from $\mathbf{z}^k$;
  $k = k + 1$;
**until** *convergence*;

Since matrices $A(\mathbf{z})$ and $B(\mathbf{u})$ are both M–matrices and diagonally dominant, classical relaxation methods, such as Jacobi and Gauss–Seidel, are convergent methods for the inner linear systems [14]. In this work we use Jacobi iterations for the reasons discussed in the next section, regarding the exploitation of parallelism of modern GPUs.

## 3. Parallel Implementation Issues on GPU

GPGPUs have gained widespread adoption in the scientific computing community and are now routinely used to accelerate a broad range of science and engineering applications, delivering dramatically improved performance in many cases. They are used to build the core of the most advanced supercomputers, as demonstrated by the latest list(s) of the TOP500 supercomputer sites[2]; cloud–computing providers are deploying clusters with multiple GPUs per node and high–speed network interconnections (e.g., Cluster GPU instances in Amazon EC2) in order to make them a feasible option for HPC as a Service (HPCaaS) [15].

[2] `http://www.top500.org/`

The NVIDIA GPU is based on a scalable array of multi–threaded, streaming multi–processors, each composed of a fixed number of scalar processors, a dual–issue instruction fetch unit, an on–chip fast memory with a configurable partitioning of shared memory, an L1 cache plus additional special–function hardware.

Computations are carried out by threads grouped into blocks; more than one block can execute on the same multiprocessor, and each block executes concurrently. Each multiprocessor employs a Single Instruction Multiple Threads (SIMT) architecture akin to the SIMD architecture of traditional vector supercomputers. The multiprocessor creates, schedules, and executes threads in groups called warps; threads in a warp start together at the same program address but can execute their own instructions and are free to branch independently. A warp executes one common instruction at a time, so the maximum performance is achieved when all threads in a warp follow the same path. A very important issue is the access to the memory subsystem on the GPU device; to achieve best performance, threads in a warp should access contiguous memory locations, executing so–called *coalesced* accesses, since this guarantees full memory bandwidth utilization.

One peculiar aspect of the GPUs is that they are *throughput oriented*, that is, they strive to achieve maximal utilization of the available arithmetic units by scheduling and launching many threads. This scheme works well under two main assumptions, first, that context switching between threads is extremely fast, and second, that there are enough resources to be shared among the threads; the resources are shared among all blocks executing on the same muiltiprocessors. Hence, there should be enough blocks to keep all multiprocessors busy, but the number of threads per block cannot increase too much because there is a fixed number of registers to be shared among all threads in a multiprocessor (see Section 4).

### *3.1. Basic Sparse Linear Algebra Operators on GPUs*

Our segmentation algorithm, as described in Algorithm 1, involves computations with sparse matrices arising from the discretization of 2D elliptic operators with standard 5–point stencils. The main computational kernel of the inner iterative linear solvers is the matrix–vector product $\mathbf{y} \leftarrow \alpha A\mathbf{x}+\beta\mathbf{y}$, where $A$ is large and sparse and $\mathbf{x}$ and $\mathbf{y}$ are column vectors. It is well known that the Sparse Matrix–Vector product (SpMV) is a memory–bound kernel; on GPGPUs it introduces additional challenges with respect to operations on dense matrices because on sparse matrices we have much less regular data

accesses patterns [16]. Indeed, efficient implementations of the operation requires to devise a specific storage format for the matrix $A$. Typical storage formats include variations on the CSR (Compressed Sparse Rows) and ELLPACK (ELL) formats; in this work we have employed an ELL–like format, which gives the best results for standard sparse matrices, such as that arising from elliptic operators [17]. The usage of the ELL format and its variants on GPUs has been analyzed in, for example, [16, 18]; the implementation used in PSBLAS is described in [17], whereas a review of many storage formats will be available in [19].

In the standard ELL format [20], an $M$-by-$N$ sparse matrix with at most $K$ nonzeros per row is stored as dense $M$-by-$K$ arrays `val` and `ja` holding the nonzero matrix values and the corresponding column indices, respectively. The rows in `val` and `ja` with fewer than $K$ nonzero elements are filled in with appropriate padding values. The ELL format thus fits a sparse matrix into a regular data structure, making it a good candidate for implementing sparse–matrix operations on SIMT architectures. In the computational kernel, each row of the sparse matrix is assigned to a small set of threads that performs the associated (sparse) dot product with the input vector `x`. The padding in the shorter rows introduces a certain amount of memory overhead with respect to the CSR format; this is acceptable if the maximum number of nonzeros $K$ is not much larger than the average, which is the case in the current application.

