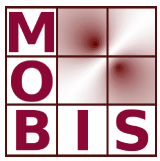
SpezialForschungsBereich F 32

Karl–Franzens Universität Graz
Technische Universität Graz
Medizinische Universität Graz



Comparing CUDA and OpenGL implementations for a Jacobi iteration

Ronan Amorim Gundolf Haase

Manfred Liebmann Rodrigo Weber dos Santos

SFB-Report No. 2008–025 Dec. 2008

A–8010 GRAZ, HEINRICHSTRASSE 36, AUSTRIA

Supported by the
Austrian Science Fund (FWF)

SFB sponsors:

- **Austrian Science Fund** (FWF)
- **University of Graz**
- **Graz University of Technology**
- **Medical University of Graz**
- **Government of Styria**
- **City of Graz**

Comparing CUDA and OpenGL implementations for a Jacobi iteration

Ronan Amorim Gundolf Haase Manfred Liebmann
Rodrigo Weber

December 19, 2008

Abstract

The use of the GPU as a general purpose processor is becoming more popular and there are different approaches for this kind of programming. In this paper we present a comparison between different implementations of the OpenGL and CUDA approaches for solving our test case, a weighted Jacobi iteration with a structured matrix originating from a finite element discretization of the elliptic PDE part of the cardiac bidomain equations. The CUDA approach using textures showed to be the fastest with a speedup of 78 over a CPU implementation. CUDA showed to be an efficient and easy way of programming GPU for general purpose problems, though it is also easier to write inefficient codes.

1 Introduction

As the performance of modern graphics hardware increases and becomes more flexible in terms of programmability many researchers apply this new technology to problems previously solved on CPUs. The graphics processor unit (GPU) consists of a set of multiprocessors designed to obtain the best performance with graphics computing. Nevertheless, its computational power can be used for general purpose computing.

Although the GPU programming for general purpose (GPGPU) are becoming more popular because of its promise of massive parallel computation, extracting a good performance of its processors is not always a simple task. In this paper we will present a comparison between two different approaches for GPU programming. The first is the OpenGL approach that uses the same programming resources avaiable as for graphics computing. The second approach uses the CUDA technology that is a C programming environment for GPU programming developed to make GPGPU easier [2].

First the problem used as test case will be presented and then the matrix structure derived from the problem is introduced. The next sections will describe the implementations on CPU, GPU with OpenGL and GPU with CUDA, respectively. Finally the results will be presented followed by the conclusion.

2 Original Problem

The problem chosen as test case is the solution of the bidomain equations that originates from the cardiac eletrophysiology modelling. There are two different components in the eletric propagation in the heart. The first is the model that describes the ionic flux through the cell membrane [1]. The second is the eletrical model for the tissue, that describes how the currents from a region of the membrane interact with the others. The bidomain equations are presented in Eqs. 1, 2 and 3.

$$\nabla \cdot (\bar{\sigma_i} + \bar{\sigma_e}) \nabla V_e = -\nabla \cdot \bar{\sigma_i} \nabla V_m \tag{1}$$

$$\nabla \cdot \bar{\sigma_i} \nabla V_m = -\nabla \cdot \bar{\sigma_i} \nabla V_e + \beta I_m \tag{2}$$

$$I_m = C_m \frac{\partial V_m}{\partial t} + I_{ion}(V_m, v) \tag{3}$$

where $\bar{\sigma_i}$ and $\bar{\sigma_e}$ are the intracellular and extracellular conductivity tensors, i.e., 3x3 symmetric matrices that vary in space and describe the anisotropy of the cardiac tissue. β is the surface to volume ratio of the cardiac cells, C_m is the membrane capacitance per unit area, V_m is the transmembrane voltage. I_{ion} is the ionic current density flowing through the membrane ionic channel and depends on the transmembrane voltage and several other variables that are represented here by v. The media by themselves are linear, with nonlinearities arising through the current-voltage relationship across the membrane (Eq. 3) which is described by a set of nonlinear Ordinary Differential Equations (ODEs). The system of Eq. 3 typically accounts for over 20 variables, such as ionic concentrations, protein channel resistivities and other cellular features.

At this point, the bidomain equations may be considered as a coupled set of elliptic Partial Differential Equation (PDE), Eq. 1, parabolic PDE, Eq. 2 and non-linear ODEs Eq. 3. Due to the highly nonlinear nature of the ODEs, fully implicit solutions are extremely difficult. Operator splitting technique is usually performed such that the numerical solution is reduced to a modular three-step scheme which involves the solutions of a parabolic PDE, an elliptic PDE and a nonlinear system of ordinary differential equations (ODEs) at each time step [4].

The domain of our problem is 2D and the discretization of the PDEs is obtained from the Finite Elements Method using square elements with bilinear interpolation. This discretization method leads to a symmetric matrix which is sparse with a main diagonal, 4 lower diagonals and 4 upper diagonals, giving a total of 9 diagonals. Figure 1 presents the results of a simulation using the same parameters as we will use.

This paper focusses on the comparison between two different GPGPU approaches for solving the linear system of equations and therefore we will use for simplicity the diagonals from the elliptic PDE and the right hand side provided by the first time iteration of an external bidomain solver.

We will implement the weighted Jacobi method to solve this linear system given by the diagonals and the right hand side. This method is not a good choice for solving this problem efficiently. Nevertheless, we are not interested in the convergence rate for this problem but to provide some comparison between two

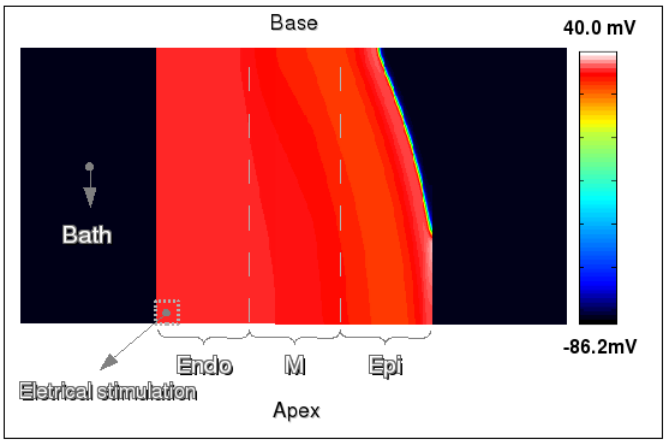


Figure 1: Visualization of a cardiac simulation in 2D.

different GPU programming approaches. The advantage of using the weighted Jacobi is its inherent parallelism. The weighted Jacobi iteration is described in Alg. 1.

for $i = 1, 2, \ldots, number\_steps$ **do**
  $\bar{x}_i s = \bar{x}_{i-1}(1-\omega) + \omega D^{-1}(f - (L+U)\bar{x}_{i-1})$
end

Algorithm 1: Weighted Jacobi iteration.

where $\bar{x}_i$ is the approximate solution vector in iteration step i. L, U and D are respectively the lower, upper and main diagonals of the matrix. And ω is a scalar for the weight.

