Bachelor-thesis: GPU-Acceleration of Linear Algebra using OpenCL

Andreas Falkenstrøm Mieritz s093065

September 13, 2012

Supervisors:
Allan Ensig-Peter Karup
Bernd Dammann

IMM-B.Sc.-2012-30

Contents

1 Problem **4**

2 Abstract **5**

3 Parallel programming **6**
3.1 OpenCL . . . 6

4 API **10**
4.1 The c/c++ API: . . . 10
4.2 MATLAB interface: . . . 11
4.3 Components: . . . 14
4.3.1 Uploading . . . 14
4.3.2 Matrix Vector . . . 15
4.3.3 Norms . . . 18
4.3.4 Vector addition . . . 19
4.3.5 Vector times constant . . . 20
4.3.6 Vector minus vector constant . . . 21
4.4 Specialized components . . . 23
4.4.1 Jacobi and RBGS . . . 23

5 The Multigrid Poisson Solver **24**
5.1 The Iterative Solver . . . 24
5.2 Testing the solver . . . 31
5.3 Examining convergence . . . 42

6 Wave problem **48**

7 Conclusion **50**

8 Further studies **50**

9 References **52**

10 Appendix **53**
10.1 OpenCL kernels . . . 53
10.1.1 Sparse matrix times vector kernel . . . 53
10.1.2 Band matrix times vector kernel . . . 54
10.1.3 Upper diagonal matrix times vector kernel . . . 56
10.1.4 Vector times constant kernel . . . 58
10.1.5 Vector plus vector kernel . . . 58

10.1.6 Vector minus vector kernel 59
10.1.7 Vector minus vector times constant kernel 60
10.1.8 Vector sum kernel 61
10.1.9 Vector 2-norm kernel 65
10.1.10 Vector ∞-norm kernel 68
10.1.11 Jacobi method kernel 71
10.1.12 RBGS - Red kernel 72
10.1.13 RBGS - Black kernel 73
10.1.14 Poisson defect kernel 74
10.1.15 Interpolate kernel 75
10.1.16 Restriction kernel 78
10.1.17 KernelHeaders 79
10.2 Mex Bindings 80
10.2.1 MexInitOpenCL.cpp 80
10.2.2 MexReleaseOpenCL.cpp 80
10.2.3 MexSetGPU.cpp 81
10.2.4 MexPrintGPU.cpp 81
10.2.5 MexHandleMatrix.cpp 82
10.2.6 MexMatrix.cpp 83
10.2.7 MexReleaseMatrix.cpp 85
10.2.8 MexBandMatrix.cpp 85
10.2.9 MexReleaseBandMatrix.cpp 87
10.2.10 MexSparseMatrix.cpp 88
10.2.11 MexReleaseSparseMatrix.cpp 90
10.2.12 MexBandMatrixVector.cpp 90
10.2.13 MexSparseMatrixVector.cpp 93
10.2.14 MexSparseMatrixVector.cpp 99
10.2.15 MexVectorMinusVector.cpp 106
10.2.16 MexVectorMinusVectorConstant.cpp 109
10.2.17 MexVectorTimesConstant.cpp 112
10.2.18 MexSum.cpp 115
10.2.19 MexNorm2.cpp 116
10.2.20 MexNormInf.cpp 118
10.2.21 MexCoarseToFine.cpp 120
10.2.22 MexFineToCoarse.cpp 121
10.2.23 MexRBGS.cpp 123
10.2.24 MexJacobi.cpp 127
10.2.25 MexPoissonDefect.cpp 129
10.3 API code 132
10.3.1 OpenCLManager.h 132
10.3.2 OpenCLManager.cpp 140

10.3.3 OpenCLHigh.h 178
10.3.4 MemoryControl.h 184
10.3.5 MathVector.h 185
10.3.6 Matrix.h 188
10.3.7 BandMatrix.h 190
10.3.8 SparseMatrix.h 192
10.3.9 preprocessor.h 196

1 Problem

The focus of this report will be to create an API of linear algebra methods in OpenCL. These will then be used for the creation of iterative methods. The advantage of making a linear algebra API instead of a series of specialized methods, is that most iterative methods can be created using linear algebra. As such, a single API could form the basis of many implementations. OpenCL was chosen as it's a very open API, supported by both Nvidia and AMD, which makes it possible to measure GPU's from Nvidia and AMD against each other. GPU's are interesting in that performance-wise they're better at calculations than CPU's, the downside is that the problem given must be highly paralellizable.

For ease of use and more rapid protyping, focus will be on a MATLAB binding of this API. As such, it'll be a goal to beat MATLAB's speed with the same components. Another important goal here is the ease at which MATLAB code can be transfered to use this API, instead of MATLAB's linear algebra. MATLAB was chosen as it's designed for matrix-vector manipulations, and as such, would give a good point of reference. It is crucial that matrices can be easily transfered from and to the GPU from MATLAB, such that MATLAB's built-in routines can be used for debugging.

The poisson problem will be used to demonstrate the usefulnes of the API. The poisson problem is chosen as it's a very known problem, with many known solvers. It is therefore a good point of reference to test the API.

2 Abstract

In this report we've created a linear algebra API using OpenCL, for use with MATLAB. We've demonstrated that the individual linear algebra components can be faster when using the GPU as compared to the CPU. We found that the API is heavily memory bound, but still faster than MATLAB in our testcase. The API components were autotuned to obtain higher performance, though the components were still bound by memory transfer rates. MEX was used for the bindings from the API to MATLAB. The API was since used for modelling the poisson problem, and was able to solve the problem. We saw in this problem that we could create more specialized components for solving the problem, and obtain faster solving times. The linear algebra components excelled at rapid prototyping, being almost as easy to write as MATLAB code, and MATLAB code could be mostly replaced line by line with code from the API. The experiences from the poisson problem was taken on to the wave equation in 2d, and we observed the same trends.

3 Parallel programming

Parallel programming is, as the name suggests, a programming paradigm in which the code is run in parallel. It's a contrast to sequential programming, where the tasks are performed one at a time, instead of simulitaniously. Parallel programming has existed for a long time, but haven't been feasible for consumer grade computers until around 2005 when Intel introduced their first dual-core CPU's[1]. Since most computers nowadays have multiple cores, many programs have been written to take advantage of these when possible. GPU's [1] are designed to be fully parallel, consisting of many weaker cores. None the less, in pure terms of GLOPS, the GPU's have shown to be superior, this is evident in the current trend for supercomputers taking advantage of GPU's rather than CPU's[1]. As such, using GPU's for general purpose calculations have become a field of interest. A reason for this, can be found in Amdahl's law[2], which says that any algorithm for which a percentage of it, σ can be turned parallel, can obtain a maximum speed increase of $\frac{1}{1-\sigma}$. This means that any highly parallelisable algorithm should be able to obtain a high performance boost as long as we don't hit any limits. Limits could be the number of processing units or memory transfer rate. If we're limited by the number of processing units, Amdahl's law can be restated as $\frac{1}{(1-\sigma)+\frac{P}{N}}$, where N is the number of processing units. We also have Gustafsons law[2] which says that under the assumption that the problem is large enough to keep all P processors busy, then the speed-up of a parallel implementation compared to a sequential implementaion is given by $P - \alpha(P-1)$. Here α is the non-parallelizable fraction of the work. These laws demonstrate advantage of parallel computation.

3.1 OpenCL

Open Computing Language or OpenCL for short, is an API created for several programming languages. It enables programs to run parallel code, eighter on the CPU, GPU or an acceleration processor. Version 1.0 was introduced in 2008[3], and version 1.2 was announced on November 15th 2011. As such, it's still a relatively new technology, still under a lot of development. OpenCL is often compared to CUDA[2]. CUDA was released in 2007 and has a more extensive API than OpenCL[4]. Programming wise it's mostly possible to convert code directly between the API's, as they're very similar, CUDA code however seem to be slightly more efficient, where comparisons are possible.

[1] Graphical Processing Unit

[2] A rival API designed by Nvidia to work only on Nvidia GPU's.

This can most likely be tributed to the fact that CUDA is minded towards Nvidia GPU's only, and as such, can make architecture specific performance improvements. None the less, it's been shown that many tricks to optimize CUDA code can be transfered to OpenCL.

In OpenCL we have to specify where to run our code. For that to work a few concepts are introduced:

- **Host: The host is the is the computer running the main OpenCL program. On a standard computer this would be the CPU.**

- **Platform: A platform is an implementation of the OpenCL standard by a given vendor. For this project, the interesting vendors are Nvidia and AMD.**

- **Device: A physical device capable of running OpenCL kernel code. Typically this will be a graphics card, but dual-chip graphics cards will show up as 2 different devices.**

Next we'll make the program ready to send code to a device. This is done by setting up the following:

- **Context: A context is a group of devices, made to execute similar code.**

- **Program: A program is a representation of all kernels compiled for a specific context.**

- **Command queue: A command queue is, as the name implies, a queue of commands to be executed.**

In this report we'll not go into depth with these three concepts, as the code will be made for a single device only.

For the parallel computations of OpenCL we introduce:

- **Kernel: A kernel represents a program to be executed on a device.**

- **Work-item: A work-item represents a single run of a OpenCL kernel. An important concept here is that all work-items can be uniquely identified, and as such, the kernel code can be made to perform slight variations based on id.**

- **Work-group: A workgroup is a collection of work-items. Each workgroup can be uniquely identified.**

With this, we can make parallel code. There's however one more important concept - memory. The memory model of OpenCL allows more freedom than most programming languages. That is, it allows you to declare variables on the different caches of a device. We denote the memory spots with

- **Host: Host memory is the global memory used on the host device. For standard x86 compatible computers this would be the RAM. OpenCL uses this as a point of reference when setting the global memory of the device.**

- **Global: Global memory is memory accessable by all work-items on a opencl device, and can therefore be seen as device's equivalent to the host's RAM.**

- **Constant: Constant memory is a sub-part of the global memory, which can be used for constant values.**

- **Local: Local memory is the memory shared by a work-group on a device. The advantage to local memory over global memory is the speed at which is can be accessed. Read/writes to local memory is far faster than to global.**

- **Private: Private memory is the memory private to a work-item. It is usually quite small, but read/writes from it are even faster than those of local memory. As such, any task which uses the same value often, will obtain better performance if that value is in the private memory.**

As described, there's a lot to gain by placing variables in the correct memory location. This is crucial to speed up kernel execution speed, and should be used if possble.

The OpenCL API consists of a series of functions to obtain information on any object used by OpenCL. This means both physical objects represented by devices, and programming objects like the command queue or kernel. These can be useful in determining what version of OpenCL a certain device supports, or whether or not it supports doubles. They can also be used to obtain the time it requires to run a kernel.

A problem that occurs when writing software in OpenCL is the different architectures. In this report the focus will be on high-performance GPU programming, and as such, the most relevant players in that market are Nvidia and AMD. Though both vendors makes graphics cards, there's some differences as to how these work. While Nvidia's GPU's have mainly been optimized towards a scalar execution model, AMD has focused on a vector execution model. That is, Nvidia GPU's doesn't really distinguish between scalars and vectors when operating, they're calculated at the same speed. AMD however can perform vector operations at the same speed as scalar operations, meaning that all scalar operations are essentially limiting the GPU.

The reason for AMD's choice is most likely to be found in the fact that the main purpose of GPU's been to process graphics. Graphics fit nicely into these vectors, eighter as RGB colors, XYZ cordinates or similar. In theory, since vectors are a collection of scalars, all code optimized for AMD GPU's should be optimized for Nvidia's aswell, whereas that's not the case the other way around.

OpenCL offers ways to vectorize your program. That is, all standard variable types has vector forms. Examples of these would be float4, float8, float16. Using these types should optimize performance on AMD GPU's.

With the release of the new AMD 7000 series of GPU's, there has been a change in architecture. The AMD 7000 GPU's appear to be scalar based aswell, making them easier to utilise for non-graphics purposes. This leads us directly to the reasoning for parallel programming on GPU's.

In this report focus will be on the best performing consumer grade GPU's of the last generation, as we don't have access to the newest generation. Specifically this means the AMD 6990 card and the Nvidia 590. Both of these cards are so called dual-gpu's, that is, they're essentially two gpu's on one card. The 590 based on the architecture, should be similar to having two 580 GPU's.

The setup used will be two near-identical computers. Both of these consists of an Intel E5620 2.4ghz processor and 12GB of ram. The computers use Ubuntu server edition 10.04.4. One of the computers has two Nvidia 690 graphics cards, and the other has two AMD 6990 graphics cards. The specs of these are given below:

GPU	RAM	memory bandwidth	processors)
6990	4GB GDDR5	320GB/s	3072
590	3GB GDDR5	327.7GB/s	1024

Table 1: Specifications for the used GPU's

We're interested mostly in the bandwidth, as the linear algebra methods will be very memory-intensive, and not very computationally intensive. It's also worth nothing that we'll be using only one GPU in the API. The numbers represented above are for the card, not the GPU's on it. Since we have two GPU's sharing the same card, we might have deviations from these numbers.