As already mentioned, threads are organized in groups called *warps*, and to maximize memory bandwidth usage each warp should handle a group of consecutive rows of the sparse matrix. The storage of the matrix data in a regular, (2D) array allows each thread to compute easily where its data (i.e., the relevant matrix row) reside in memory given only the row index and the size of the 2D array: it enables coordination among the various threads in accessing consecutive elements of the `ja` and `val` arrays, thereby maximizing the utilization of memory bandwidth through so-called *coalesced* memory accesses. These techniques are standard practice on GPUs, as attested by the analysis in [16]; in our case, the matrices $A(\mathbf{z})$ and $B(\mathbf{u})$ are very regular, with almost all rows having the same number of nonzeros and very few rows having fewer nonzeros. These features make the ELL format very attractive, since padding will be quite limited, with only a few smaller rows impacted. Moreover, our equations are discretized with a 5-point stencil, therefore the resulting matrices will have short rows; in our implementation [17] we choose dynamically the number of threads per row based on the average number

of nonzeros, and in this case we always use one thread per row. The performance we get for these matrices is on par with that of the HYB format from the NVIDIA cuSPARSE library [21]; note that even though we do have an interface available for cuSPARSE, we cannot use it directly in this application because their data structures are opaque, whereas one of the critical issues for this application is the ability to update the matrix coefficients.

A further critical issue in writing efficient codes for GPU is the need to move data from main memory to the GPU memory (*host* memory to *device* memory in CUDA nomenclature), and vice versa. Unfortunately, the above data movement is very slow compared to the high bandwidth internal to the GPU, and this is one of the major challenges in GPU programming [22]. Therefore, we have to prearrange relevant data to reside on the GPU; this can be achieved by using a careful object–oriented design in which the GPU–enabled data structures keep track of the location of the data on the host/device side and move data only when needed. More details can be found in [10], together with performance results quantifying the data movement overhead. The linear solver code shown in Fig. 1 relies on this infrastructure and works on the CPU or the GPU according to the dynamic (i.e., runtime) type of its arguments without the need for any modifications.

### *3.2. Data management*

A salient feature of the method implemented in this application, as in general non-linear iterative computations, is the fact that we are solving a sequence of linear systems in which the coefficient matrix changes at each non–linear iteration. Thus, the computations necessary to update the coefficient matrices use a significant percentage of the total runtime (see Section 4); it is therefore essential to optimize them.

The library framework provides the functionality necessary to incrementally build or update a sparse matrix from a sequence of triples $(row, col, val)$ provided by the user; therefore the application will have to

1. "Walk" through the discretization mesh and build the updates to the coefficients for $A(\mathbf{z})$ and $B(\mathbf{u})$;
2. Pass these coefficients to the library to store them into the matrix data structure.

Since this is a very heavy part of the application (see 4), it should be implemented on the GPU, and again care should be taken to avoid as much as possible data traffic between host and device. Hence

- The generation of the updates must be performed in the GPU;
- The updates to the sparse matrix structure must work with both the matrix and the coefficient updates already in the GPU device memory;

For the second item, we needed to extend the sparse matrix management layer of PSBLAS to make available sparse matrix and vector update functions on the GPU, since they had previously only been implemented for CPU. The new functionality has been included in the PBSLAS framework for general non–linear computations on GPUs and will be made available publicly with the next release of the toolkit.

In our application, at each non–linear iteration only a subset of the matrix coefficients are updated; therefore the matrix generation functions have been split into an initial phase in which we generate the entire linear system matrices (this is done on the CPU, since it needs only be executed once), and a selective update phase to be performed at each non–linear iteration. The updates are arranged by groups of rows; the number of rows in a group should be sufficiently large as to reduce the overhead of the calls to the update routines, but not too large to avoid excessive memory footprint and (on the CPU) to make effective use of the cache; in the current implementation on the CPU we used groups of 512 rows, which in our tests gave the best results.

The optimal size and schedule of the updates on the GPU is guided by different considerations. The GPU code that regenerates the coefficients inspects all the matrix rows, with each thread responsible for the updates of one row. The number of rows to be handled in a single invocation must be much greater than with the CPU because there must be enough threads to keep all the multiprocessors busy. On the other hand, each multiprocessor should have a number of thread blocks available, but not too many, because they share the available registers; for the update generation code on the device employed in Section 4, a block size of 256 is adequate, given the maximum number of registers per multiprocessor available. The total number of threads, and hence the number of rows in a group, is then determined by ensuring that each multiprocessor receives a given number of blocks, and multiplying by the number of multiprocessors on the particular device; the experiments of Section 4 have been run with groups of 16640 rows.