3 Data storage

As was described in the previous section the discretization leads to a matrix with 9 diagonals. Although the matrix is symmetric, we store all the 9 diagonals to keep the simplicity and to achieve a good performance, since, in this manner, we do not need a different indexing for different diagonals. Thus, all diagonals in our storage format have the same size, with the beginning of the lower diagonals being filled with zeros and the same for the ending of the upper diagonals. The matrix storage is illustrated on Fig. 2, and shows a simplication with only 3 diagonals, one main diagonal D, one lower diagonal L and one upper diagonal U. The figure presents how the diagonals are filled with zeros to make the matrix multiplication easier without the need of a different indexing or expensive range tests during runtime.

For the multiplication, our approximation can be viewed as a 9 stencil on 2D. Fig. 3 presents the layout of our 9 point stencil, where the element with the circle will be multiplied by the main diagonal and each other by another diagonal, in the case of a matrix vector multiplication.

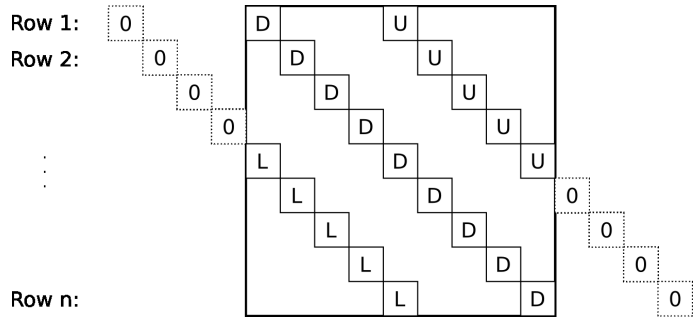


Figure 2: Matrix storage format. It is a simplification with only 3 diagonals instead of 3 by 3 diagonals.

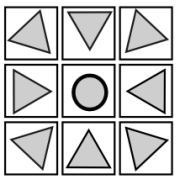


Figure 3: 9 point stencil.

4 CPU implementation

A CPU version of the weighted Jacobi was also implemented to be compared with the GPU implementations. The matrix storage format is mainly the same of the storage described in section 3 but now, all the diagonals are stored in one single vector with a different ordering to improve the cache hit rate. Instead of concatenating all the diagonals in one vector, the rows of the matrix are concatenated in such a way that the *ith* element of the first diagonal is followed by the *ith* element of the second diagonal and so on. The Fig. 4 presents the matrix storage used on the CPU version of the iterative solver. It is again a simplification with only 3 diagonals. Among several different storage formats that were implemented for the CPU version, this showed to be the most efficient, because of the locality of the matrix data for each iteration step.

The algorithm for the CPU version was implemented using the C programming language and the GCC 4.1.3 compiler. Fig. 5 presents the C code used to implement the weighted jacobi on the CPU.

5 GPU implementation using OpenGL

The OpenGL approach is the most complicated for those who do not have an understanding of computer graphics. The main idea behind using OpenGL to perform general purpose computation is to map some of the computer graphics concepts into CPU programming concepts.

The basic idea is that in computer graphics we have textures that are used as input data for the rendering. And the rendering draws to the frame buffer. So now we can use this machine of rendering graphics to perform some general

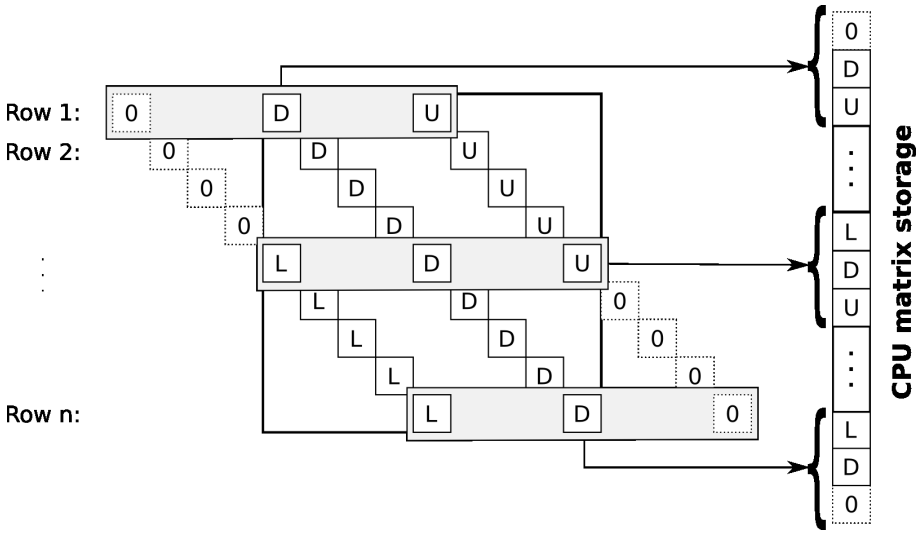


Figure 4: CPU matrix storage. The elements of each row of the diagonals are stored sequentially.

purpose computation. Therefore, for our GPGPU approach, the textures will be used to provide the input data necessary to our computation, the rendering will perform the computation and, finally, the results will be written in the frame buffer. A very simplified point of view of the GPU programming using OpenGL, there are much more details involved in this task.

The graphics hardware is implemented as a pipeline, and, in recent GPUs, there are some stages of this pipeline that can be programmed. For example, the vertex processor and the fragment processor. The fragment processor will be used in our implementation since it fits better to the problem and it is the most powerful processor on the GPU.

The fragment processor executes a fragment shader, that is a program written to the fragment processor [3]. The fragment shader is responsible for calculating the color of individual pixels, and, in our case, for calculating the new approximation for individual elements of the solution vector. The shader can only write its own position, or in other words, scatter writes are not possible. But textures can be randomly accessed though is better to keep the access with a certain locality because of the texture cache.

Now, with these basic concepts we can go further. A set of OpenGL extensions provides a 32 bit floating point support and also allows us to perform an offscreen rendering. Then we need to choose an orthogonal projection and a proper viewport configuration that will provide a mapping of 1 texel to exactly 1 pixel because we do not want any interpolation being done with our input and output data. The Fig. 6 shows a piece of code demonstrating how to make the mapping and to generate the offscreen framebuffer for computing.