The specifications have been taken from AMD and Nvidias official sites[5][6]

4 API

An API[3] have been created so that OpenCL can be used on different problems, using general components. It's designed to be quite light-weight, that is, it shouldn't take up many resources, while still being quite versatile. The advantage of having an API is that we can drastically reduce the amount of code neccersary to solve a problem. Another reason is more practical, if we know that each component in the API works as expected, then any error in an implementation must be in the code using the API instead. This makes error-finding much easier. There are two parts of the API. The first part is the c/c++ part, which is where the actual OpenCL programming is done. The second part is a MATLAB wrapper using the MEX[4] interface. The MATLAB interface is made so that we can take advantage of MATLAB's tools. Specifically, MATLAB has many advantages when it comes to analysing matrices and plotting results. These parts are not time-critical and as such, we might as well take advantage of MATLAB here. There exists other libraries designed for linear algebra, which would seem to make this small library redundant. That's however not the case, as these libraries have a tendency to be quite extensive, and make use of shared libraries. Examples here include BLAS[7] and ArrayFire[8]. BLAS is more general and can use both CPU's and GPU's, whereas ArrayFire is designed for use with GPU's only. These libraries were not designed to link against MATLAB, and as a result, this library was created for that very purpose. There's also a feature in MATLAB worth mentioning: GPUArray. GPUArray is a matlab representation of a matrix on the GPU using CUDA. There's however some restrictions to this, primarily, there's no support for anything but dense matrices. As such, it's not designed for linear algebra, but instead designed for specific routines like fourier-transforms. The first part the API that'll be discussed is the c/c++ part:

4.1 The c/c++ API:

For the c/c++ API a monolithic structure was choosen. Reason for this was two-fold. First of all, with a monolithic structure there could be a single class in charge of controlling all OpenCL calls. This way all kernel calling code could be hidden way elegantly and without sacrifising efficiency. For

[3] Application Programming Interface
[4] MATLAB Executable

the actual OpenCL code, the c bindings were choosen as opposed to the c++ bindings. The reasonning is that many of the benefits of c++ aren't present in the c++ bindings. These include programming templates and function overloads, both of which are used in the non-kernel code. There's also another reason to avoid the c++ bindings, and that is interfaces. The OpenCL c++ code uses the STL libraries and the STL libraries use of managed memory doesn't mix well with MATLAB. The reason is we require many of the variables to be persistent, and as such, we need them to be handled by MATLAB's memory management. This is not possible with standard c++ as MATLAB employs only standard c allocating and deallocating.

To sum up, the API is handled by this monolithic class singleton. Then to interact with that a series of other classes are introduces, to represent data:

- Sparse matrix, CSR formatted.
- Vector
- Band matrix
- Full Matrix

And wrapper functions are implemented for the different possible components using these. As mentioned before there'll be a lot of focus on the MATLAB interaction in this thesis. The reason for the coupling between MATLAB and c/c++ is to use the best of both worlds; the speed of c, and the testing capabilities of MATLAB. The MATLAB MEX interface is described in the following:

4.2 MATLAB interface:

As said in the last section, the MATLAB interface was created so that the examined iterative methods could be easily analyzed, while still retaining most of the efficiency of c/c++.

When dealing with MATLAB's MEX interface, the hardest part seemed to be handling of persistent memory. My solution was to make almost everything persistent, and take over MATLAB's memory management with regard to my API. This also means that I got more control over the memory management, and as such, garbage collection will not influence the iterative methods.

As the API is built around a monolithic structure, I needed a way to pass the monolithic class around between the different MEX functions. This could not be done directly, and as such, the solution was to convert the memory adress of this class to an unsigned 64 bit integer, which is then passed to MATLAB. By doing this MATLAB can pass on this value to other functions and they in turn can turn the value back into a pointer to the memory location of the monolithic class.

The same idea is employed when dealing with matrices and vectors. The MEX interface constist of many functions divided into the following categories:

- Matrix creation. These functions all return a handle to the created matrix, and takes MATLAB matrices as input.

- Matrix release. Functions for releasing matrices, to avoid wasting memory.

- Linear algebra. These functions are created for handling the linear algebra. All of them can eighter create a new matrix if needed as the returnvalue, or take a swap matrix and use it instead. It was chosen that they should be able to return matrices for easier debugging, as that secures that the matrices are independant, and that a problem cannot happen as a result of a wrong matrix being manipulated somewhere. Using swap matrices is recommended for speed as it'll be shown that uploading matrices is a slow process.

- General OpenCL parts. Initing OpenCL, finding GPU's, choosing GPU, releasing OpenCL. These parts are all necersary for the use of OpenCL with this API.

- Specialized components. These will be used to provide a point of reference for the linear algebra solution of the poisson problem.

A small example of using the API is included below in MATLAB code.

```
%get opencl handle
gpu = MexInitOpenCL();

%find GPU's:
MexPrintGPU(gpu);

%choose gpu to use: (gpu 0) and enable doubles.
MexSetGPU(gpu,0, 1)

%create matrices:
A = zeros(3,1);
```

```
B = sparse([1,2,3],[1,2,3],[1,2,3]);

%move matrices to GPU:
AHandle = MexMatrix(gpu, A);
BHandle = MexSparseMatrix(gpu, B);

%create single precision matrices:
C = single(A);

%create single precision matrices on GPU:
CHandle = MexMatrix(gpu, C);
DHandle = MexSparseMatrix(gpu, B, 1);

%perform calculations:
outDouble = Mexmatrix(gpu, A);
outSingle = MexMatrix(gpu, C);

%sparse matrix times vector for both single and double
    precision:
MexSparseMatrixVector(gpu, BHandle, AHandle, outDouble);
MexSparseMatrixVector(gpu, DHandle, CHandle, outSingle);

%obtain results for use in MATLAB:
A = MexHandleMatrix(gpu, outDouble);
C = MexHandleMatrix(gpu, outSingle);

%clean up:
MexReleaseSparseMatrix(gpu, BHandle);
MexReleaseSparseMatrix(gpu, DHandle);
MexReleaseMatrix(gpu, AHandle);
MexReleaseMatrix(gpu, CHandle);
MexReleaseMatrix(gpu, outDouble);
MexReleaseMatrix(gpu, outSingle);

%release opencl:
MexReleaseOpenCL(gpu);
```

A noteworthy thing regarding the AMD implementation of OpenCL on Ubuntu using MATLAB is that MATLAB resets the DISPLAY enviorement variable. As such, to use the API with the MEX bindings on AMD GPU's, the following command must be run first:

setenv('DISPLAY',':0');

When using OpenCL in this manner, coupled to MATLAB, there's a few overheads we need to take into consideration. First of all we need to know if there's any overhead in using the MEX interface, secondly, we need to know the time it takes to call an OpenCL kernel. If we know both of these overheads, we can then subtract them from our timings of other functions, and as such, find out how much time they use computing. We've measured this as the average of a 100 calls. We found that

	MEX	Kernel
ms	0.0127	0.006

Table 2: Overheads for API

There's also other things to take into consideration, namely tuning of our methods. There's a easy tricks to use here. First one would be loop unrolling, that is, using loops in programming costs resources, so if a loop can be avoided by repeating code, it's worth it. OpenCL provides #pragma calls for loop unrolling, making it possible to retain the better programming style of using loops, while retaining the speed of avoiding loops. Another tip is autotuning. Autotuning is a matter of finding optimal parameters for a kernel. In most cases we can divide this into the amount of work per thread (RPT) and the number of threads per workgroup (TPW). By selecting these carefully, we can obtain higher performance. The idea would be to do this beforehand for a library, and save the best values for a given GPU.

4.3 Components:

The components are the OpenCL kernels in the API. Most of them describe mathmatical functions, one exception being the uploading component. For optimal use, it's best for the components to scale linearly in time with the number of points they're used on. If this is true, we can expect that the entire problem that we use the components to solve will scale linearly in time, making it easier to scale up the problems. Furthermore, all components have been timed using the GPU's built in clock in accordance with the Nvidia OpenCL best practices guide[9].

4.3.1 Uploading

Uploading a matrix or vector from MATLAB to the GPU:

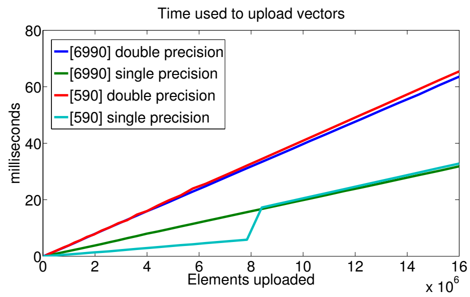

Figure 1: Test of uploading speeds

We observe a linear tendency, which makes sense, as the data is transfered continiously to the GPU memory. As we observe, transfering data to the GPU is a rather slow process compared to the other operations. As such, if we want a fast method, we want to upload as little data as possible. This however will not be a problem, as the iterative methods we'll be implementing rely on all data being on the GPU only, that is, we have no need to transfer data after the problem is uploaded. It's important here to understand that this timing is for a full matrix/vector only. In the case of a sparse matrix, we need to reorder it first, which can take a while. In that case, this graph is not representative of the time needed before said sparse matrix can be used.

We also note that the uploading speeds of the two GPU's are very similar. This is expected behaviour, as they have similar bandwidths, as shown earlier.

4.3.2 Matrix Vector

Matrix vector multiplication has been implemented in the API for all kinds of matrices. The first we'll focus on is the CSR formatted sparse matrix vector product.

Compressed sparse row (CSR) is a sparse matrix format which consists of 3 vectors. The first vector represents the non-zero values of the matrix, read first by row left to right, then by column top to bottom. The second vector represents the column index corresponding to the values. The third vector represents the indexes belonging to a row, that is, it contains an index per row corresponding to an index in the two other vectors. It is then clear that all indexes from this starting index and to the starting index of the next row will belong to the given row. The last index is the number of rows plus one, so the algorithm doesn't have to check if we're at the last row.

An example of this formatting with indexes starting from 1, would be the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 2 & 4 & 6 \\ 0 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \end{bmatrix}$$

Which would be represented by the vectors:

$$V_{val} = \begin{bmatrix} 1 & 3 & 2 & 4 & 6 & 5 \end{bmatrix}, V_{col} = \begin{bmatrix} 1 & 4 & 2 & 3 & 4 & 2 \end{bmatrix}, V_{row} = \begin{bmatrix} 1 & 3 & 6 & 6 & 7 \end{bmatrix}$$

The CSR format was chosen because of it's parallel potential. MATLAB uses a compressed column format, much similar to the CSR, which is great for a CPU, as you can load the first value of a vector to the cache and then multiply this by each value in the row, thereby keeping everything very memory efficient and fast. In contrast this cannot be done with the CSR format, but, the CSR format allows each line to be calculated on it's own, thereby enabling threads to work in parallel.

The kernel for this can be found in 10.1.1. The first thing presented is autotuning of the method on the GTX 590:

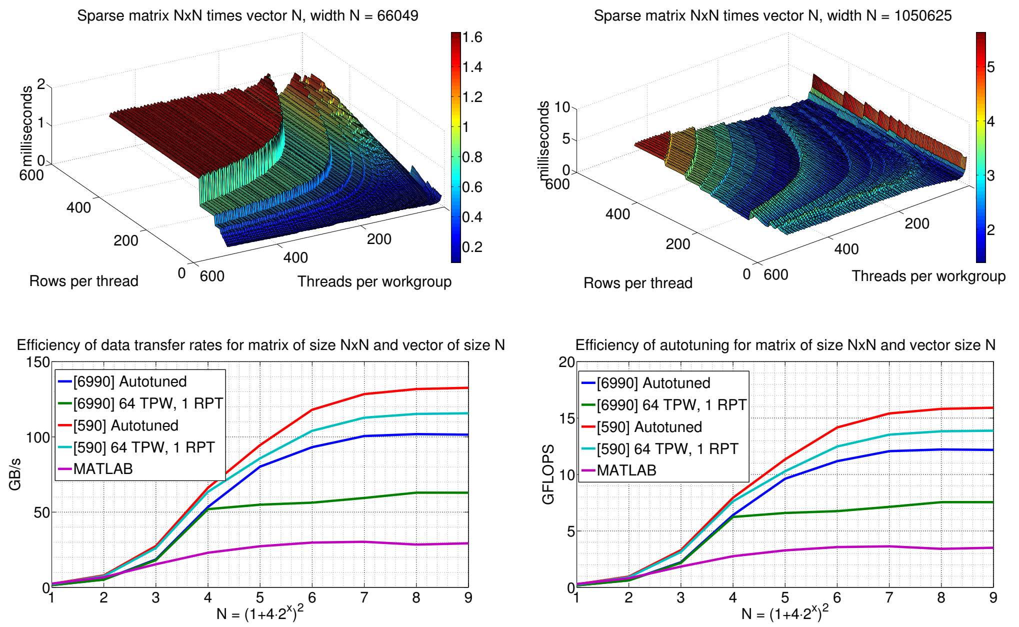

Figure 2: Demonstration of autotuning and performance calculation for a double precision sparse matrix with 6 non-zero values per row.

As demonstrated by the plots, there's a clear reason for why autotuning is important. First of all, as expected, it's about parallelity. The smaller the problem, the less rows should be used per thread. This logically ensures that as many threads as possible are working simulitaniously. As such, the number of rows per thread becomes interesting only for higher values of N.

The next matrix component is the band matrix. The band matrix is a matrix that can be represented as a diagonal with a bandwidth. A simple representation with bandwidth 1 would be

$$\begin{bmatrix} 1 & 2 & & & & & \\ 3 & 4 & 5 & & & & \\ & 7 & 8 & 9 & & & \\ & & 10 & 11 & 12 & & \\ & & & 13 & 14 & 15 & \\ & & & & 16 & 17 & 18 \\ & & & & & 19 & 20 \end{bmatrix}$$

It's clear that we want to avoid zero values where we can avoid so. That's however not entirely true, for speed, we store the matrix shown above internally as the matrix

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 11 \\ 12 & 13 & 14 \\ 15 & 16 & 17 \\ 0 & 18 & 19 \end{bmatrix}$$

As can be seen, we choose to store extra zero values. This is done for three reasons. First of all, we can speed up the kernels by not having them check how many values there are per row first. Secondly, in an actual implementation for a problem, we could perform loop unrolling, that is, we know exactly how large a band is needed, and as such, we can replace the for loop in the kernel with just the calculations, thereby speeding things even more up. Thirdly, we would gain nothing by not storing it other than a little extra space. The reason is, that a work-group in OpenCL will only be as fast as it's slowest thread, and as such, even if a few threads performed faster, we would gain nothing.