The sequence of coefficients update on the GPU has been carefully arranged to guarantee coalesced accesses to both the two-dimensional image data used in the update formulae as well as to the buffers holding the updates

for subsequent copy into the matrix data structure. The image data is stored in a two–dimensional dense array, hence its accesses are quite efficient since they do not involve indirect addressing; the number of arithmetic operations is however relatively low, hence this kernel is also memory–bound and adds difficulties in exploiting GPUs capabilities.

### *3.3. Linear Solvers*

As mentioned in Section 2, both $A(\mathbf{z})$ and $B(\mathbf{u})$ are $M$–matrices, therefore both Jacobi and Gauss–Seidel methods converge. It would therefore seem natural to employ the Gauss–Seidel method, since it has better convergence properties for matrices arising from discretizations of elliptic partial differential equations; however this does not necessarily mean that it will achieve the fastest overall solution time.

The reason for this particular behaviour lies in the architectural characteristics of GPU devices. Each individual processing element is slower than most commodity CPU cores; the speedup in the application is determined by the ability to keep many processing elements constantly busy. To achieve this, the GPU programming model is based on the usage of many threads, more than the available processing elements, relying on a very fast context switching mechanism; it is the responsibility of the application to make sure that there is enough thread–level parallelism to fully utilize the device.

In the sparse matrix–vector product it is common to assign the computations related to one row of the matrix to an individual thread; similarly, each thread is responsible for the computation of the sum of two elements of a vector, or of their products. If the matrix size is sufficiently large we automatically have enough parallelism to achieve a good speedup. The Jacobi method relies on the execution of a matrix–vector product of the off–diagonal part of the matrix, thus having one less nonzero element per row, and on the element–by–element product by the vector containing the inverses of the diagonal elements; it is therefore relatively easy to achieve a good speedup.

The situation is quite different with the Gauss–Seidel method. The solution of a sparse triangular system with the lower part of the coefficient matrix is a very difficult operation for the GPU; in principle, the solution of a triangular system has a strong dependency of each row on previous ones. In dense triangular systems this problem is usually overcome by proceeding in the computation by blocks of data, and parallelizing on the matrix–vector products corresponding to the off–diagonal blocks. However in the sparse matrix case, the off–diagonal blocks will be sparse, therefore the amount of work

to be distributed among threads will tend to be small; this is compounded with the fact that the structure of most sparse matrices, including the ones appearing in this application, has most nonzeros concentrated around the main diagonal, hence most of the work is in the diagonal blocks. Moreover, the amount of nonzeros in the lower triangle is half than that of the complete matrix, and similarly for the upper part; not only there are few nonzeros in the system solution, but also in the product by the upper triangle.

In summary, solving sparse triangular systems on a GPU is a difficult and inefficient task; in this situation, the Jacobi method enjoys a parallelization advantage over the Gauss–Seidel that more than compensates the increase in the number of iterations, especially for relatively well–conditioned systems and for low accuracy requests, as is the case in our application.

Another important consideration is that in our application, as we will see in Section 4, for usual parameter choices, the linear systems are not too difficult to solve; therefore the coefficient update phase actually becomes the most time consuming of the whole application. In this context, the Jacobi iteration requires the update of one iteration matrix (the off–diagonal part of the original $A(\mathbf{z})$ or $B(\mathbf{u})$), and of one vector containing the inverses of the diagonal elements. The Gauss–Seidel iteration instead requires two matrix updates, one on the lower part and one on the upper part. This is due to the fact that in most sparse storage formats it is too expensive to extract "on–the–fly" one triangle from a matrix: to perform multiplication by the lower triangle having the storage for the complete matrix, given that we do not have dense rows, we would have to inspect each and every element in a row to detect whether it lies in the lower or upper triangle. It is much more convenient to perform this screening only once and store the two triangles separately; however this results in the matrix update kernels being called twice, and therefore being generally more expensive, for Gauss–Seidel method compared to Jacobi also on CPU (see Section 4).