Usually, we attach a texture to the framebuffer to write the results of the computation on. Nevertheless, is not possible to use a texture as input and output at the same time. The strategy for implementing the weighted Jacobi with OpenGL is to implement the fragment shader as one step of the weighted

```
for(int k = 0;k <numits; k++)
{
        diag_p = diag0;
        for(int i=0; i<N; i++)
        {
                offset = i*9;
                xout[i] = xin[i]*oneminusw + w*(f[i] - (*(diag_p+4)*xin[i-nx-1] +
                                                        *(diag_p+3)*xin[i-nx] +
                                                        *(diag_p+2)*xin[i-nx+1] +
                                                        *(diag_p+1)*xin[i-1] +
                                                        *(diag_p+5)*xin[i+1] +
                                                        *(diag_p+6)*xin[i+nx-1] +
                                                        *(diag_p+7)*xin[i+nx] +
                                                        *(diag_p+8)*xin[i+nx+1]))/ *(diag_p);
                diag_p+=9;
        }
        xswap = xin;
        xin = xout;
        xout = xswap;
}
```

Figure 5: Code section from weighted Jacobi CPU implementation.

```
/// viewport transform for 1:1 pixel-texel - data mapping
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, width, 0.0, height);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glViewport(0, 0, width, height);

/// creating the Framebuffer and setting up the offscreen
/// rendering
glGenFramebuffersEXT(1, fb);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, *fb);
```

Figure 6: Example showing how to set up the mapping and the offscreen rendering in OpenGL.

Jacobi. Then, after one computation, the input texture is switched with the output texture, and this process continues until the last step of the iterative solver. This swapping technique with the textures is also known as ping-pong technique. There are several languages for implementing shaders on GPU, the language chosen for this implementation was the GLSL [3].

Now we will describe two different implementations of the weighted Jacobi using OpenGL. And we will focus on the fragment shader implementation.

5.1 OpenGL implementation using Luminance Texture

Since the graphics hardware is build to work on graphics, the textures are usually chosen to hold 4 values (the RGBA channels) for each texture element, and the hardware is optimized to deal with these four elements at once in the shader processor. Although using 4 elements at once could be better to extract more performance, it can be tricky to write the shader to deal with the four values.

An alternative is to use the luminance texture that allows us to deal with only one element on the fragment shader. Using the luminance texture keeps the fragment shader very simple. The Fig. 7 presents some pieces of the code of the fragment shader for this implementation.

```
uniform float weight;


void main(void)
{
        vec2 topleft = gl_FragCoord.xy;
        /// loading old solution approximation
        float x    = texture2DRect(texture_x_old, topleft).x;
                                (...)
        float x_u8 = texture2DRect(texture_x_old, topleft + vec2( 1, 1)).x;

        /// loading right hand side
        float f = texture2DRect(texture_f, topleft).x;

        /// loading matrix values
        float diag_l4 = texture2DRect(texture_diag4_0, topleft).x;
                                (...)
        float diag_u8 = texture2DRect(texture_diag8_0, topleft).x;

        /// computing new approximation for element in topleft coordinate
        gl_FragColor.x = (1.0-weight)*x + weight*(f - (
                                diag_l4*x_l4 +  diag_u3*x_u3 +
                                diag_l2*x_l2 +  diag_u1*x_u1 +
                                diag_l5*x_l5 +  diag_u6*x_u6 +
                                diag_l7*x_l7 +  diag_u8*x_u8))/diag;
}
```

Figure 7: Some pieces of code of the fragment shader.

The place holders for similar code as above and below that line, (...), are not part of the fragment shader code, but are there to show that in that part of the code the same kind of operation was done for other data. The only purpose is to keep the code compact to be presented here. The *texture2DRect* fetches data from the texture at the *topleft* position. The *gl_FragColor.x* is the element receiving the computation. It is also possible to write to multiple render targets, i.e., on each shader you can write different values to more than one texture. But in our case it is not necessary to use this technique since we only need to compute the new approximation of the solution vector. Using the *luminance* texture is indicated by the *.x* since we are only reading and writing to one of the four available channels (.xyzw, or .rgba). One advantage of the graphics hardware with textures is that we do not need to worry about an out of range access at the border since the values will be clamped or repeated depending on how you configure your texture. The additional zeros in the diagonals, see Fig. 2, avoid invalid data calculation and therefore, we can benefit from the acceleration by using textures.

5.2 OpenGL implementation using RGBA Texture

We need to be careful using the RGBA texture because of the access pattern of our solution vector when loading the data to the computation. On Fig. 8

we present the access pattern on the texture for the old solution approximation since loads and writes 4 values at once. Because of the access pattern showed on section 3 it is necessary to arrange the data in such a way that is possible to perform all the computation on one instruction instead of computing separately for each channel on the RGBA channels.

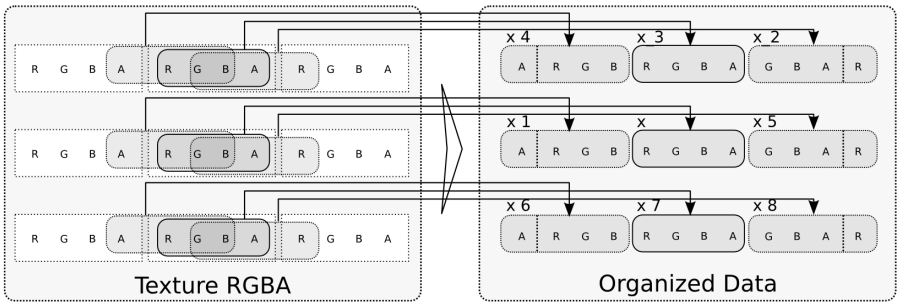


Figure 8: Changing the data storage to perform all the computations at once.