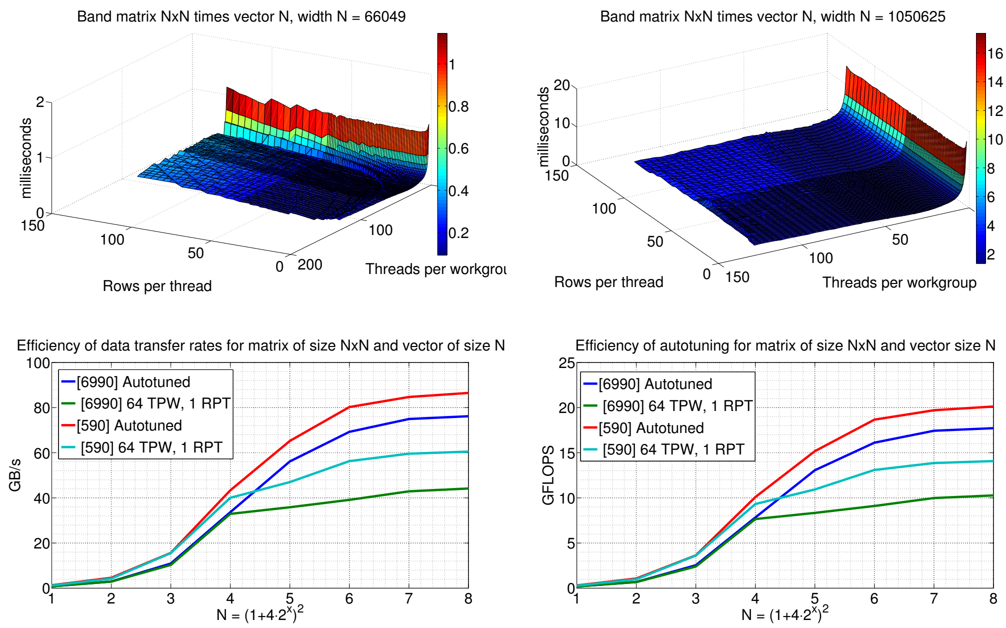

Figure 3: Demonstration of autotuning and performance calculation for a double precision band matrix with bandwidth 3.

The band matrix times vector performs better than the sparse matrix vector. This is expected, as the same amount of work requires less information to perform with the band matrix, as we know exactly where the elements are stored. This kernel is still naive though, we could use our knowledge that the rows share indexes of the vector for bandwidths higher than 1. As such, there's room for performance improvements, should that be needed. MATLAB was left out of this comparison, as MATLAB has no native type for band matrices. As such, if we were to compare, it would be against a sparse matrix solution in MATLAB, which has already been shown to be worse than our sparse matrix implementation.

4.3.3 Norms

For error measure we usually need norms. For that we've implemented the $||\cdot||_2$ and $||\cdot||_\infty$ norms, as well as the sum operator. They all have fairly similar performance, as their implementations are mostly the same. The kernels can be found in appendixes 10.1.8, 10.1.9, 10.1.10. The idea is that the workgroup sizes equals to the reduction sizes in the power of 2. That is, each thread on the GPU starts by reducing up to reduction size elements, and saving it in memory. The afterwards, the threads starts to work on the local memory representations. It can be seen as a reverse pyramid structure.

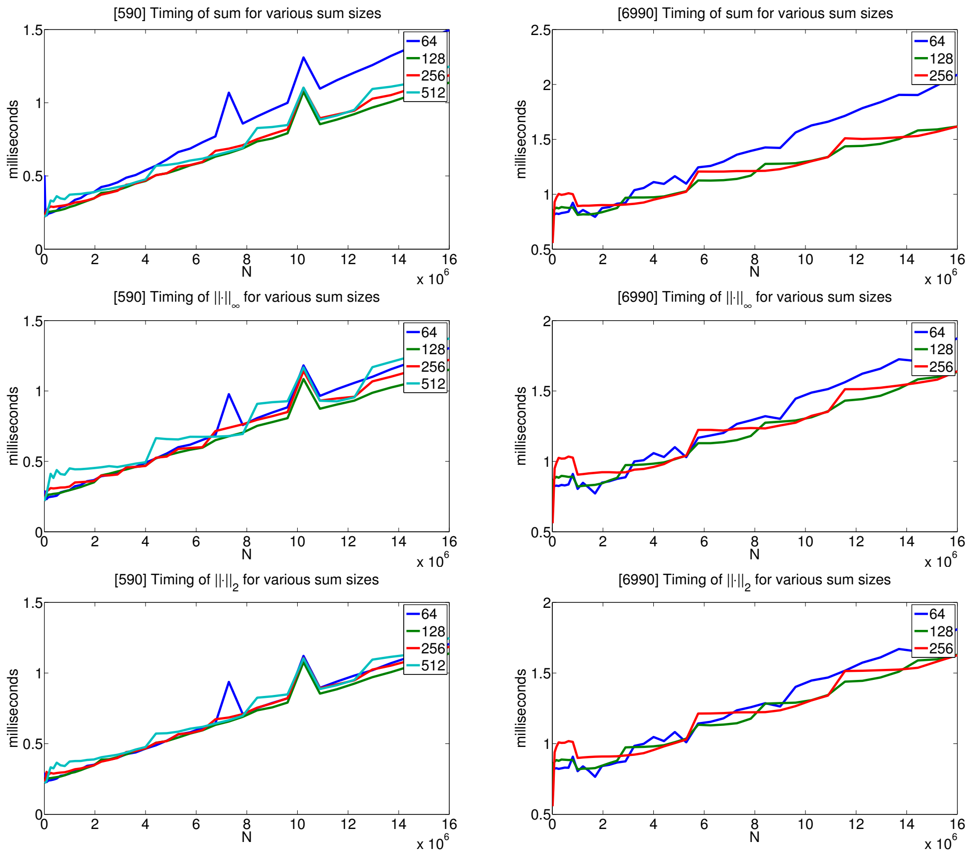

Figure 4: Norm measurements for the 590 and 6990 GPU's with N elements

As we can observe, their performance is mostly the same. This is due to very similar implementations. They all take advantage of local memory to store results. This idea was shown to be efficient[**?**]. Our implementation is slightly less efficient as it uses more checks, as it's not limited to specific sizes of vectors, none the less, it's still far more efficient than a fully naive implementation. We notice that 128 seems to be the optimal number of the reduction size for the 590 GPU. We also notice that the 6990 seems to be slower. It's also not possible to use a reduction size of 512 for the 6990 GPU.

4.3.4 Vector addition

Both vector plus vector and vector minus vector have been implemented, but since they share the same performance, I've only presented vector plus vector. While it's called vector plus vector, it can work for all data types, but, it only applies to the array of values. That is, in the sparse matrix we only store values for given coordinates, but we can still add a vector to a sparse matrix, it'll just think that the sparse matrix is a vector, and as such, the values will not be placed correctly. This can be used to change values if you're careful and know the structure of the sparse matrix well enough. The kernel can be found in 10.1.6

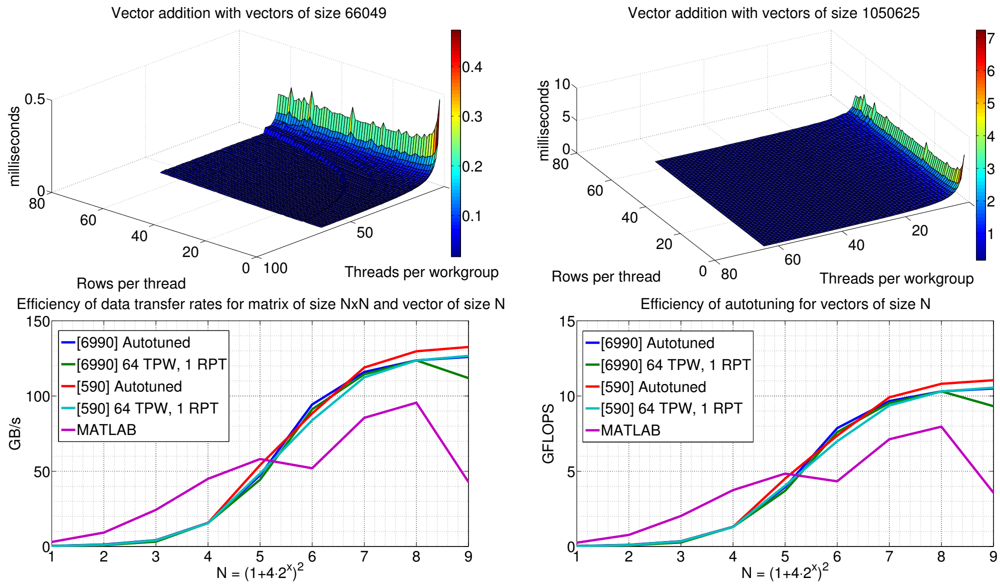

Figure 5: Demonstration of autotuning and performance calculation for a double precision band matrix with bandwidth 3.

As can be seen, the performance is not very good. This is unfortunely as expected, as we have to load two values and write one value from global memory. In contrast, we only perform one arimethic operation, as such, the amount of "number crunching" done compared to the memory load is very low. There's not much that can be done to improve this, as no indexes are required by multiple threads of eighter vector. Still, the performance is about the same as the sparse matrix vector operator, and as such, if just the method we use doesn't rely too heavily on adding vectors together, we should be able to obtain acceptable performance.

4.3.5 Vector times constant

Vector times constant have been implemented and measured as seen below. While it's called vector times constant, it works for all data types. This is a consequence of all matrix types being built as extentions of the vector type. The kernel can be found in 10.1.4

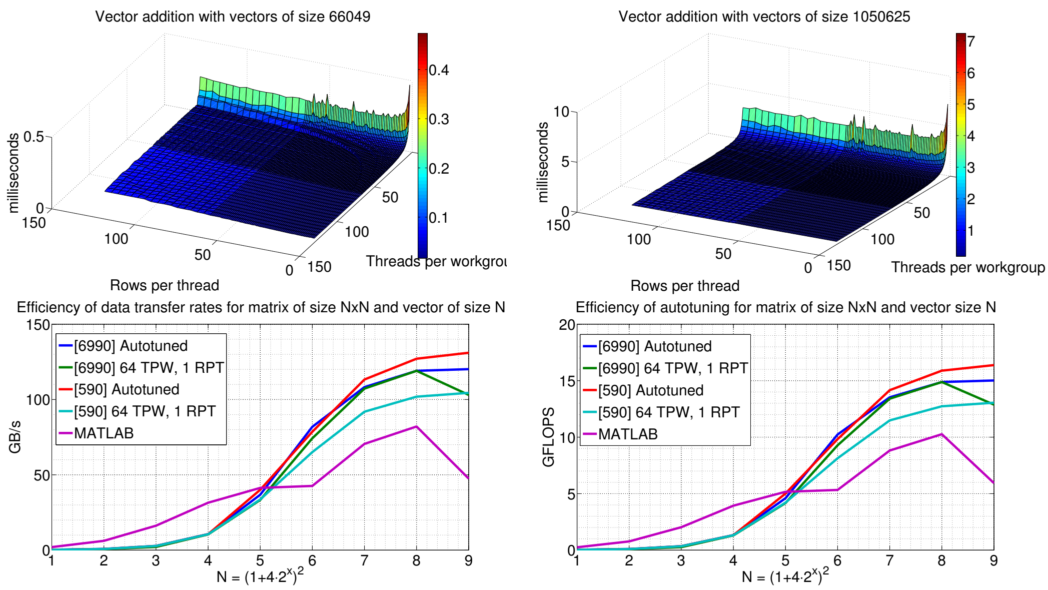

Figure 6: Demonstration of autotuning and performance calculation for a vector multiplied by a constant

Again we observe that our kernel doesn't acchieve a very high performance, and again, the blame is given to the fact that it's just one arimethic operation per 1 read and 1 write to global memory. There's not much to do to increase performance here, we can however observe that MATLAB is even slower. The last linear algebra component we'll look into is a combination of the vector times constant and vector minus vector, called vector minus vector constant.

4.3.6 Vector minus vector constant

Vector times constant have been implemented and measured as seen below. While it's called vector times constant, it works for all data types. This is a consequence of all matrix types being built as extentions of the vector type. The kernel can be found in 10.1.7

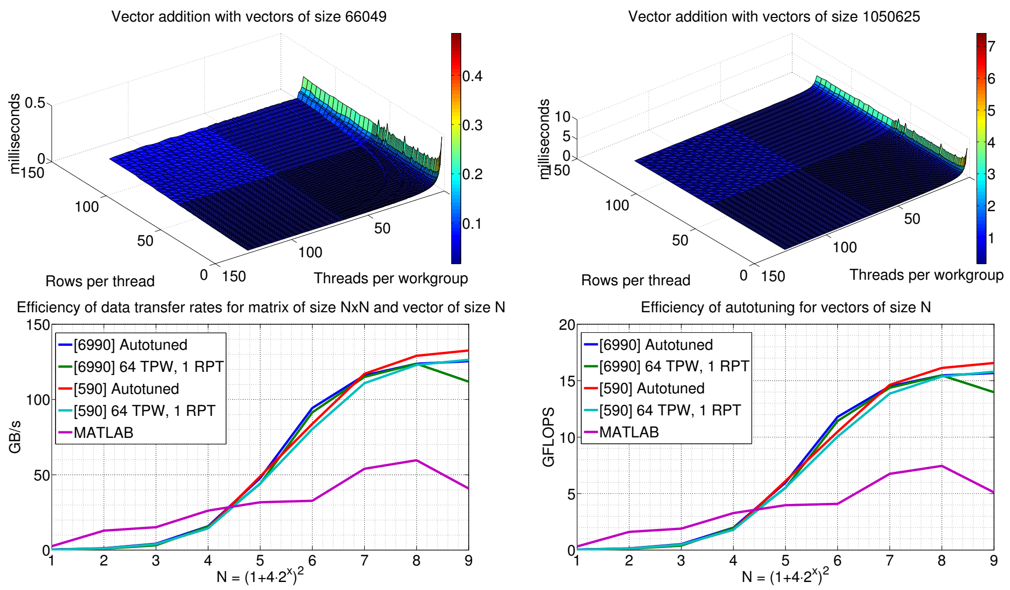

Figure 7: Demonstration of autotuning and performance calculation for a vector multiplied by a constant

As seen, the performance of this method is on par with vector minus vector and vector times constant, so the extra work we perform is essentially free.