A final consideration is in order regarding the software implementation we propose: to substantiate the claim to expressiveness made in Section 1, we show in Fig. 1 the code for the Jacobi iteration. The code makes use of the following kernels:

- Matrix–vector product: `psb_spmm(alpha,a,x,beta,y,desc,info)` computes
$$\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$$

```
!
! Unroll the loop so that XIT and XV can exchange roles as
! x_{k} and x_{k+1}
!
itx = 0
do

  call psb_geaxpby(done,bv,dzero,xit,desc,info)
  call psb_spmm(-done,a_nd,xv,done,xit,desc,info)
  call xit%mlt(ainvd,info)

  nrmden = psb_genrmi(xit,desc,info)
  call psb_geaxpby(done,xit,-done,xv,desc,info)
  nrmdiff     = psb_genrmi(xv,desc,info)

  itx = itx + 1
  if ((nrmdiff <= tol*nrmden).or.(itx > maxit)) then
    call psb_geaxpby(done,xit,dzero,xv,desc,info)
    exit
  end if
  call psb_geaxpby(done,bv,dzero,xv,desc,info)
  call psb_spmm(-done,a_nd,xit,done,xv,desc,info)
  call xv%mlt(ainvd,info)

  nrmden = psb_genrmi(xv,desc,info)
  call psb_geaxpby(done,xv,-done,xit,desc,info)
  nrmdiff     = psb_genrmi(xit,desc,info)

  itx = itx + 1
  if ((nrmdiff <= tol*nrmden).or.(itx > maxit)) then
    exit
  end if

end do
```

Figure 1: Code for the Jacobi iteration

- Scaled vector sum: `psb_geaxpby(alpha,x,beta,y,desc,info)` computes
$$\mathbf{y} \leftarrow \alpha\mathbf{x} + \beta\mathbf{y}$$
- Vector norms: `nr = psb_genrmi(x,desc,info)`
$$nr = \|\mathbf{x}\|_\infty$$
- Element–wise vector product: `call x%mlt(v,info)`
$$x(i) \leftarrow x(i) \cdot v(i), \quad \forall i = 1, \ldots, n.$$

The argument `desc` keeps track in a transparent manner of the necessary communication steps required when the code is run as an MPI application; within each MPI process, the other arguments contain the data (vectors and matrices) that are involved in the local part of the computation. This code is therefore capable of running in either a distributed–memory environments based on multi–core CPUs, where MPI is available, or on cores paired with GPUs, with no modifications, indeed not even a recompilation, according to the dynamic type of the data that is passed inside the various sparse matrices. Note in particular that the choice of the data storage format is made at the application level by simply declaring a variable of the desired type; this approach works even for storage formats developed at a different time than the main library, as indeed is the case of our GPU plugin, thus allowing the freedom to experiment with many variations (for a complete discussion see [10]).

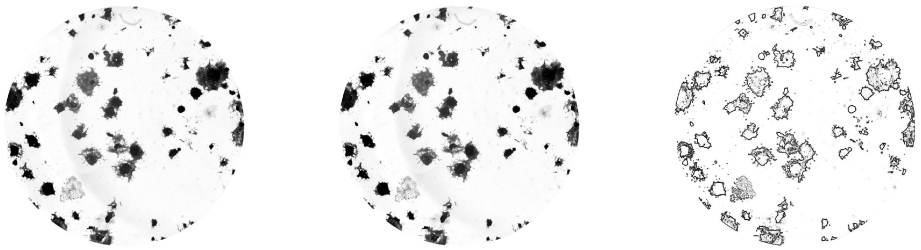
## 4. Performance Analysis

In the following we discuss results of the PSBLAS–based GPU implementation of Algorithm 1 with respect the corresponding sequential and parallel implementation on multi-core CPUs, when grayscale real images of mouse stem cell colonies of increasing size were segmented.

We ran our code on a hybrid node of the *yoda* cluster operated by the Naples branch of ICAR–CNR. Each computing node is composed of 2 eight–core CPUs Intel Sandy Bridge E5–2670 and 192 GB of RAM and it is equipped with a GPGPU Nvidia K20, with 2496 cores organized in 13 multiprocessors, equipped with 5GB of RAM. We built our code with the GNU

compilers suite version 4.9.1, a development version of PSBLAS 3.3 and of the PSBLAS-EXT plugin, and version 2.0 of MVAPICH.