A piece of the shader code that performs this data change is shown on Fig. 9. The only differences between the two GLSL shaders, is the data reordering and the luminance shader, we use, to load the data and to store the data, only 1 channel, with $.x$, now we do not need this since we will use all the 4 channels.

```
void main(void)
{
        vec2 topleft = gl_FragCoord.xy;

        /// loading old solution approximation
        vec4 x_l4 = texture2DRect(texture_x_old, topleft + vec2(-1,-1));
        vec4 x_l3 = texture2DRect(texture_x_old, topleft + vec2( 0,-1));
        vec4 x_l2 = texture2DRect(texture_x_old, topleft + vec2( 1,-1));
        vec4 x_l1 = texture2DRect(texture_x_old, topleft + vec2(-1, 0));
        vec4 x     = texture2DRect(texture_x_old, topleft);
        vec4 x_u5 = texture2DRect(texture_x_old, topleft + vec2( 1, 0));
        vec4 x_u6 = texture2DRect(texture_x_old, topleft + vec2(-1, 1));
        vec4 x_u7 = texture2DRect(texture_x_old, topleft + vec2( 0, 1));
        vec4 x_u8 = texture2DRect(texture_x_old, topleft + vec2( 1, 1));

        /// loading right hand side
        vec4 f = texture2DRect(texture_f, topleft);

        /// loading matrix values
        vec4 diag_l4 = texture2DRect(texture_diag4_0, topleft);
                              (...)

        /// putting the data on the proper order
        x_4.r = x_4.a;          x_4.gba = x_3.rgb;
        x_2.a = x_2.r;          x_2.rgb = x_3.gba;
        x_1.r = x_1.a;          x_1.gba = x.rgb;
        x_5.a = x_5.r;          x_5.rgb = x.gba;
        x_6.r = x_6.a;          x_6.gba = x_7.rgb;
        x_8.a = x_8.r;          x_8.rgb = x_7.gba;

        /// computing new approximation for element in topleft coordinate
        gl_FragColor = (1.0-weight)*x + weight*(...))/diag;

}
```

Figure 9: Piece of the fragment shader with the data reordering.

6 GPU implementation using CUDA

The CUDA programming model consists of extending the C language by allowing the programmer to define C functions, called *kernels*, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions [2].

Each thread is given a unique ID that is accessible within the kernel through a built-in variable. The thread ID is a 3-component vector, so that the the threads can be identified uniquely using one-dimensional, two-dimensional, or three-dimensional index. Threads are separated by blocks, so each block has its own threads and executes in a multiprocessor. Kernels in different blocks cannot communicate or exchange data. Threads within a block can cooperate among themselves by sharing data through some shared memory and synchronizing their execution to coordinate memory accesses. This is one of main advantages of using CUDA instead of OpenGL, since these cooperations cannot be achieved using the OpenGL approach. Fig. 10 shows a very simple CUDA code that is capable of assign a scalar to the whole vector.

```
#include <stdio.h>

__global__ void assign_scalar(float *vec, float scalar){
        int tid = blockDim.x*blockIdx.x + threadIdx.x;
        vec[tid] =  scalar;
}

int main()
{
        dim3 dimBlock(256);
        float *vec_dev;
        cudaMalloc((void**)&vec_dev, sizeof(float)*dimBlock.x);
        assign_scalar<<<1, dimBlock>>> (vec_dev, 3.0);
        return 0;
}
```

Figure 10: Very simple CUDA program. Just assign a scalar to the whole vector.

The CUDA code must be compiled using the CUDA Compiler. And here is the disadvantage of CUDA over the OpenGL approach since the CUDA is only available for NVIDIA hardware.

CUDA also provides more flexibility for the memory usage. There are different memory access possibilities, such as global memory, shared memory and textures, each one has its advantages and disadvantages as will be shown in the next sections.

6.1 CUDA weighted Jacobi simple implementation

The first attempt of of a CUDA implementation uses simply the same code and data structure for the solution vector as on the CPU. We will use only the global memory for loading and writing the data. Using the global memory faces the problem on accessing the elements on the domain boundary since we will have out of the range accesses and it will probably crash the program or introduce wrong values as NaN which are not zero when multiplied by the zero of the

diagonals on these boundary areas. Therefore, we should allocate more memory in the same fashion as on CPU implementation and for the same reasons. The kernel for this implementation is shown on Fig. 11.

```
__global__ void wjacobi(float *diag0_4, float *diag0_3, float *diag0_2,
                        float *diag0_1, float *diag0_0, float *diag0_5,
                        float *diag0_6, float *diag0_7, float *diag0_8,
                        float *x, float *x_out, float *f, float weight,
                        int numits, int nx, int ny)
{
        const int i =  blockDim.x*blockIdx.x + threadIdx.x;
        const int k = i + nx + 1;
        const int N = nx*ny;

        if(i<N){
                x_out[k] = (1.0 -weight)*x[k] + weight*(f[i]- ( diag0_4[i]*x[k-nx-1] +
                                                                diag0_3[i]*x[k-nx  ] +
                                                                diag0_1[i]*x[k   -1] +
                                                                diag0_5[i]*x[k   +1] +
                                                                diag0_6[i]*x[k+nx-1] +
                                                                diag0_7[i]*x[k+nx  ] +
                                                                diag0_8[i]*x[k+nx+1]))/diag0_0[i];
        }
}
```

Figure 11: Cuda weighted jacobi skipping the boundaries as the CPU code.

We can see on Fig. 11 that the diagonals are stored in 9 different vectors in the CUDA code because no cache is available for this kind of memory access. Due to the boundary the indexing for accessing the x vector is different. The x index is summed with $nx + 1$, where nx is the width of the domain. This simple CUDA code is capable to perform our computation correctly. But the performance is very limited due to uncoalesced memory access caused by the indexing of the x vector. The coalesced memory access will be explained on the next section and a new version of the code improving the coalesced memory will be presented.

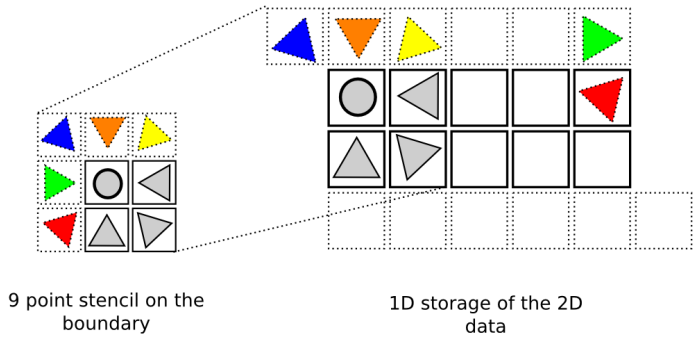


Figure 12: Data storage for the simple weighted Jacobi.

Fig. 12 shows that our data is not stored in 2D as expected but in 1D. Thus we do not need to allocate more memory for laterals boundaries just the width size plus 1 for the beginning of the vector and the same for the vector ending. It is possible because our diagonals are filled with zeros at the elements that

belongs to the boundaries. We only need to take care of not accessing data out of range and filling the ghost data of the boundaries with some valid value.