A general tendency we've seen for all these methods, is that the Nvidia GPU 590 has come out in the lead. Especially in the matrix multiplications. This could be a consequence of the architectural differences, where the AMD 6990 is built around using vector types to handle everything, instead of single elements. We've also seen waves when autotuning, which clearly shows that there's a big difference between autotuning and not autotuning. The advantage of autotuning is that we can secure that we're at the bottom of the waves. The waves themselves are logical to explain. All threads have to perform the same number of iterations, so if we can increase the number of threads or work per thread slightly, we may end up with less iterations total. The waves represent this, as the high points are badly chosen values of work per thread and threads per workgroup. It also demonstrates that threads work best when coupled in workgroups. It's also clear that the algorithms are memory bound. It should therefore be possible to create faster routines, specialized for the job.

4.4 Specialized components

For the poisson problem a series of specialized components have been implemented, to give a point of reference of the performance of the general components. Each of these have been tested on the Nvidia 590 GPU, to provide insight into their performance.

4.4.1 Jacobi and RBGS

The Jacobi and RBGS components are implemented. Their kernels can be found in 10.1.12, 10.1.13 and 10.1.11. Each of these perform 4 additions and 3 multiplications per interior point in a grid.

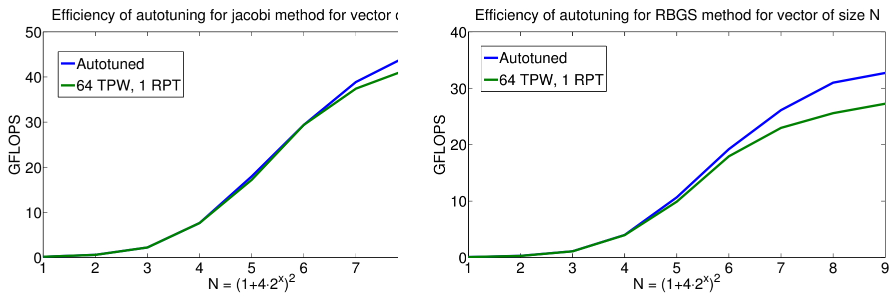

Figure 8: Autotuning results for the jacobi and RBGS method using doubles

As we can see, these kernels have a quite high amount of GFLOPS compared to the native components, even though they're not optimized codewise. I've also implemented a defect calculation operator. The kernel can be found in 10.1.14.

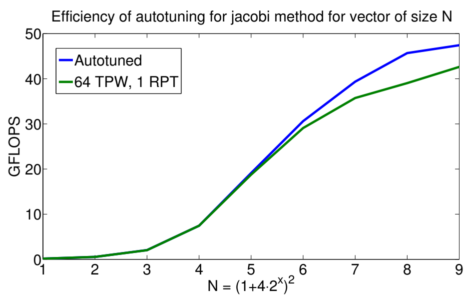

Figure 9: Autotuning of the defect calculator using doubles

Again we observe that the specialized methods perform better.

Much of this extra performance can most likely be attributed to the kernels being simplier. That is, the matrix vector kernels all contain for loops.

For an actual problem these could be removed, if we knew exactly what to replace them with. As such, while these kernels are faster, then by no means are the kernels as fast as they can be. These specalized kernels are meant only as a point of reference to see if it's worth it to implement kernels for a specific problem as opposed to a more general approach. In that case we've nearly tripled performance over the matrix vector operations.

5 The Multigrid Poisson Solver

In the following we'll examine the poisson problem

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y) \ \in \Omega([0, 1]^2) \tag{1}$$

With dirichlet boundary conditions:

$$u(x, y) = 0, \quad (x, y) \in \partial\Omega \tag{2}$$

To solve the problem numerically we'll employ a multigrid iterative solver. The idea of an iterative multigrid solver, is, as the name implies to solve the problem iteratively on multiple grids. We do this, as the iterative method shows to be more efficient in terms of error reduction on coarse grids, while being unable to obtain the accuracy that we want. Therefor, we try to reduce the errors on the coarse grids, and then reduce the remaining errors on the finer grids, thereby hopefully taking all advantages of the different grids and none of the weaknesses.

5.1 The Iterative Solver

In the following we'll introduce the components required to solve the problem (1) iteratively on a grid. All components will be presented in 3 versions. One version written in MATLAB, one version written as a hybrid of MATLAB and OpenCL, and the last one is pure OpenCL with specialized kernels. These will be measured against eachother, to see how much time is needed to solve the problem for a given level of accuracy.

First we need a solver. A solver is an operator S on the form

$$U^{[k+1]} = SU^{[k]}$$

One important property of our solver is that it converges. We describe this as $\rho(S) < 1$, where $\rho(S)$ is the convergence factor for one iteration. If it doesn't converge, then we cannot hope to solve our problem, however, for the poisson problem we know of two solvers that converge[10]. The first one is the Jacobi solver, and the second one is the red-black gauss-seidel solver, or RBGS for short.

The jacobi solver can be described by the stencil

$$\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}$$

By a stencil, we mean an equation to calculate a specific property per interior point. In this case, the stencil is used to calculate as estimate of $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$. The idea is that

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x^2} = 4U_{i,j} - U_{i,j+1} - U_{i,j-1} - U_{i+1,j} - U_{i-1,j} + O(h^2)$$

And here we see why finer grids can obtain higher accuracy. The error depends on h^2. If we substitute this into the equation (1), we obtain a discrete solver

$$4U_{i,j} - U_{i,j+1} - U_{i,j-1} - U_{i+1,j} - U_{i-1,j} + O(h^2) = f_{i,j}$$

Which leads to

$$U_{i,j}^{[k+1]} = \frac{1}{4}(U_{i,j+1}^{[k]} + U_{i,j-1}^{[k]} + U_{i+1,j}^{[k]} + U_{i-1,j}^{[k]} + f_{i,j}) \quad (3)$$

We'll also look at the Red-Black Gauss-Seidel iterative solver. It's the same as the jacobi solver, with the change that it works on specified grid points only, instead of the entire grid. Simply said, the grid is split up into red points and black points, as demonstrated by

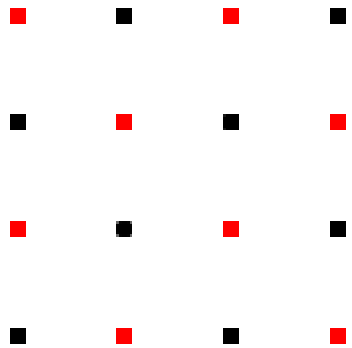

Figure 10: Red-Black ordering

The operator S^{Red-GS} works on the red points only, and $S^{Black-GS}$ works on the black points only. The entire smoothing operator is given as

$$S^{RB} = S^{Red-GS} S^{Black-GS}$$

At first it would seem to be the same as the jacobi solver, but there's one important difference. After the first operator is done, the points that the second operator will use have already been updated, as such, we expect to see faster convergence. There's however a possible drawback, which concerns the ability to run these in parallel. As long as there's a sufficient number of red and black points, we should be able to run the smoother in parallel, but for small grids, the jacobi method will probably be faster per iteration. Even on larger grids, the jacobi method should be slightly faster per iteration, as it requires only one matrix-vector product.

We observe the numerical efficiency of the methods. That is, how fast they can decrease the defect

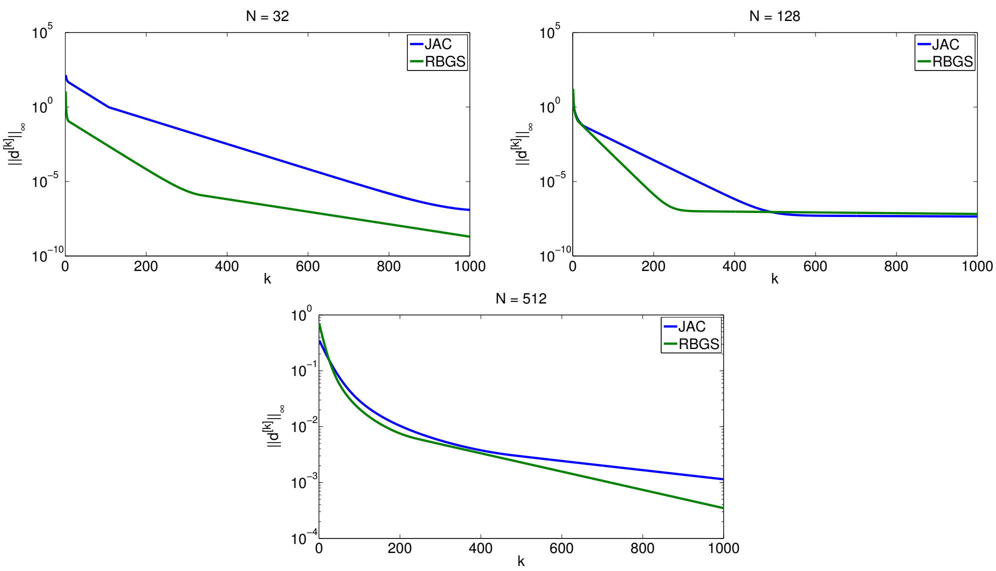

Figure 11: Jacobi and GS-RB smoothers measured against eachother.

As expected, we see that the RB-GS method converges faster. The jacobi method on the other hand should have a faster execution time. This is demonstrated in the plots:

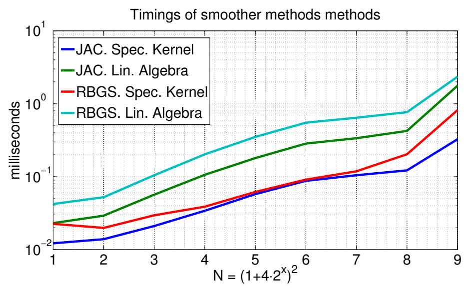

Figure 12: Execution times of the smoother methods on the AMD 6990 GPU. (double precision)

These timings have been written down in the following table. Here we have that S.K is short for specialized kernel and L.A. is short for linear algebra.

N:	5^2	9^2	17^2	33^2	65^2	129^2	257^2	513^2	1025^2
$JAC^{S.K.}$	0.0122	0.0139	0.0211	0.0342	0.0578	0.0879	0.1049	0.1221	0.3268
$JAC^{L.A.}$	0.0231	0.0293	0.0568	0.1064	0.1803	0.2849	0.3371	0.4246	1.7702
$RBGS^{S.K.}$	0.0226	0.0199	0.0296	0.0389	0.0619	0.0913	0.1187	0.2033	0.8181
$RBGS^{L.A.}$	0.0423	0.0524	0.1049	0.2029	0.3517	0.5508	0.6433	0.7669	2.3509

Table 3: Timings of the smoother methods on the AMD6990 GPU in milliseconds (double precision)

N:	5^2	9^2	17^2	33^2	65^2	129^2	257^2	513^2	1025^2
$JAC^{S.K.}$	0.0120	0.0129	0.0189	0.0306	0.0506	0.0747	0.0878	0.0969	0.2239
$JAC^{L.A.}$	0.0236	0.0285	0.0559	0.1055	0.1747	0.2746	0.3230	0.3737	1.0125
$RBGS^{S.K.}$	0.0210	0.0180	0.0270	0.0362	0.0573	0.0830	0.0992	0.1393	0.4171
$RBGS^{L.A.}$	0.0402	0.0513	0.1039	0.2022	0.3442	0.5388	0.6273	0.7010	1.7369

Table 4: Timings of the smoother methods on the AMD6990 GPU in milliseconds (single precision)

It should be clear that the RBGS method is superior on large scale problems. Another thing we realise is the very need for multigrid solvers, as seen from the iteration plots, we're doing a lot of work on the large grids, without getting an equal payoff. That is, the convergence rate isn't very good on the large grids, thereby ruining anything we could gain from solving the system on a large grid.

Furthermore, we also realise, as expected, that the linear algebra methods are inferior to more specialized methods in terms of execution speed. The linear algebra methods however are not much slower. The jacobi smoother is ≈ 6 times slower on the largest grid, while the RBGS smoother is ≈ 3 times slower. If we think about this, then it still makes much sense to utilize these linear algebra components for any kind of prototyping, and if speed is crucial, more specialized components can be used. We also see that single precision is quite a bit faster, on sufficiently large grids, but even on low grids, there's still time to save, meaning that any calculations that aren't required to be in double precision, can be calculated more efficiently in single precision.