In Fig. 2 we show the original image (a), the computed image (b) and the corresponding edge set (c), when the image size is $1024 \times 1024$, corresponding to a mesh–size $h = 0.0009$, $\alpha = \beta = 1.0$ and $\epsilon = 10^{-3}$, which satisfies the condition $h/\epsilon < 1$. The stopping criterion for Algorithm 1 was based on the maximum norm of the overall system residual and the numerical results were obtained when the requested accuracy was set to $10^{-10}$, while the inner linear iterations were stopped when the relative maximum norm of the difference between two successive linear solutions was smaller than $10^{-3}$.



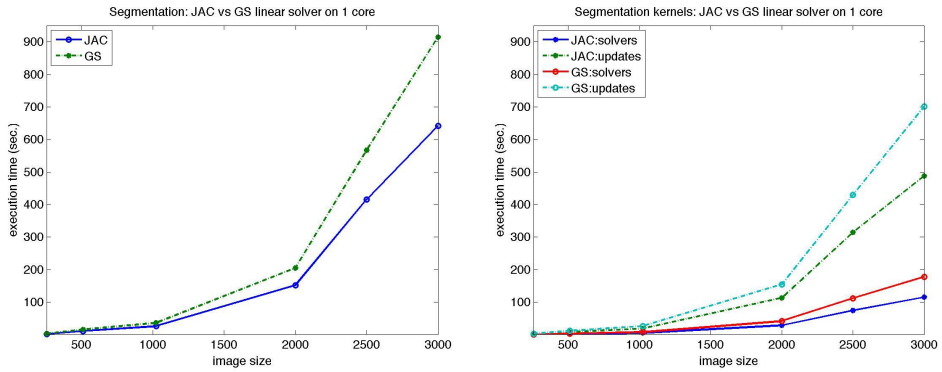
(a) original image (b) computed image (c) edge set

Figure 2: Segmentation Results

In the following we discuss performance results for increasing image sizes $nx = ny = 256, 512, 1024, 2000, 2500, 3000$ ($nx$ and $ny$ are the number of image pixels in each dimension). Model and algorithmic parameters are the same as in the case of the image in Fig. 2 per each size, therefore, for the first two image sizes the condition $h/\epsilon < 1$ is violated, but this allowed us in this discussion to release the choice of $\epsilon$ from the image size without observing significative changes in the obtained edge set.

In Fig. 3 we show the executions times of the segmentation codes based on the Jacobi (JAC) linear solver and on the Gauss–Seidel (GS) one, when they run on a single core of a CPU. In details, we show both the total execution times to reach convergence (on the left) and the execution times of the main kernels of the codes, i.e., accumulated times of linear solvers and of matrices updates (on the right). We can observe that, as already mentioned, the best total execution times are obtained when the JAC solver is applied. This behaviour is due to the small number of iterations needed to obtain

the requested accuracy (see discussion in [5]), for both the linear solvers on the inner linear systems. Indeed, since the most computational demanding kernels are the updates of the matrices at each non–linear iteration and, as discussed in Section 3.3, the application of GS requires more sparse matrix updates (two per each one of the linear systems) than the JAC solver, the best execution times per each phase of the code are observed when JAC linear solver is applied. On many common CPU models and across many different matrices the CSR format gives a very good speed for matrix-vector product; however for our application on the Intel Sandy Bridge processor, we found that the ELL format is more performant, although not by a very large a2mount. Hence all CPU runs have been performed using the ELL format.



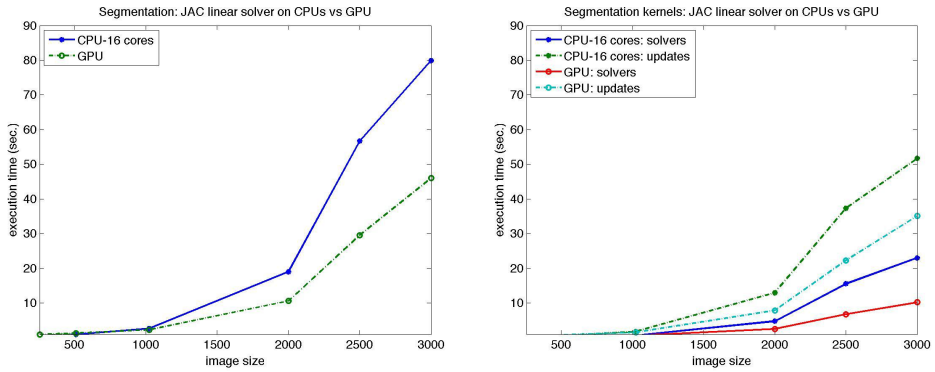

(a) Total times

(b) Main kernels

Figure 3: Execution Times: JAC vs GS linear solver on CPU.