6.2 CUDA weighted Jacobi and coalesced memory access

Our previous version of CUDA code suffers from uncoalesced memory access for all x vector accesses, for writing and reading. Accordingly to [2] the multiprocessor on the GPU creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps. The global memory access by all threads of a half-warp is coalesced into one or two memory transactions if it satisfies the following three conditions:

1. All threads must access
   - Either 32-bit words, resulting in one 64-byte memory transaction,
   - Or 64-bit words, resulting in one 128-byte memory transaction,
   - Or 128-bit words, resulting in two 128-byte memory transactions;
2. All 16 words must be located in the same segment of a size equal to the memory transaction size.
3. Threads must access the words in sequence: The k^{th} thread in the half-warp must access the k^{th} word.

Since our data is a float with 32-bits we have one memory transaction of 64 bytes, or 16 words of 32-bits. The second and third condition are the problems of our previous CUDA implementation. When we skip the $nx + 1$ elements it leads immediately to uncoalesced memory access for all x accesses, since the 16 words will not be located in the same segment of size equal to the memory transaction size that is 64 bytes. And the k^{th} thread will access the k^{th} word only if the nx is equal to a multiple of 16 minus 1 element. For example if nx is equal to 31 the writing operation will be coalesced.

Our objective in this implementation is to achieve coalesced memory access at least for the writing and for reading one element of the old solution vector. To accomplish this we need to align the the memory skipping to be multiple of 16. Therefore, instead of using $nx + 1$ to skip the boundary we should have an offset as is given below:

```
offset = (nx + 1 + (16 - 1))/16 * 16;
```

This code uses integer division so the $offset$ will be the smaller multiple of 16 that fits $nx + 1$.

Although we solved the coalesced memory problem for writing and reading one element of the old approximate solution vector, we will still have coalesced memory that will depend on the width of our domain. On Fig. 13 the coalesced memory access is shown on green, the width dependent coalescing in yellow and the uncoalesced in red, for our 9 stencil.

It is important to say that even in the best case we will have only 3 coalesced memory accesses if one of the yellow elements in the same row of the stencil become coalesced due to the width of the domain then the others are necessarily uncoalesced.

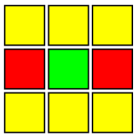


Figure 13: Green represents coalesced, yellow width dependent coalesced and red uncoalesced memory accesses.

Since we still have the width constraint, the next step is to achieve a coalescing independent of the width of the domain which always results in 3 coalesced memory accesses for the 9 point stencil.

6.3 CUDA weighted Jacobi in a 2D grid

Now we want to make sure that we will have 3 coalesced memory accesses for our 9 stencil. CUDA provides an easy way to pad the memory, ensuring that the allocation for a given width is appropriately padded to meet the alignment requirements, i.e., more memory is allocated at the end of each row to ensure the data alignment on the next row. In order to take advantage of this padding we should use a 2D grid since it will support the correct indexing of the elements. The function *cudaMallocPitch* is responsible for allocating the padded memory. On Fig. 14 the coalesced memory access is shown on green and the uncoalesced on red, for our 9 stencil with the padded memory.

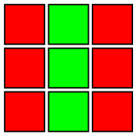


Figure 14: Green represents coalesced and red uncoalesced memory accesses for the padded memory.

To simplify the kernel, now instead of skipping the $width + 1$ or $width + some\_alignment\_to\_16\_words$, we skip 2 rows of the memory padded. It is important to notice that the other data structures such as the diagonals of the matrix and the right hand side of the vector must also be padded. Despite the grid is 2D, we are still using 1D data blocks which will be important to the next implementation.

Now we ensure that our coalescing will always be the same independently of the width of the domain. But we do not have yet a good implementation since there is only $\frac{1}{3}$ of coalesced memory access on the 9-stencil. An idea to improve this coalescing is to use the *shared memory* and this approach will be presented in the following implementation.

6.4 CUDA weighted Jacobi using shared memory

The shared memory is much faster then the global memory because it is on-chip. In fact, for all threads of a warp, accessing the shared memory is as fast

as accessing a register as long is there are no bank conflicts between the threads itself [2].

Now we will use the shared memory to avoid some uncoalesced memory access. The idea behind is that the data in shared memory is shared between all the threads within a block. Therefore the only threads that will have uncoalesced memory access will be the first and the last threads of each block. See Fig. 15 for details.

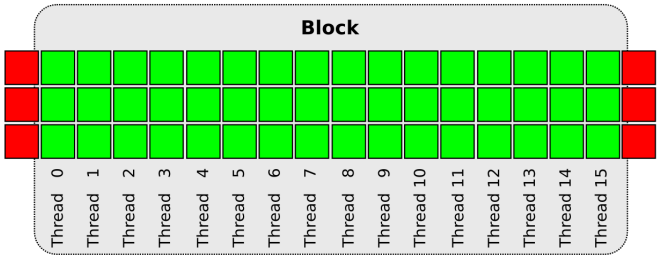


Figure 15: Green represents coalesced and red uncoalesced memory accesses for loading from global memory to shared memory.

Each thread within a block will load those elements from the x vector that are green on Fig. 14 from the global memory to the shared memory. This guarantees for the implementation in Sec. 6.3 that the access will be coalesced. Thus, only the first and last thread within the block will have uncoalesced reads. This strategy reduces the number of uncoalesced memory access drastically. The Fig. 16 shows a section of the kernel implementation that loads the data to the shared memory and in the end the threads are synchronized to ensure that all threads have already loaded their data part to the shared memory.

6.5 CUDA weighted Jacobi using textures

A simple strategy to avoid the coalesced memory access without too much effort is to use textures for the x vector. The textures are cached, differently to the global memory. This cache is optimized for 2D locality which supports perfectly the data access pattern of the 9-point stencil. Fig. 17 presents the kernel for the texture version of the algorithm.

We used 1D grids and blocks in this implementation because the 1D CUDA textures have some better properties than the 2D textures for our purposes. We must use 1D textures for texturing from linear memory such that is possible to swap the output data to be the next input data for the iterative solver, without the need of copying the memory for the next input in each step.

7 Results

For comparison purposes we executed each algorithm presented in the previous sections with 1000 steps of the weighted Jacobi. Only the Jacobi steps have been used for timing, the setup of the problem and the memory transfer from CPU to GPU are not included in the execution time. In the tables the CPU implementations will be referred as CPU, the OpenGL code using luminance textures

```
/// allocate shared memory
__shared__ float xl[BLOCKDIMX +2];
__shared__ float xd[BLOCKDIMX +2];
__shared__ float xu[BLOCKDIMX +2];

const int iloc = threadIdx.x + 1;

/// copy data from global GPU memory
/// into shared memory in the block
xl[iloc] = x[tidx-px];
xd[iloc] = x[tidx];
xu[iloc] = x[tidx+px];

if (threadIdx.x == 0){
        xl[0] = x[tidx-px-1];
        xd[0] = x[tidx-1];
        xu[0] = x[tidx+px-1];
}
if (threadIdx.x == blockDim.x-1){
        xl[blockDim.x+1] = x[tidx-px+1];
        xd[blockDim.x+1] = x[tidx+1];
        xu[blockDim.x+1] = x[tidx+px+1];
}
__syncthreads();
```

Figure 16: Section of code that fetches data from global GPU memory into local shared memory in the block.

as OGL_Luminance, the OpenGL code using RGBA textures as OGL_RGBA, the first CUDA code as CUDA_Simple, the second as CUDA_Aligned, the third as CUDA_2D, the fourth as CUDA_Shared and the last one as CUDA_Texture.