For the multigrid solver we need a method of transfering solutions between grids. For that we introduce the restriction and interpolation operators I_{2h}^{h}, I_{h}^{2h}. They exist to transfer our guess of a solution to a coarser or finer grid. We define them by the stencils

$$I_{2h}^{h} = \frac{1}{4} \left] \begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix} \right[_{2h}^{h}$$

$$I_{2h}^{h} = \frac{1}{16}\left]\begin{matrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{matrix}\right[_{h}^{2h}$$

Since I_{2h}^{h} transfers from a coarse grid to a fine grid, its stencil is used on the fine grid. As such, it'll estimate an average of 1,2 or 4 coarse grid points, therefore we've denoted it with the reversed brackets. Both of these operators have been implemented in two ways. One is a direct specialized kernel, while the other relies on the linear algebra components, that is, they're both based on the linear algebra operations, but the specialized kernels perform the operations without the use of matrices.

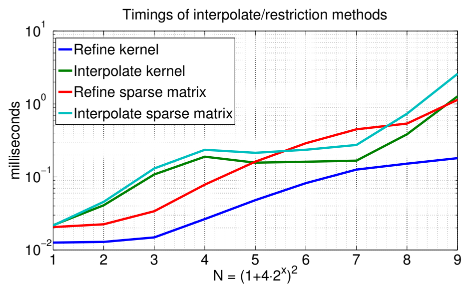

Figure 13: Execution times of the transfer methods on the AMD 6990 GPU. (double precision)

N:	5^2	9^2	17^2	33^2	65^2	129^2	257^2	513^2	1025^2
$I_h^{2h}(S.K.)$	0.0126	0.0129	0.0148	0.0264	0.0481	0.0823	0.1261	0.1521	0.1810
$I_{2h}^{h}(L.A.$	0.0218	0.0409	0.1082	0.1895	0.1570	0.1616	0.1673	0.3845	1.2734
$I_h^{2h}(S.K.)$	0.0206	0.0224	0.0339	0.0785	0.1620	0.2898	0.4499	0.5391	1.1413
$I_{2h}^{h}(L.A.)$	0.0216	0.0456	0.1307	0.2355	0.2142	0.2361	0.2745	0.7354	2.5898

Table 5: Timings of the transfer methods on the AMD6990 GPU in milliseconds (double precision)

N:	5^2	9^2	17^2	33^2	65^2	129^2	257^2	513^2	1025^2
$I_h^{2h}(S.K.)$	0.0134	0.0115	0.0104	0.0175	0.0457	0.0828	0.1231	0.1531	0.1833
$I_{2h}^{h}(L.A.)$	0.0225	0.0351	0.0668	0.1090	0.1428	0.1501	0.1527	0.3310	1.0508
$I_h^{2h}(S.K.)$	0.0225	0.0201	0.0227	0.0468	0.1467	0.2702	0.4178	0.4955	0.7916
$I_{2h}^{h}(L.A.)$	0.0239	0.0415	0.0854	0.1430	0.2071	0.2317	0.2631	0.6264	2.0229

Table 6: Timings of the transfer methods on the AMD6990 GPU in milliseconds (single precision)

We see that as before, using single precision is faster. Furthermore, as before, the specialized kernels have an edge over the linear algebra kernels. It's worth to note however that we won't need to use these transfers as often as the smoothers, thus if we're looking for more speed in our solver, it would be wise to start by increasing the speed of the smoothers. The last part we'll need for our multigrid solvers is a way to calculate the defect. Here the defect is the residuals between the left hand side and the right hand side of the discretized form of (1). We find that the defect is related to the errors of our system, that is, given the real solution at a grid point, $u_{i,j}$ we have the following relation to the error $e_{i,j}$ and our guess $U_{i,j}$ at said grid point:

$$u_{i,j} = U_{i,j} + e_{i,j}$$

From the discretized form of (1) we get the following definition of the defect:

$$D_{i,j} = 4U_{i,j} - U_{i+1,j} - U_{i-1,j} - U_{i,j+1} - U_{i,j-1} - f_{i,j}$$

And we observe that by substituting the definition of the error into this we have that

$$D_{i,j} = 4e_{i,j} - e_{i+1,j} - e_{i-1,j} - e_{i,j+1} - e_{i,j-1}$$

Which can be rewritten to form an iterative solver for the errors:

$$e_{i,j}^{[k+1]} = e_{i+1,j}^{[k]} + e_{i-1,j}^{[k]} + e_{i,j+1}^{[k]} + e_{i,j-1}^{[k]} - D_{i,j}$$

We see that this means that

$$e_h^{[k+1]} = S_h e_h^{[k]} - \frac{1}{4} D_h^{[k]}$$

We reconize this form as being one we can handle with our smoothers. The methods for calculating the defect have been implemented and are presented:

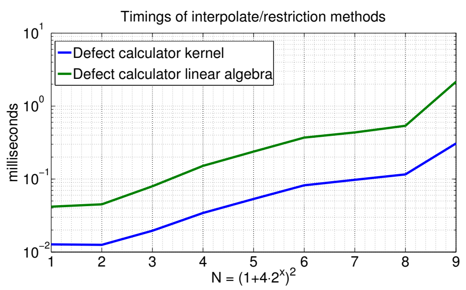

Figure 14: Execution times of defect calculation on the AMD 6990 GPU. (double precision)

N:	5^2	9^2	17^2	33^2	65^2	129^2	257^2	513^2	1025^2
$D(S.K.)$	0.0127	0.0126	0.0196	0.0343	0.0533	0.0821	0.0976	0.1158	0.3076
$D(L.A.)$	0.0418	0.0451	0.0796	0.1518	0.2379	0.3700	0.4355	0.5369	2.1559

Table 7: Timings of defect calculation on the AMD6990 GPU in milliseconds (double precision)

N:	5^2	9^2	17^2	33^2	65^2	129^2	257^2	513^2	1025^2
$D(S.K.)$	0.0125	0.0125	0.0195	0.0337	0.0523	0.0787	0.0932	0.1022	0.2384
$D(L.A.)$	0.0441	0.0443	0.0782	0.1488	0.2315	0.3583	0.4191	0.4760	1.3843

Table 8: Timings of defect calculation on the AMD6990 GPU in milliseconds (single precision)

We observe that the specialized method is faster as expected. With these components we have everything we need to create our multigrid solver.

5.2 Testing the solver

Two multi-grid solvers have been implemented. One full multigrid solver (FMG), that is, it starts with a guess on the coarsest grid and improves this guess before sending it to a finer grid.

Algorithm 1 Full Multigrid

choose initial guess on coarsest grid $u^{[0]}$.
$u^{[0]} \leftarrow MultiGrid_V(k, u^{[0]}, b^{[0]}, v1, v2)$
for $k = 2, 3, ..., k_{max}$ **do**
  $u^{[k]} \leftarrow I_{k-1}^{k} u^{[k-1]}$
  $u^{[k]} \leftarrow MultiGrid_V(k, u^{[k]}, b^{[k]}, v_1, v_2)$
end for

The multigrid solver starts with a guess on the finest grid, and improves on that.

Algorithm 2 Multigrid v cycle

Choose initial guess $u^{[0]}$.
while $n < n_{max}$ **do**
  $n \leftarrow n + 1$
  $u^{[n]} \leftarrow MultiGrid_V(k, u^{[n-1]}, b_h, v_1, v_2)$
  $d^{[n]} \leftarrow b - Au^{[n]}$
end while

After the initial step, they both use a multigrid v-cycle (MGV). The full multigrid solver should however have the advantage that the error starts at a lower level, thereby reaching a given level of accuracy in fewer iterations.

Algorithm 3 $MultiGrid_V(k, u_h, b_h, v_1, v_2, v_{cor})$

if $k = 1$ **then**
  for $i = 1, .., v_{cor}$ **do**
    $u_h \leftarrow SMOOTH(u_h, b_h)$
  end for
else
  for $i = 1, .., v_1$ **do**
    $u_h \leftarrow SMOOTH(u_h, b_h)$
  end for
  $d_h \leftarrow A_h u_h - b_h$
  $d_{2h} \leftarrow I_h^{2h} d_h$
  $e_{2h} \leftarrow MultiGrid_V(k-1, 0, d_{2h}, v_1, v_2, v_{cor})$
  $e_h \leftarrow I_{2h}^{h} e_{2h}$
  $u_h \leftarrow u_h - e_h$
  for $i = 1, .., v_2$ **do**
    $u_h \leftarrow SMOOTH(u_h, b_h)$
  end for
end if

There's another thing we need to consider. As shown by the derivation of the solver, the error depends on h^2. This means that for coarse grids, we might not need as high precision as on the fine grids. We know from the components that the computing time depends on the precision type, so it should be possible to decrease computing time without sacrifising too much precision of the final solution. This method is known as mixed precision, and has been proven efficient on other problems [11]. In the following we'll test the FMG and MGV solvers for varying numbers of pre and post smoothings. We'll denote the number of post-smoothings as v_2 and pre-smoothings as v_1. This is performed on all grids but the coarsest, where we perform v_{cor} smoothings.

We've tested both implementations with varying parameters v_1, v_2, v_{cor} and for various tolerances of $||D||_2$ and $||D||_\infty$. A selected few are presented.

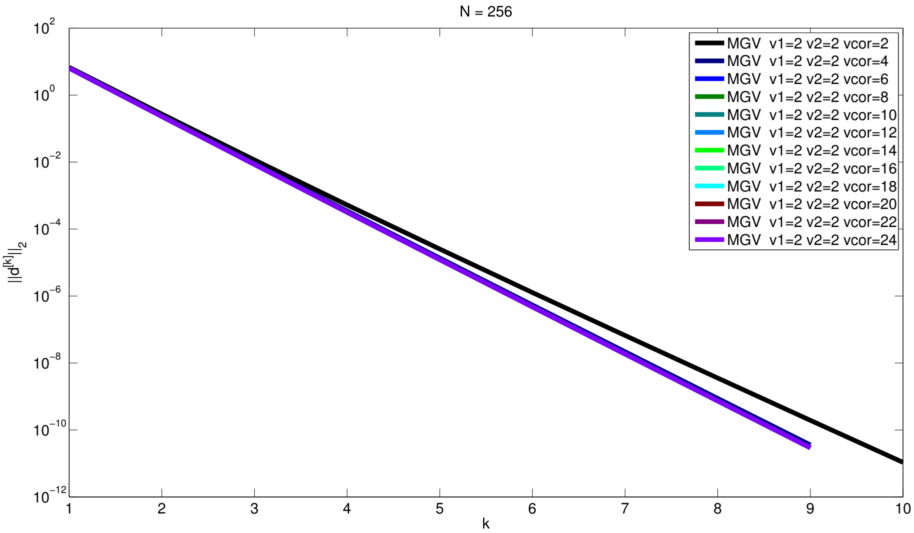

Figure 15: Testing the multigrid method with RBGS on a grid of size $[257, 257]$ with low values of v_1, v_2 and 6 subgrids

We see here that for a grid of this size, there's not much change in varying v_{cor}, we can at most save one iteration. Next we'll check if we can improve these results by increasing v_1, v_2.

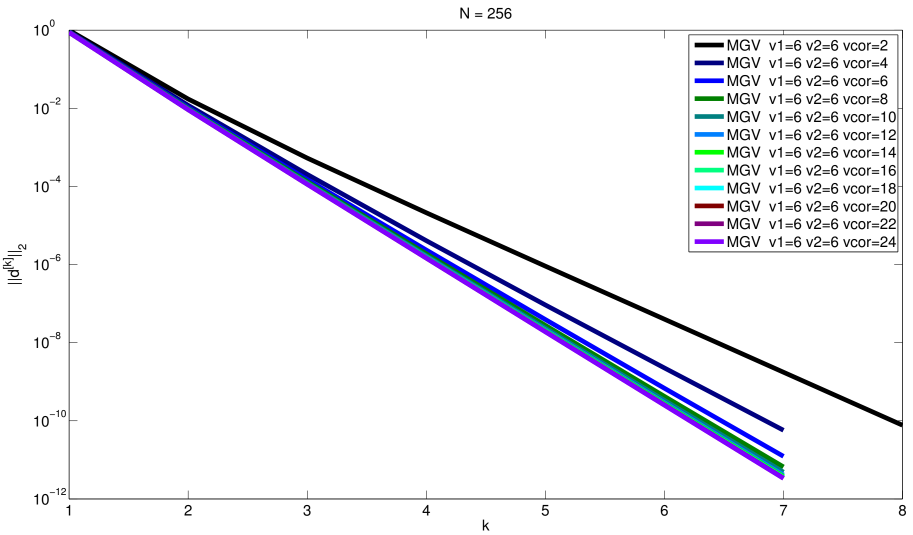

Figure 16: Testing the multigrid method with RBGS on a grid of size $[257, 257]$ with higher values of v_1, v_2 and 6 subgrids

As we can see when comparing to the previous example, there's a lot to gain in terms of iterations required by increasing v_1, v_2. This can be attributed to the fact that the grid is small enough for v_1, v_2 to cause more rapid changes. If instead we had looked at a larger grid, we should see that v_1, v_2 will have less effect:

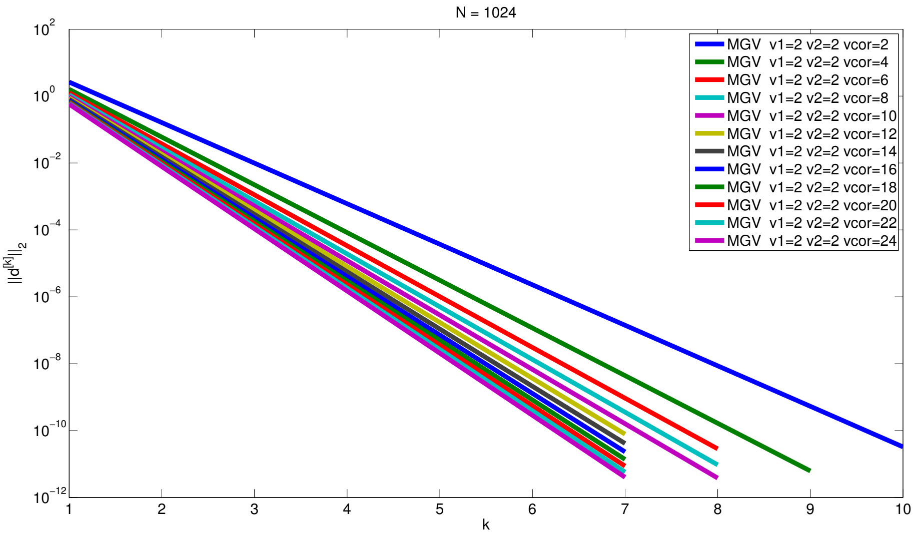

Figure 17: Testing the multigrid method with RBGS on a grid of size $[1025, 1025]$ with low values of v_1, v_2 and 8 subgrids

We see here that for a grid of this size, there's much to be gained by choosing an appropriate value of v_{cor} for a given tolerance. Here's it's important to remember that each iteration we spare represents much less work to be done, even if each subiteration takes a little more work. As before, we'll investigate the effects of v_1, v_2:

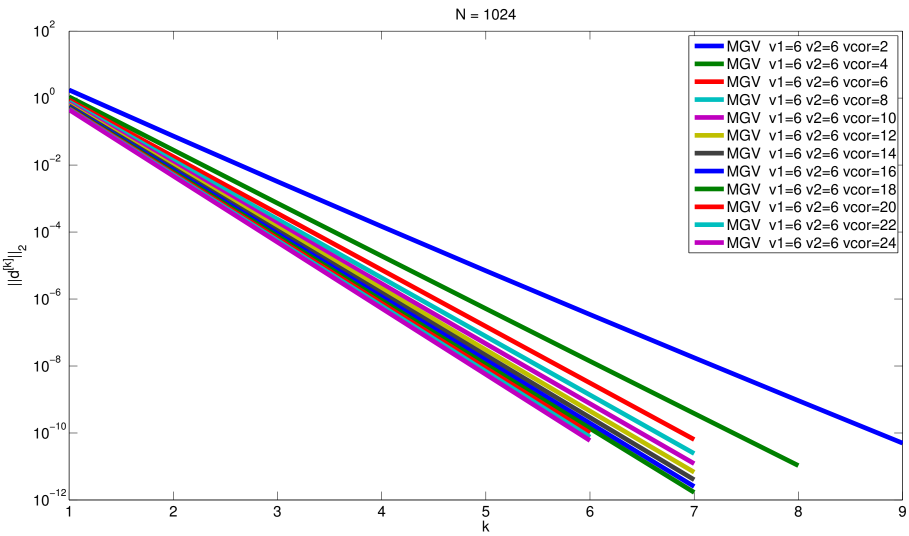

Figure 18: Testing the multigrid method with RBGS on a grid of size $[1025, 1025]$ with higher values of v_1, v_2 and 8 subgrids

We notice that the changes are minimal, as discussed before. That is, the grid is so large that the smoothing is primarily done on the smaller grids. As such, it's mostly a waste to perform large amounts of smoothings on the large grid.

One of biggest problems with the MGV approach is that start with a noisy guess. As such, there's a lot of errors which have to be corrected first on the coarser grids, and most work on the finer grids could be done more efficiently by the coarser grids. This leads us to investigate full multigrid. Full multigrid means starting with a guess on the coarsest grid, smoothe there, then interpolate to a finer grid and smoothe there and so on, until we reach the grid size we want. This should give us a much better start-guess, which we can then continue with using MGV. We examine this in the same way as for the MGV method:

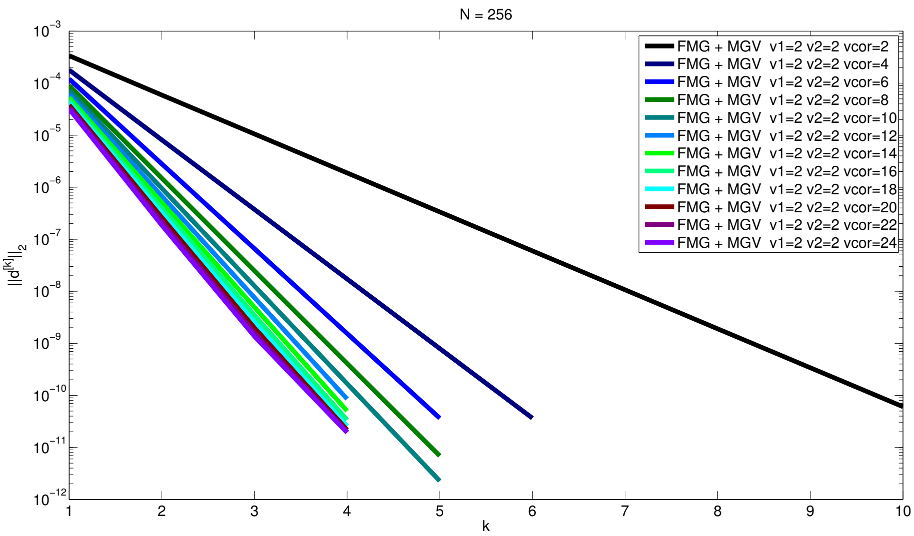

Figure 19: Testing the full multigrid method with RBGS on a grid of size $[257, 257]$ with low values of v_1, v_2 and 6 subgrids

As we can see, based purely on the number of MGV iterations needed, we've acchieved qute an improvement by using the FMG method first. Based on the definition of the FMG method, we can see that it must be more expensive than one MGV iteration. None the less, it's not by much, and as such, we can expect faster solving times by using FMG. To examine the importance of v_1, v_2 we solve for higher values:

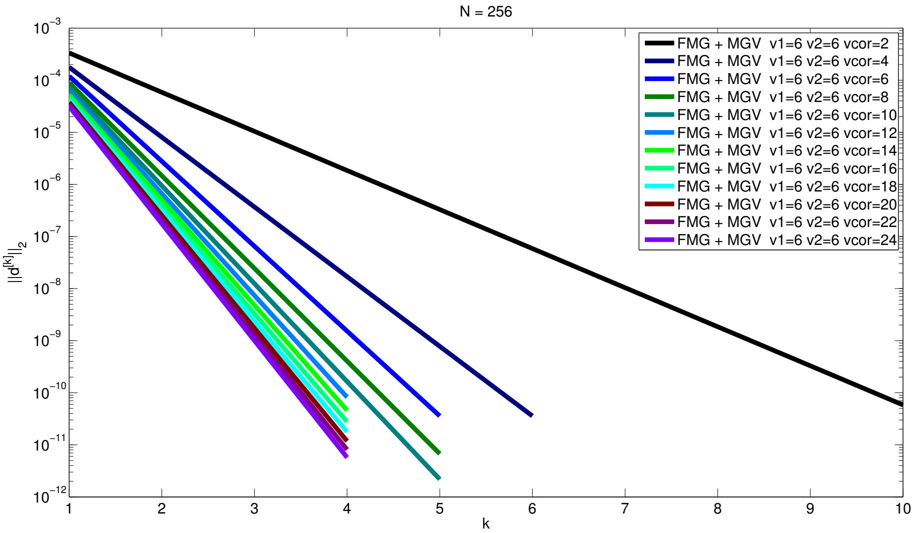

Figure 20: Testing the full multigrid method with RBGS on a grid of size $[257, 257]$ with higher values of v_1, v_2 and 6 subgrids

Here we see than unlike the previous example, here the values won't cause much change. We obtain higher precision as can be seen, but the number of iterations required are mostly the same. We expect this tendency to be true for higher grids aswell, as they'll benefit just as much from FMG. This is examined:

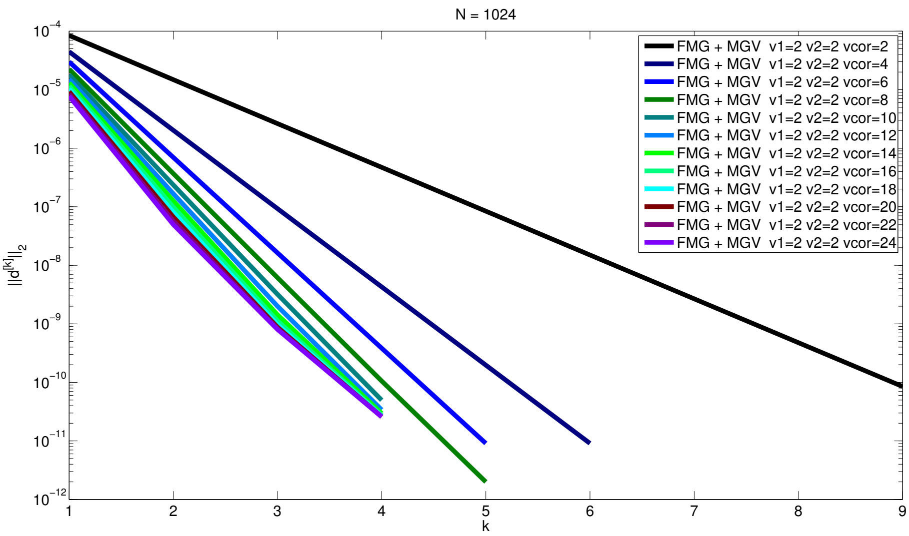

Figure 21: Testing the full multigrid method with RBGS on a grid of size $[1025, 1025]$ with low values of v_1, v_2 and 8 subgrids

We observe that higher grids seem to behave the same way, becoming much more efficient in terms of iterations.

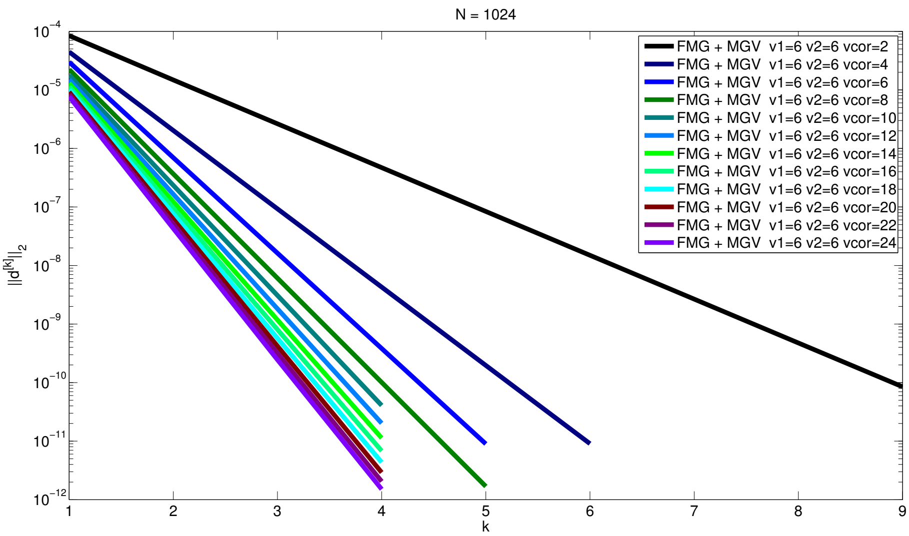

Figure 22: Testing the full multigrid method with RBGS on a grid of size $[1025, 1025]$ with higher values of v_1, v_2 and 8 subgrids

And as before, we see that there's not much point in high values of v_1, v_2 for grids of this size, unless we want much higher precision in few iterations.

If we compare FMG+MGV with just MGV, we find that the number of iterations required for a given precision level is reduces so drastically, that FMG would seem to be worth the effort. To look further into this, we need to see the timings of the methods. As such, these have been plotted:

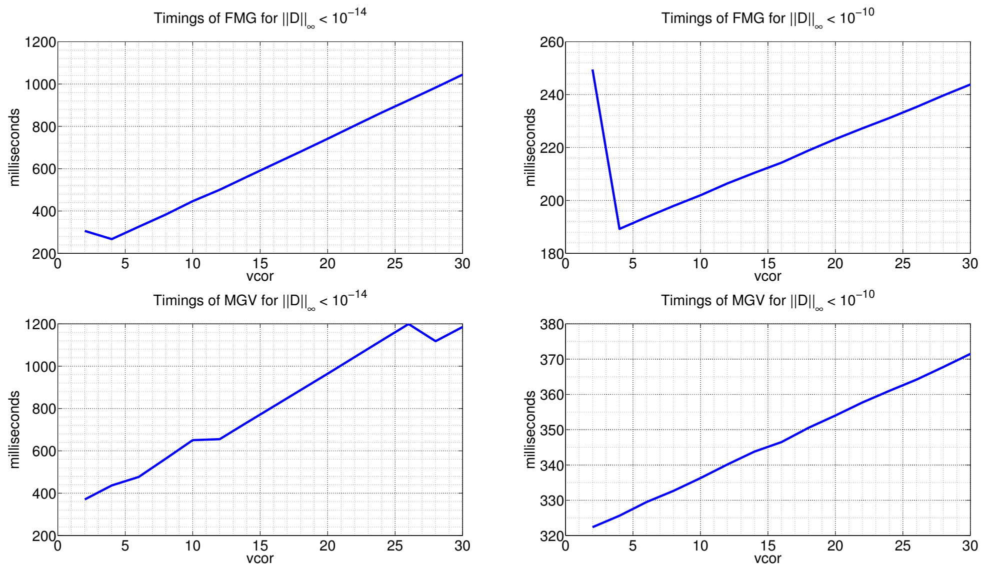

Figure 23: Solving times for FMG+MGV and MGV with $v_1 = v_2 = 2$ (mixed precision, RBGS)