We now shift our focus on the performance behaviour of our code, based on the JAC solver, when it runs in a parallel setting. We consider performances of the same code when it runs on the 2 eight-core CPUs of a single node and when it runs on a single GPU of the same node. In the multiple CPU case, data parallelism is introduced by a domain decomposition approach, i.e, by partitioning the image into 2D subimages and assigning a subimage to each available processing unit. This decomposition results into a a general row–block distribution of the matrices involved in the computation, since to each element of the subimage there corresponds a row in the sparse matrices involved in Algorithm 1. The general row–block distribution is the one assumed by PSBLAS for its parallelization by using MPI.

In Fig. 4 we show the execution times of the overall segmentation (on the left) and the execution times of the main kernels of the segmentation code (on the right). We can see that both parallel runs allow to obtain a large

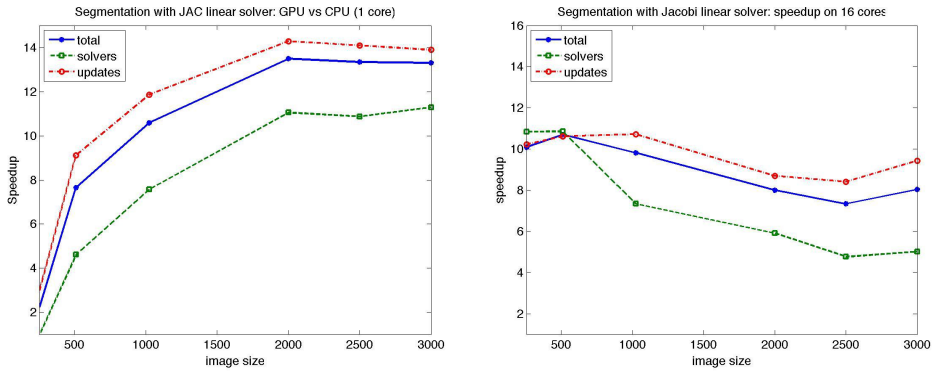



(a) Total times (b) Main kernels

Figure 4: Execution Times of parallel segmentation: GPU vs CPU-16 cores

reduction of the execution time with respect to the sequential run. Indeed, for the image with the largest size, total execution time is reduced from about 640 seconds on 1 core (see Fig. 3) to about 83 seconds and 46 seconds on the dual eight-core CPUs and on the GPU, respectively. We can observe that, as expected, the total execution time of the code on 16 cores is less than the execution time on the GPU when the size of the image is small. Indeed for the two smaller image sizes we have execution times of about 0.2 and 1 seconds on 16 cores, respectively, while on the GPU we have about 0.96 and 1.3 seconds, respectively. Very similar execution times of about 2.5 seconds can be observed for the image size of $1024 \times 1024$, while the execution times on the GPU becomes better than that on 16 cores for increasing image sizes. The CPU advantage at small sizes is related to the efficient use of the cache memory hierarchy, while memory bandwidth limits performance when image size increases. On the other hand, the SIMT architecture benefits of a larger number of available threads when the image size and hence the involved sparse matrix dimensions increase, leading to a speedup of about 2 on all the sufficiently large image sizes for the overall segmentation. If we look at the main kernels of the computation (see Fig. 4 right), we can observe that speedup is obtained by GPU both in the linear solvers, where

the execution time at largest size is reduced from about 23 seconds to about 10 seconds with respect to the run on 16 cores, and in the matrix updates kernels, where the execution time is reduced from about 52 seconds to 35 seconds. For our test cases and image sizes, we can conclude that GPU runs of the segmentation code allow to obtain better execution times when image sizes are sufficiently large if compared with parallel runs on the multi-core CPUs of a single node.




(a) GPU vs 1 core

(b) 16 cores vs 1 core

Figure 5: Speedup

Finally, in order to analyze the efficiency of the different versions of the parallel code in using the available resources, in Fig. 5 we show the speedup obtained against a single CPU core by the GPU code (on the left) and by the 16 cores of a single node (on the right), at varying matrix dimensions. The GPU code is capable of obtaining an overall speedup from about 11 to about 13 for typical test cases with sizes from $1024 \times 1024$. In more details, as expected, the solver kernels are able to exploit the parallel capabilities of GPU when image size increases, due to the increasing number of available threads efficiently using GPU resources, and speedup of the solver kernels rapidly increases with the image size till to reach a value of more than 11 in the case of the largest size. This value of the speedup was expected if we consider performance results of the sparse matrix-vector product available for our GPU and sparse matrices with the same sparsity pattern and dimensions of our problem (see [17, 19]), where maximum speedup between 9 and 11 were measured for different sparse matrix storage schemes. The best value of