Three different hardwares environments have been used for comparison. The first is a Dual-Core AMD Opteron with a GeForce 8600GT graphics card, the second is an AMD Athlon X2 4200+ with a GeForce 8800GT graphics card, the third is an AMD Phenom(tm) 9950 Quad-Core Processor with the GTX280 graphics card. The hardware environments will be referred in the text as OP_8600GT, AT_8800GT and PH_280, respectively.

The domains sizes have always the height fixed with 1024 elements and only the width is changed, since the height has no influence in the coalesced memory access for our implementations. All the implementions use single precision floating points. For the CUDA code the block size choosen was 64 remembering that our 2D implementation also uses a 1D block.

Our first comparison is between the OpenGL codes. Fig. 18 shows that using RGBA textures have some performance advantage over luminance textures only for the graphics card 8600GT that is the least powerfull in our hardware environments. It was not possible to execute the OpenGL codes on the PH_280 since it is dedicated for CUDA.

The first tests with CUDA code showed the influence of the width in the coalesced memory accesses. As we already know from section 6.2 the CUDA_Simple implementation has more coalesced memory accesses only when the width of the domain is a multiple of 16 minus 1 element. And the same happens for the CUDA_Aligned in this case and when the width of the domain is multiple of 16 plus 1 element. Fig. 19 shows this behavior and shows the improvement of

```
texture <float, 1, cudaReadModeElementType> xtex;

__global__ void wjacobi(float *diag0_4, float *diag0_3, float *diag0_2,
                        float *diag0_1, float *diag0_0, float *diag0_5,
                        float *diag0_6, float *diag0_7, float *diag0_8,
                        float *x_out, float *f, float weight, int numits, int nx, int ny)
{
        const int i =  blockDim.x*blockIdx.x + threadIdx.x;
        const int N = nx*ny;

        if(i<N){
                x_out[i] = tex1Dfetch(xtex, i)*(1.0-weight) +
                          weight*(f[i]-(tex1Dfetch(xtex, i-nx)*diag0_3[i]+
                                        tex1Dfetch(xtex, i+nx)*diag0_7[i]+
                                        tex1Dfetch(xtex, i-1)*diag0_1[i]+
                                        tex1Dfetch(xtex, i+1)*diag0_5[i]+
                                        tex1Dfetch(xtex, i-nx-1)*diag0_4[i]+
                                        tex1Dfetch(xtex, i-nx+1)*diag0_2[i]+
                                        tex1Dfetch(xtex, i+nx-1)*diag0_6[i]+
                                        tex1Dfetch(xtex, i+nx+1)*diag0_8[i]))/diag0_0[i];
        }

}
```

Figure 17: Cuda kernel using textures as input data.

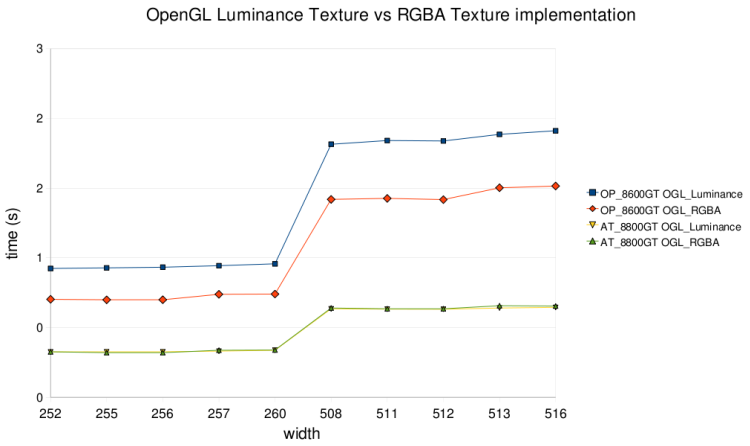


Figure 18: OpenGL implementations comparison

CUDA_2D when this width requirement is not fulfilled. We see also that the coalesced memory requirement is relaxed in the new graphics card GTX280.

Table 1 just shows a subset of the results in Fig. 19 with width of 508, 511, 512, 513 and 516.

We have already showed that our first and second implementations has been improved by data alignment. The next step is to compare CUDA_2D with CUDA_Shared. Fig. 20 shows that the shared memory really improves the performance.

Table 2 shows again a subset of the results of Fig. 20 with width of 508, 511, 512, 513 and 516.

As we can see, the CUDA_Shared really improves the performance of our

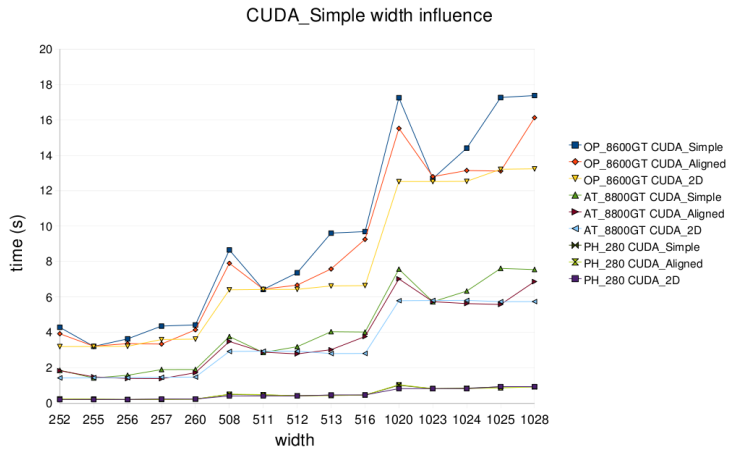


Figure 19: Width interference in CUDA_Simple.

Table 1: Timings for CUDA_Simple and CUDA_Aligned.