We observe that the solving time is highly dependant on the required accuracy level and v_{cor} values. It makes sense that v_{cor} means less when we require very fine precision, as the work done on the coarser grids will become much less important once they've been used to create an acceptable approximation. We therefore also look at the timings for variations of v_1, v_2:

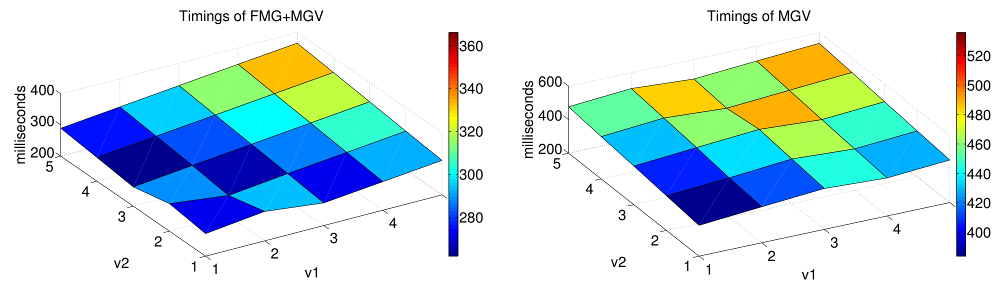

Figure 24: Solving times for FMG+MGV and MGV with $v_{cor} = 4$ (mixed precision, RBGS)

We see that higher values of v_1, v_2 results in higher total solving time for MGV, whereas FMG+MGV can benefit from slightly higher values of v_2, though this is likely to be specific for the given start guess and the given precision required. We could also try to give an estimate of the time required to solve the poisson problem, by looking at our algorithms. A single MGV iteration requires v_{cor} smoothings on the coarsest grid and $v_1 + v_2$ smoothings on all other grids. Furthermore, for all grids except for the coarsest, we need to perform one restriction, and for all grids except for the finest we need to peform one interpolation. On all grids except for the coarsest we need to calculate the defect. Lastly, in our implementation it's also required to perform 2 matrix times element operations on all grids except for the finest. Taking all of this together, we find that the time required for one iteration can be expressed as The total time used for restriction:

$$T_{RES} = \sum_{k=2}^{k_{max}} T_{RES}^{(k)}$$

The total time for interpolation:

$$T_{INT} = \sum_{k=1}^{k_{max}-1} T_{INT}^{(k)}$$

The total time for smoothing:

$$T_{SMOOTH} = v_{cor} \sum_{k=1}^{k_{max}-1} T_{SMOOTH}^{(k)} + (v_1 + v_2) \sum_{k=2}^{k_{max}} T_{SMOOTH}^{(k)}$$

The total time for defect calculation:

$$T_{DEF} = \sum_{k=2}^{k_{max}} T_{DEF}^{(k)}$$

The total time for matrix element operations:

$$T_{ME} = 2 \sum_{k=1}^{k_{max}-1} T_{ME}^{(k)}$$

Furthermore, since we also check if we've reached the desired level of accuracy, we have to add one defect calculation and one norm calculation on the finest grid:

$$T_{CHECK} = T_{DEF}^{(k_{max})} + T_{NORM}^{(k_{max})}$$

Using the above calculations, we can also find the time it takes to perform FMG. We don't need the norm calculator or the extra defect calculator on the finest grid. FMG is given by the sum of the times it takes to perform one MGV iteration per grid. Adding together all these timings, we should have an estimate of the time required. Doing this for the AMD6990 GPU we obtain the following times:

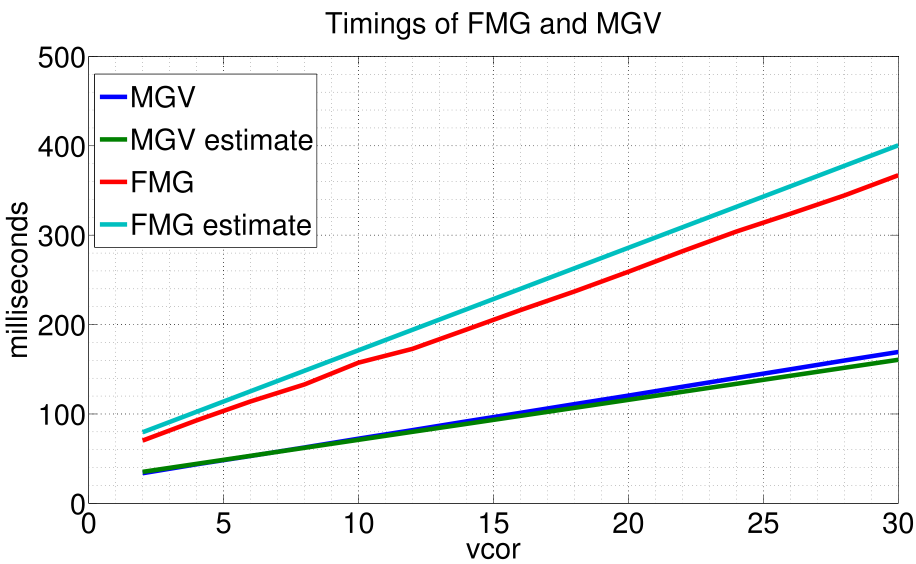

Figure 25: Comparing timings of FMG and MGV on the AMD 6990 GPU with estimates

As we can see, the FMG estimate shows off the right tendency, but it would seem to be slightly off. The MGV estimate on the other hand is very precise, though it's worth to note of course that MGV will be run multiple times, and as such, any inaccuracy will be multiplied aswell. It's unexpected that the FMG estimate is higher than the actual timing, and it would suggest that the timings of the individual components are slightly off. It's important here to note that it's just slightly, as each component is used alot, especially for the FMG method. Therefore, the estimate is still relatively good. Next we'll look at convergence rates for our multigrid solver. That is, how much we can decrease the defect per iteration. By using local fourier analysis (LGA)**??** we should be able to predict the convergence rates.

5.3 Examining convergence

We'll examine the convergence rates using LFA. We do this both the jacobi and RB-GS smoothers. The idea of local fourier analysis is that "any general discrete operator, nonlinear with nonconstant coefficients, can be linearized locally and replaced locally by an operator with constant coefficients"[10]. We employ the basis grid functions

$$\varphi_h(\theta, x) = e^{i\theta_1 x_1/h} e^{i\theta_2 x_2/h}$$

And seek to find this operator with constant coefficients for each of our operators. We do this so that we can analyse the two-grid convergence factor $\rho_{loc}(M_h^{2h})$. LFA is used on infinite grids, as such, boundaries are not taken into account. We define two grids given by

$$G_h = \{x = kh := (k_1 h, k_2 h),\ k \in \mathbb{Z}^2\}$$
$$G_{2h} = \{x = kh := (2k_1 h, 2k_2 h),\ k \in \mathbb{Z}^2\}$$

We see that they have several grid points in common, due to the definition of the grid points, which is important for two-grid analysis as it means that

$$\varphi_h(\theta + (\pm\pi, \pm\pi)), x) = \varphi_{2h}(2\theta, x)$$

The first operator we'll analyse is L_h. We try to find its local replacement $\tilde{L}_h$ with the equation

$$L_h \varphi(\theta, x) = \tilde{L}_h(\theta)\varphi(\theta, x), \quad (x \in G_h)$$

Since this operator can be described by a difference stencil, we know that the solution is given as[10]

$$\tilde{L}_h = \sum_{\kappa} s_\kappa e^{i\theta\cdot\kappa}$$

And by inserting the s_κ from the stencil, we have that

$$\tilde{L}_h(\theta) = \frac{1}{h^2}(4 - (e^{i\theta_1} + e^{-i\theta_1} + e^{i\theta_2} + e^{-i\theta_2}))$$

Next we'll look at the S operator. Both S^{JAC}, S^{Red-GS} and $S^{Black-GS}$ can be described by a combination of grid-points from the current iteration and grid-points from the next iteration, that is,

$$L_h^+ \overline{w}_h + L_h^- w_h = f_h$$

Where

$$L_h^- = \frac{1}{h^2} \begin{bmatrix} & -1 & \\ -1 & 0 & -1 \\ & -1 & \end{bmatrix}, \quad L_h^+ = \frac{1}{h^2} \begin{bmatrix} & 0 & \\ 0 & 4 & 0 \\ & 0 & \end{bmatrix}$$

We still have that these operators form the previous L_h operator:

$$L_h = L_h^+ + L_h^-$$

By substituting this into the equation we have that

$$L_h^+ \overline{v}_h + L_h^- v_h = 0$$

Which means that

$$\overline{v}_h = \frac{-L_h^-}{L_h^+} v_h$$

Which is the smoother S_h. From this we can deduce that

$$\tilde{S}(\theta, x) = \frac{-\tilde{L}_h^-}{\tilde{L}_h^-}$$

Since we already have a method of calculating $\tilde{L}_h(\theta)$, we can easily calculate $\tilde{L}_h^+(\theta)$ and $\tilde{L}_h^-(\theta)$. We find that

$$\begin{aligned} \tilde{L}_h^+(\theta) &= \frac{4}{h^2} \\ \tilde{L}_h^-(\theta) &= \frac{1}{h^2}(-e^{-i\theta_1} - e^{i\theta_1} - e^{-i\theta_2} - e^{i\theta_2}) \end{aligned}$$

Two-grid analysis is performed by first defining an operator M_h^{2h}:

$$M_h^{2h} = S_h^{v_2} K_h^{2h} S_h^{v_1}$$

Where K_h^{2h} is the coarse grid correction operator, that is, it's the operator that corrects all errors on a subgrid. As we have that

$$\varphi_h(\theta, x) = e^{i\theta_1 x_1/h} e^{i\theta_2 x_2/h}$$

We can define 4 θ values for the fine grid per value for the coarse grid:

$$\theta^{(0,0)} := (\theta_1, \theta_2) \quad \theta^{(1,1)} := (\overline{\theta}_1, \overline{\theta}_2)$$
$$\theta^{(1,0)} := (\overline{\theta}_1, \theta_2) \quad \theta^{(0,1)} := (\theta_1, \overline{\theta}_2)$$

Where

$$\overline{\theta}_i = \begin{cases} \theta_i + \pi & \text{if } \theta_i < 0 \\ \theta_i - \pi & \text{if } \theta_i \geq 0 \end{cases}$$

These 4 values of θ have the property that

$$\varphi_h(\theta^\alpha, x) = \varphi_h(\theta^{(0,0)}, x)$$

That is, the basis functions on the fine grid coincide on the coarser grid, making it impossible to distinguish between them. In order to deal with this, we define the space E_h^θ as the span of these four basis functions.

Next we create a coarse grid operator. The idea is that it should reduce the low frequencies.

$$\hat{I}_h - \hat{I}_{2h}^h(\hat{L}_{2h}(2\theta))^{-1} I_h^{2h}(\theta)\hat{L}_h(\theta)$$

With $\hat{I}_h$ being the $[4, 4]$ identity matrix. What this means is that the error on the fine grid is transfered to the coarse grid operator, it creates a copy of the error using the identity matrix, and then calculates the defect by $D = \hat{L}_h(\theta)e$. The defect is restricted, then calculated back to the exact error on the coarse grid, and interpolated up again, then subtracted from the error on the fine grid. As such, all errors on the coarse grid are removed by this operation. There's however one problem, this method doesn't work for pattern smoothers like the RBGS method. We reconize that the point of the error correcter is to elliminate the errors on a lower grid. Since we have influences from multiple gridpoints for each point in the coarser grid, we'll assume that all coarse grid points are correctly resolved instead. That is, we'll assume an error correction operator which elliminates any low frequencies. This is defined as

$$\hat{Q}_h^{2h} = \begin{bmatrix} 0 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

Using this instead of the error correction from before we define the two-grid convergence factor by

$$\rho(S_h^{v_2} Q_h^{2h} S_h^{v_1}) := \sup\{\rho(\hat{S}_h^{v_2}(\theta)\hat{Q}_h^{2h}\hat{S}_h^{v_1}(\theta)), \theta \in T^{low}\}$$

Due to Q being a identity matrix with one element removed, we can reduce the definition:

$$\rho(S_h^{v_2} Q_h^{2h} S_h^{v_1}) := \sup\{\rho(\hat{Q}_h^{2h} \hat{S}_h^{(v_1+v_2)}(\theta)), \theta \in T^{low}\}$$

We find that

$$\hat{S}_h^{RED} = \frac{1}{2}\begin{bmatrix} A+B & -A-B & & \\ -A+B & A-B & & \\ & & C+D & -C-D \\ & & -C+D & C-D \end{bmatrix}$$

And

$$\hat{S}_h^{BLACK} = \frac{1}{2}\begin{bmatrix} A+B & A+B & & \\ A-B & A-B & & \\ & & C+D & C+D \\ & & C-D & C-D \end{bmatrix}$$

With

$$A = 1,\ B = \frac{1}{2}\cos(\theta_1) + \frac{1}{2}\cos(\theta_2),\ C = 1,\ D = \frac{1}{2}\cos(\theta_1) - \frac{1}{2}\cos(\theta_2)$$

This means that

$$\hat{S}_h = \hat{S}_h^{BLACK}\hat{S}_h^{RED} = \frac{1}{4}\begin{bmatrix} 2B(A+B) & 2B(-A-B) & & \\ 2B(A-B) & 2B(-A+B) & & \\ & & 2D(C+D) & 2D(-C-D) \\ & & 2D(C-D) & 2D(-C+D) \end{bmatrix}$$