about 11, corresponding to about 17 Gflops of measured performance with the ELL format. This is much less than the declared double-precision peak performance of 1100 Gflops on the K20; however we know that the SpMV kernel is memory-bound, and therefore it is more realistic to measure the sustained memory bandwidth. Our PSBLAS software has been measured sustaining 145 GB/s, which is reasonably good compared to the declared peak bandwidth of 208 GB/s, and also compared with a sustained bandwidth between 135 and 140 GB/s reported in [23] for the K20c. At this time there is no widely accepted way to define peak performance for sparse kernels, although some efforts in this direction look promising [24]. Finally, for the dense image matrix update kernels, we observe a speedup of about 14 for image sizes $1024 \times 1024$ or larger.

If we look at the parallel performance of the code on multi-core CPUs, we can observe a different behaviour of the speedup of the overall segmentation as well as of its main kernels, varying image sizes, with respect to the GPU code. As expected, speedup of the overall segmentation decreases with the image size due to the impact of memory bandwidth to the solver kernels. The best speedup of about 11 is obtained in the solver kernel for image size $512 \times 512$, while a speedup of about 9 is measured in the case of the largest image size. We can observe that also on CPUs the matrix updates obtain better speedup than the solvers since this computation, in a parallel distributed memory setting, has a better ratio between computation and communication, indeed, at each update phase, data communication is only required to updates matrix values corresponding to boundary points among different image partitions. However, the impact of memory bandwidth when dealing with increasing image size is also observed on the update phase, whose speedup decreases to values between 9 and 10 for image sizes larger than $2000 \times 2000$.

To conclude our discussion, we can observe that in both CPU and GPU the attained level of performance is quite low with respect to the theoretical peak performance; this is not too surprising given that most of the kernels are memory-bound. A single 8-core Sandy-Bridge node has a peak performance of 166 Gflops, or approximately 20 Gflops per core. However for a memory-bound kernel a more meaningful measure [24] would be obtained by computing the time to move the nonzero coefficients through the memory bus, and using this as a lower bound on the time for a sparse matrix-vector product. The Sandy-Bridge peak bandwidth is 51.2 GB/s; measuring the sustained single-core bandwidth with the STREAM benchmark from

`http://www.cs.virginia.edu/stream/` gives an effective rate of 13.5 GB/s, to which there corresponds a peak performance of 2.4 Gflops, not too far from our measured speed of 1.9 Gflops with the matrices arising from our application in the case of the largest image size. To have a better bandwidth utilization we need to use more than one core; indeed, the best performance we got on 8 cores for a single SpMV is about 4.8 Gflops, and going to the full 16-cores node, the best sustained computation rate is 7.4 Gflops, again for the largest image size. Note that for the smallest image size performance at 16 cores is much higher because the size is sufficiently small to fit most data into the (aggregated) cache memories, thus boosting the computation speed.

## 5. Concluding Remarks

In this paper we have described a parallel implementation for modern GPUs of a generalized relaxation method for image segmentation. Our main goal was to develop an efficient software module for accurate segmentation of thousands of images, as required in high-throughput screening platforms in Computational Biology.

Our implementation is based on an available software framework which allows efficient and transparent use of massively parallel architectures including heterogeneous nodes. We show that for a typical test case arising from biological experiments our GPU implementation is able to reach speedup in line with that available in the literature for memory-bound applications with respect to the corresponding multi-core CPU implementation. Our code is designed with modern features based on dynamic data type implemented in the base software platform, and therefore it can run without modifications on either CPU or GPU nodes in a way transparent to the end users.

Future work will include the extension of the application code for running on heterogeneous platforms made of multi-core and many-core nodes to be used for massively parallel image segmentation.

## 6. Acknowledgments

We would like to thank Dr. Laura Casalino of the Stem Cell Fate Laboratory at the Institute of Genetics and Biophysics (IGB) of the National Research Council (CNR) of Italy for images used in our analysis.

We also thank the reviewers for their insightful comments which helped us in improving our paper.

[1] A. Shariff, J. Kangas, L. P. Coelho, S. Quinn, R. F. Murphy, Automated image analysis for high content screening and analysis, J. Biomolec. Screening 15 (2010) 726–734.