Hardware	Implementation	Width				
		508	511	512	513	516
OP_8600GT	CUDA_Simple	8,658s	6,429s	7,371s	9,608s	9,694s
OP_8600GT	CUDA_Aligned	7,907s	6,450s	6,674s	7,586s	9,258s
OP_8600GT	CUDA_2D	6,412s	6,427s	6,438s	6,626s	6,647s
AT_8800GT	CUDA_Simple	3,769s	2,875s	3,199s	4,047s	4,023s
AT_8800GT	CUDA_Aligned	3,496s	2,890s	2,797s	3,029s	3,773s
AT_8800GT	CUDA_2D	2,931s	2,944s	2,948s	2,809s	2,817s
PH_280	CUDA_Simple	0,509s	0,479s	0,431s	0,468s	0,468s
PH_280	CUDA_Aligned	0,520s	0,495s	0,423s	0,442s	0,471s
PH_280	CUDA_2D	0,420s	0,420s	0,421s	0,475s	0,477s

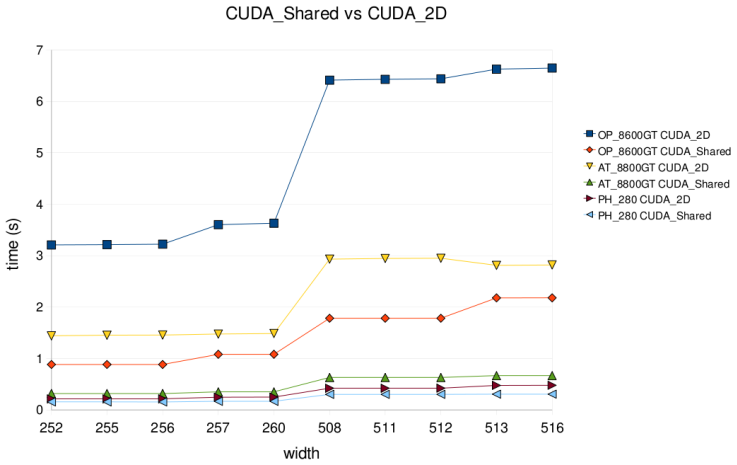


Figure 20: Time comparison between CUDA_Aligned and CUDA_Shared.

algorithm in comparison with the CUDA_2D.

Finally we will present the comparison between the CPU, CUDA_Shared,

Table 2: Time (s) for CUDA_Aligned and CUDA_Shared.

Hardware	Implementation	Width				
		508	511	512	513	516
OP_8600GT	CUDA_2D	6,413s	6,428s	6,438s	6,626s	6,648s
OP_8600GT	CUDA_Shared	1,782s	1,782s	1,782s	2,178s	2,178s
AT_8800GT	CUDA_2D	2,932s	2,945s	2,949s	2,810s	2,818s
AT_8800GT	CUDA_Shared	0,630s	0,630s	0,630s	0,666s	0,666s
PH_280	CUDA_2D	0,421s	0,421s	0,421s	0,476s	0,478s
PH_280	CUDA_Shared	0,301s	0,301s	0,301s	0,305s	0,305s

CUDA_Texture, OGL_Luminance and OGL_RGBA. The CUDA_Simple, CUDA_Aligned and CUDA_2D will not be included in the comparison since, as we already seen, they do not have good performance compared with the CUDA_Shared.

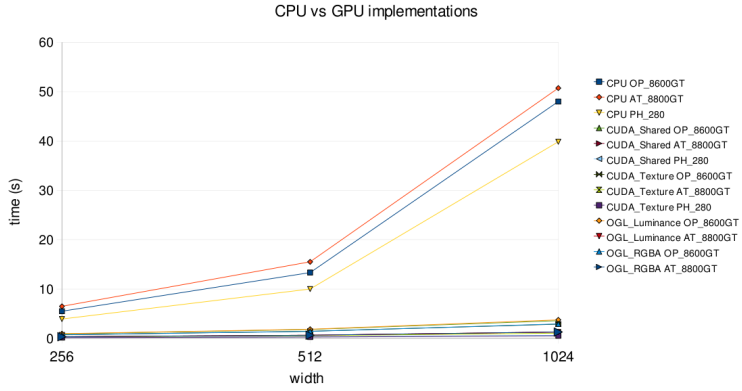


Figure 21: Comparison between CPU and GPU implementations.

The graph showed in Fig. 21 indicates the superiority of the GPU performance in comparison with the CPU implementations. Fig 22 presents the same timings of Fig. 21 without the CPU timings. And we can see that the best performance between the GPU implementations is achieved by the CUDA_Texture implementation.

Table 3 shows the timings for CPU and GPU implementations and the respective speedup of the GPU to the CPU, for each hardware environment. The problem size for this comparison is 1024x1024, i.e., height and width are 1024.

Table 3: Speedup of GPU to CPU implementations with width of 1024 elements.

	OP_8600GT		AT_8800GT		PH_280	
	Time	Speedup	Time	Speedup	Time	Speedup
CUDA_Shared	3,563s	13,47	1,259s	40,29	0,599s	66,51
CUDA_Texture	2,868s	16,73	1,042s	48,68	0,506s	78,74
OGL_Luminance	3,737s	12,84	1,277s	39,72	-	-
OGL_RGBA	2,902s	16,54	1,274s	39,81	-	-

Table 4 shows the GFLOPS achieved for our CPU implementation compared with the theoretical peak performance of the processor.

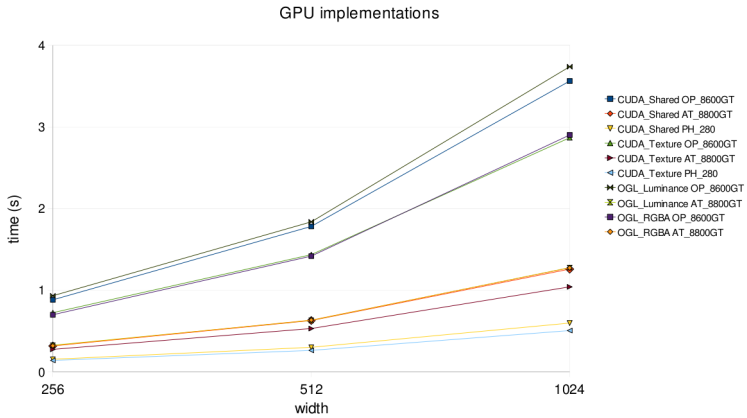


Figure 22: Comparison between GPU implementations.

Table 4: Comparison between theoretical peak performance in GFLOPS and performance achieved for the CPU implementation.

	CPU Peak	CPU GFLOPS	%/P.GF.
OP_8600GT	2,60	0,81	31,10
AT_8800GT	2,60	0,76	29,42
PH_280	12,50	0,97	7,79

Table 4 shows the GFLOPS achieved for our GPU implementations compared with the theoretical peak performance of the graphics cards.

Table 5: Comparison between theoretical peak performance in GFLOPS and performance achieved for the GPU implementation.

	GPU	CUDA_Shared		OGL_Luminance		OGL_RGBA		CUDA_Texture	
	Peak	GF.	%/P.GF.	GF.	%/P.GF.	GF.	%/P.GF.	GF.	%/P.GF.
OP_8600GT	113,28	8,83	7,79	10,10	8,92	13,01	11,48	12,06	10,65
AT_8800GT	336,00	24,99	7,44	29,55	8,79	29,64	8,82	33,22	9,89
PH_280	933,12	52,52	5,63	-	-	-	-	68,40	7,33

Table 6 shows the memory transfer rate achieved for our CPU implementation compared with the theoretical bandwidth of the processors.

Table 6: Comparison between theoretical memory bandwidth in GB/s and transfer rates achieved for the CPU implementation.

	CPU Peak	CPU GB/s	%/P.GB/s
OP_8600GT	21,2	1,63	7,68
AT_8800GT	12,8	1,54	12,03
PH_280	17,1	1,96	11,47

Table 7 shows the memory transfer rate achieved for our GPU implementations compared with the theoretical bandwidth of GPUs.

Table 7 shows some transfer rates higher than the theoretical bandwidth which can be explained by the fact that the theoretical bandwidth is based on

Table 7: Comparison between theoretical memory bandwidth performance in GB/s and transfer rates achieved for the GPU implementation.

	GPU	CUDA_Shared		OGL_Luminance		OGL_RGBA		CUDA_Texture	
	Peak	GB/s	% GB/s	GB/s	% GB/s	GB/s	% GB/s	GB/s	% GB/s
OP_8600GT	22,4	21,93	97,9	20,90	93,32	26,92	120,18	27,24	121,60
AT_8800GT	57,6	62,07	107,75	61,16	106,18	61,34	106,49	74,99	130,19
PH_280	141,7	130,43	92,05	-	-	-	-	154,44	108,99

the memory transfer from the device memory. But the shared memory and the texture cache are memories inside the multiprocessor being much faster than the device memory. Because of the 9 point stencil, for each iteration of the Jacobi solver several threads share data that are already in these fast memory areas, significantly reducing the access time for these data.

Table 8 shows the transfer rates based only on transfers from the device memory, excluding the accesses from texture cache and shared memory.

Table 8: Comparison between theoretical memory bandwidth performance in GB/s and transfer rates achieved for the GPU implementation only considering the device memory accesses.

	GPU	CUDA_Shared		OGL_Luminance		OGL_RGBA		CUDA_Texture	
	Peak	GB/s	% GB/s	GB/s	% GB/s	GB/s	% GB/s	GB/s	% GB/s
OP_8600GT	22,4	15,35	68,53	12,54	55,99	16,15	72,11	16,34	72,96
AT_8800GT	57,6	43,45	75,43	36,69	63,71	36,80	63,89	44,99	78,12
PH_280	141,7	91,30	64,44	-	-	-	-	92,66	65,40

8 Conclusion

In this paper we presented a comparison between two GPGPU programming approaches using as test case a weighted Jacobi iterative solver for the bidomain equations. Additionally a CPU implementation was provided. The OpenGL approach is the most difficult since it requires a previous knowledge of computer graphics to understand how to implement a simple GPGPU program. But, once you have your own library for GPGPU using OpenGL, it is not a hard task anymore. Nevertheless, CUDA brings it all ready and you do not need to know about computer graphics. CUDA also provides additional memory accesses patterns. Although is not so simple to understand how to improve the coalesced memory accesses, the textures can be used almost in the same fashion as on the OpenGL with only a few restrictions. Another advantage of using CUDA is that the code can be easier to read and support. One drawback of CUDA is that it is only available for NVIDIA graphics cards but on the other hand getting a code to be portable using OpenGL may not be an easy task.

Sec. 7 showed the differences between the CUDA implementations and the dependency of the hardware for both CUDA and OpenGL. The OpenGL code using RGBA textures had almost the same performance of the CUDA code using textures on the least powerful graphics card. But for the GeForce 8800GT the CUDA code was clearly faster than our OpenGL code. The CUDA code also had an impressive speedup of 78 over the CPU implementation on the AMD Phenom 9950 Quad-Core Processor with the GTX280 graphics card, which shows how powerful the graphics cards can be for solving general purpose problems. Of course, we cannot always have a speedup like this, it will depend on the level

of parallelization that is possible to achieve on a particular problem. The GPU implementations are limited by the memory bandwidth, since it dominates the computing times.

The 9 diagonals storage proposed also covers the unsymmetric case. Some code optimizations can still be performed taking advantage of the symmetry of our problem. The diagonals can also be stored in the shared memory for the CUDA_Shared version but, for this case, we need to change the block dimension for 2D blocks which can lead to additional conditionals for loading the data on the boundaries of the blocks. But a simple change of our CUDA_Shared implementation storing only the diagonal next to the main diagonal on shared memory showed a performance improvement of 7% on our test case. For OpenGL implementations the use of the symmetry of the matrix leads to a performance penalty of 10% since it is necessary the use of conditionals for the correct indexing.

Although CUDA programming is much easier to learn and apply to solve problems, it is also much easier to write inefficient codes. More effort has to be applied to extract the performance of the hardware when using CUDA since you have more flexibility it can lead you to write an inefficient code. But after understanding the different kinds of memory access and what are the advantages and disadvantages of each one, you can easily write efficient codes if your problem allows. And a final advantage of CUDA over OpenGL is the support of double precision for the new GTX 200 series NVIDIA graphics cards.

Future hardware platforms will consist of many core architectures similar to GPGPUs, Intel's Larabee. Therefore algorithm development for many core architectures are mandatory an NVIDIA + CUDA provide an ideal test system for that.

References

[1] A. L. HODGKIN AND A. F. HUXLEY, *A quantitative description of membrane current and its application to conduction and excitation in nerve.*, J Physiol, 117 (1952), pp. 500–544.

[2] NVIDIA, *CUDA Compute Unified Device Architecture*, 2.0 ed., 7 2008. Programming Guide.

[3] R. J. ROST, *OpenGL(R) Shading Language (2nd Edition)*, Addison-Wesley Professional, 2005.

[4] C. XAVIER, R. SACHETTO, V. VIEIRA, R. WEBER DOS SANTOS, AND W. MEIRA, *Multi-level parallelism in the computational modeling of the heart*, Computer Architecture and High Performance Computing, (2007).