[2] G. Aubert, P. Kornprobst, Mathematical Problems in Image Processing. Partial Differential Equations and the Calculus of Variations, 2nd ed., Springer-Verlag, New York, USA, 2006.

[3] L. Bar, et al., Mumford and Shah model and its applications to image segmentation and image restoration, in: Handbook of Mathematical Methods in Imaging, Vol. I, 2011, pp. 1095–1157.

[4] A. Vitti, The Mumford–Shah variational model for image segmentation: An overview of the theory, implementation and use, ISPRS J. of Photogrammetry and Remote Sensing 69 (2012) 50–64.

[5] P. D'Ambra, G. Tartaglione, Solution of Ambrosio-Tortorelli model for image segmentation by generalized relaxation method, Communications in Nonlinear Science and Numerical Simulation 20 (2015) 819–831.

[6] S. Filippone, M. Colajanni, PSBLAS: A library for parallel linear algebra computation on sparse matrices, ACM Trans. Math. Softw. 26 (4) (2000) 527–550.

[7] S. Filippone, A. Buttari, Object-oriented techniques for sparse matrix computations in Fortran 2003, ACM Trans. Math. Softw. 38 (4) (2012) 1–20.

[8] A. Buttari, P. D'Ambra, D. di Serafino, S. Filippone, 2LEV-D2P4: a package of high-performance preconditioners for scientific and engineering applications, Applicable Algebra in Engineering, Communication and Computing 18 (2007) 223–239.

[9] P. D'Ambra, D. di Serafino, S. Filippone, MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95, ACM Trans. Math. Softw. 37 (3) (2010) 7–23.

[10] V. Cardellini, S. Filippone, D. Rouson, Design patterns for sparse-matrix computations on hybrid CPU/GPU platforms, Scientific Programming 22 (1) (2014) 1–19.

[11] L. Ambrosio, V. M. Tortorelli, Approximation of functional depending on jumps by elliptic functionals via Γ-convergence, Comm. Pure Appl. Math. 43 (1990) 999–1036.

[12] A. Braides, Γ-convergence for Beginners, Oxford University Press, Oxford, UK, 2002.

[13] J. Ruge, K. Stüben, Algebraic multigrid, in: S. F. McCormick (Ed.), Multigrid Methods, Frontiers in Applied Mathematics, SIAM, Philadelphia, USA, 1987, pp. 73–130.

[14] Y. Saad, Iterative Methods for Sparse Linear Systems, II ed., SIAM, Philadelphia, USA, 2003.

[15] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, R. Doallo, General-purpose computation on GPUs for high performance cloud computing, Concurrency and Computation: Practice & Experience 25 (12) (2013) 1628–1642.

[16] N. Bell, M. Garland, Implementing sparse matrix-vector multiplication on throughput-oriented processors, in: Proc. of Int. Conf. on High Performance Computing, Networking, Storage and Analysis, SC '09, ACM, New York, NY, USA, 2009, pp. 18:1–18:11.

[17] D. Barbieri, V. Cardellini, A. Fanfarillo, S. Filippone, Three storage formats for sparse matrices on GPGPUs, Tech. Rep. DICII RR-15.6, Università di Roma Tor Vergata, `art.torvergata.it` (February 2015).

[18] F. Vázquez, G. Ortega, J. J. Fernández, E. M. Garzón, Improving the performance of the sparse matrix vector product with GPUs, in: Proc. of 10th IEEE Int. Conf. on Computer and Information Technology, CIT '10, 2010, pp. 1146–1151.

[19] D. Barbieri, V. Cardellini, A. Fanfarillo, S. Filippone, Sparse matrix-vector multiplication on GPGPUs submitted to (ACM Trans. Math. Softw.).

[20] R. Grimes, D. Kincaid, D. Young, ITPACK 2.0 user's guide, CNA-150, Center for Numerical Analysis, University of Texas (1979).

[21] NVIDIA Corp., CUDA cuSPARSE library, http://developer.nvidia.com/cusparse (2014).

[22] C. Gregg, K. Hazelwood, Where is the data? Why you cannot debate CPU vs. GPU performance without the answer, in: Proc. of 2011 IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS), 2011, pp. 134–144.

[23] PARALUTION User Manual, Tech. rep., www.paralution.com (2015).

[24] D. Langr, P. Tvrdìk, Evaluation criteria for sparse matrix storage formats, IEEE Transactions on Parallel and Distributed Systems DOI: 10.1109/TPDS.2015.2401575.