From this we can see that

$$\hat{S}_h^v = \frac{1}{2}\begin{bmatrix} B^{2v-1}(A+B) & B^{2v-1}(-A-B) & & \\ B^{2v-1}(A-B) & B^{2v-1}(-A+B) & & \\ & & D^{2v-1}(C+D) & D^{2v-1}(-C-D) \\ & & D^{2v-1}(C-D) & D^{2v-1}(-C+D) \end{bmatrix}$$

Which means we have an easy way to calculate $\hat{S}_h^v$ for various v. The form of the matrix also implies that we have an easy way to calculate it's eigenvalues, since

$$\hat{Q}\hat{S}_h^v = \frac{1}{2}\begin{bmatrix} & & & \\ B^{2v-1}(A-B) & B^{2v-1}(-A+B) & & \\ & & D^{2v-1}(C+D) & D^{2v-1}(-C-D) \\ & & D^{2v-1}(C-D) & D^{2v-1}(-C+D) \end{bmatrix}$$

By setting up the eigenvalues equation

$$\begin{aligned}
0x_1 + 0x_2 + 0x_3 + 0x_4 &= 2\lambda x_1 \\
B^{2v-1}(A-B)x_1 + B^{2v-1}(-A+B)x_2 + 0x_3 + 0x_4 &= 2\lambda x_2 \\
0x_1 + 0x_2 + D^{2v-1}(C+D)x_3 + D^{2v-1}(-C-D)x_4 &= 2\lambda x_3 \\
0x_1 + 0x_2 + D^{2v-1}(C-D)x_3 + D^{2v-1}(-C+D)x_4 &= 2\lambda x_4
\end{aligned}$$

From here we can see that x_1 must be 0. x_2 is used only in it's own line, and since $x_1 = 0$, it's rather easy to find a fitting λ for $x_2 \neq 0$. x_3, x_4 are dependant only on eachother. From this we can find the eigenvectors with non-zero eigenvalues. We know that there can be only 2, as there's only one possible value of λ for $x_2 \neq 0$, and likewise for $x_3 \neq 0,\ x_4 \neq 0$.

$$\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ C+D \\ C-D \end{bmatrix}$$

With associated eigenvalues $\lambda_1 = \frac{B^{2v-1}}{2}(-A+B),\ \lambda_2 = D^{2v}$. And we find that

v	1	2	3	4	5
ρ_{loc}	0.2500	0.0625	0.0335	0.0245	0.0194

Table 9: Estimated convergence factors of RBGS on poisson problem

We've found that these convergence factors hold for the first few steps, for sufficiently small grids. It would seem that the coarse grid corrector is unable to cope on larger grids. We've examined the convergence factors for a grid of size 129x129, with a coarsest grid of size 5x5, and found the following convergence factors, measured as $\rho = \frac{D^{[K-1]}}{D^{[K]}}$, where K is the last iteration before the solver has converged. 20 iterations was used on the coarsest grid.

v	1	2	3	4	5
ρ	0.3	0.074	0.063	0.056	0.045

Table 10: Measured convergence factors of RBGS on poisson problem

As can be seen, our convergence estimates are higher than the estimates. It would seem that Q isn't a good enough replacement for the coarse grid corrector in this case. The estimates given are not bad though. Even if they're too low, then they can still give us an estimate of the number of iterations required. We'll have to expect a few extra iterations, but we can estimate the required number of iterations for a solver, given the initial defect.

In the following we'll look at the wave equation. The wave equation is a variant of the poisson equation. It is given as

$$\begin{aligned} \phi = \tilde{\phi},\ z = \zeta, \\ \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial z^2} = 0,\ -h \leq z < \zeta \\ (n_x, n_z)^T \cdot (\partial_x, \partial_y)\phi = 0,\ (x, z) \in \partial\Omega \end{aligned}$$

The physical situation is a water-tank with depth h and free surface elevation ζ. The poisson RBGS solver has been implemented in GPULAB[5]. We can show that the solvers behave as expected. If we discretize the problem to have 35 grid points in the x direction and 9 in the z direction, and without using subgrids, we obtain

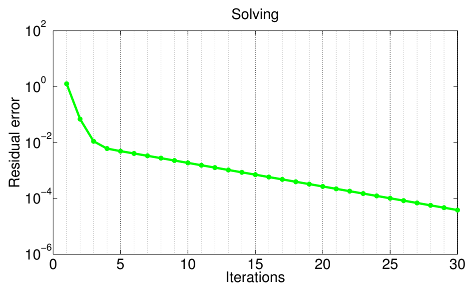

Figure 26: Solving the wave equation using RBGS with $v_1 = 2, v_2 = 2, v_{cor} = 20$

And as expected, we obtain a rather bad convergence. Like before, we can improve this by using multiple grids:

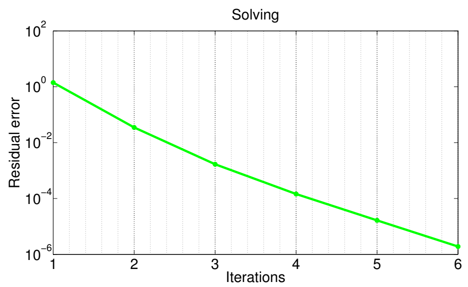

Figure 27: Solving the wave equation using RBGS with $v_1 = 2, v_2 = 2, v_{cor} = 20$ and 2 subgrids

[5] A library developed at IMM for easier modelling with GPU's in CUDA.

And we obtain a quite good convergence rate. In the previous chapter, we found that increases to v_1, v_2 would result in mostly no gain, while being more expensive. This is examined:

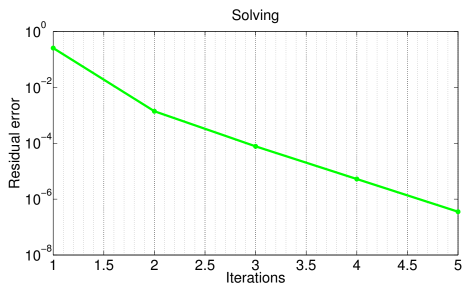

Figure 28: Solving the wave equation using RBGS with $v_1 = 6, v_2 = 6, v_{cor} = 20$ and 2 subgrids

While we save one iteration, the cost per iteration increases from $0.008479s$ to $0.011618s$, and with the number of iterations used, nothing is saved by performing many pre and post smoothings.

6 Wave problem

The wave equation as said, is a variant of the poisson equation. It is given as

$$\begin{aligned} \phi = \tilde{\phi},\ z = \zeta, \\ \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial z^2} = 0,\ -h \leq z < \zeta \\ (n_x, n_z)^T \cdot (\partial_x, \partial_y)\phi = 0,\ (x, z) \in \partial\Omega \end{aligned}$$

The physical situation is a water-tank with depth h and free surface elevation ζ. The poisson RBGS solver has been implemented in GPULAB[6]. We can show that the solvers behave as expected. If we discretize the problem to have 35 grid points in the x direction and 9 in the z direction, and without using subgrids, we obtain

[6] A library developed at IMM for easier modelling with GPU's in CUDA.

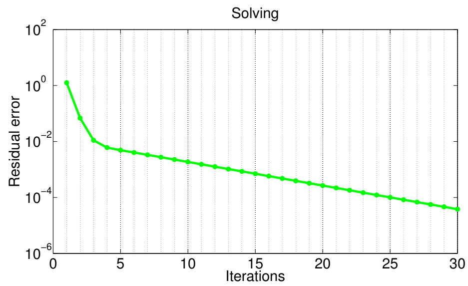

Figure 29: Solving the wave equation using RBGS with $v_1 = 2, v_2 = 2, v_{cor} = 20$

And as expected, we obtain a rather bad convergence. Like before, we can improve this by using multiple grids:

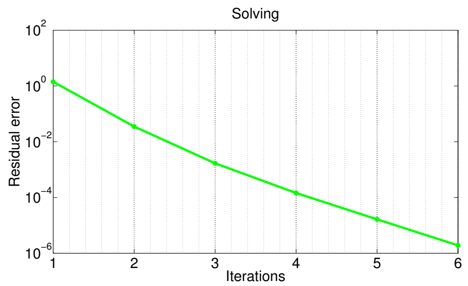

Figure 30: Solving the wave equation using RBGS with $v_1 = 2, v_2 = 2, v_{cor} = 20$ and 2 subgrids

And we obtain a quite good convergence rate. In the previous chapter, we found that increases to v_1, v_2 would result in mostly no gain, while being more expensive. This is examined:

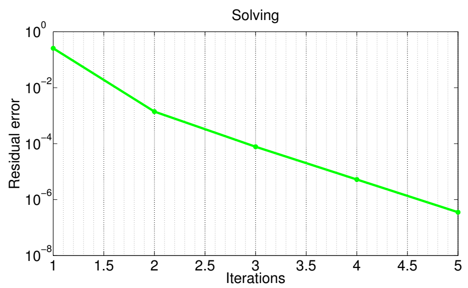

Figure 31: Solving the wave equation using RBGS with $v_1 = 6, v_2 = 6, v_{cor} = 20$ and 2 subgrids

While we save one iteration, the cost per iteration increases from $0.008479s$ to $0.011618s$, and with the number of iterations used, nothing is saved by performing many pre and post smoothings.

7 Conclusion

With the current implementation of the API, it would be recommended to use Nvidia GPU's. This could change with the new series of AMD GPU's, the 7000 series, as there's been a change in the architecture. We've also shown that MATLAB routines can be sped up using OpenCL, to take advantage of the GPU instead of the CPU. Furthermore, while it may be hard to make MATLAB use external libraries, it would seem that an API built specifically for MATLAB can fit really well. Both the AMD6990 and the Nvidia 590 GPU's benefitted mostly equal from autotuning, though the 590 GPU seemed to be better suited for the kernels. This could very well be a consequence of the architectural differences between the 6990 and the 590 GPU's. None the less, both cards proved themselves capable of speeding up MATLAB code through OpenCL.

Another interesting result came from the specialized components for the poisson problem. Individually they were faster than the corresponding linear algebra implementations, but in most cases it was by less than a factor of 6. So while they can be used to speed up the solving time of a problem, then they're not neccersary unless time is critical, at which point, they can be used to supplement the API if needed. The specialized components could of course be improved even more to gain more speed, but the disadvantage is the loss of protyping speed. The poisson problem was solved as expected, and converged as it should. As such, we've proven that the individual parts of the API works as intended, and if we were to solve a new problem, we won't have to worry about errors in the API.

Furthermore we saw that MATLAB through MEX can communicate with a library based entirely around pointers. This allowed us to create a consistent library, as opposed to a library where the initialization would have to be done before every action.

8 Further studies

It could be interesting to look into the usage of multiple GPU's to further speed up this library. Refering again to gustafsons law, there's a lot to gain, if we could connect the GPU's.

It could be interesting to look into expanding the library, including the implementation of matrices, like the upper diagonal matrix. It could also be interesting to just implement more methods like the vector minus vector times constant, as these methods can speed up the library, offering more computations for the same bandwidth.

9 References

References

[1] "Matthew Scarpino". *OpenCL in Action*. Manning, 2011.

[2] "John L. Gustafson". Reevaluating amdahl's law. http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.html, July 2012.

[3] "Khronos Group". Opencl - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, July 2012.

[4] "Nvidia". What is cuda. http://developer.nvidia.com/what-cuda, July 2012.

[5] "Nvidia". Geforce 590 specifications. http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-590/specifications, July 2012.

[6] "AMD". Radeon 6990 specifications. http://www.amd.com/us/products/desktop/graphics/amd-radeon-hd-6000/hd-6990/pages/amd-radeon-hd-6990-overview.aspx#3, July 2012.

[7] "Wikipedia.org". Basic linear algebra subprograms. http://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms, July 2012.

[8] "Accelereyes". Arrayfire. http://www.accelereyes.com/products/jacket, July 2012.

[9] Nvidia. NVIDIA OpenCL Best: Practices Guide. http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf, August 2009.

[10] "Ulrich Trottenberg", "Cornelis Oosterlee", and "Anton Schüller". *Multigrid*. Elsevier, 2001.

[11] "Alfredo Buttari", "Jack Dongarra", "Jakub Kurzak", "Piotr Luszczek", and "Stanimore Tomov". Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy. http://www.netlib.org/lapack/lawnspdf/lawn180.pdf, October 2006.

10 Appendix

10.1 OpenCL kernels

All kernels have versions for doubles, floats and mixing of these, as such, there'll be many mostly identical kernels.

10.1.1 Sparse matrix times vector kernel

```
#include <KernelHeaders.h>
__kernel void SparseMatrixVectorFF(__global float * matData,
    __global unsigned int * matCol, __global unsigned int *
    matRow, __global float * vecData, __global float *
    returnData, unsigned int rowVectorLength)
{
  unsigned int j;
  unsigned int i = get_global_id(0);
  float temp;
  unsigned int start, end, col;
  while (i < rowVectorLength-1)
  {
    start = matRow[i];
    end = matRow[i+1];
    temp = matData[start] * vecData[matCol[start]];
    for (j = start+1; j < end; j += 1)
    {
      col = matCol[j];
      temp += matData[j] * vecData[col];
    }
    returnData[i] = temp;
    i += get_global_size(0);
  }
}

#ifdef __DOUBLE_ALLOWED__

__kernel void SparseMatrixVectorDF(__global double * matData,
    __global unsigned int * matCol, __global unsigned int *
    matRow, __global float * vecData, __global double *
    returnData, unsigned int rowVectorLength)
{
  int a = 0;
}

__kernel void SparseMatrixVectorDD(__global double * matData,
    __global unsigned int * matCol, __global unsigned int *
```

```
    matRow, __global double * vecData, __global double *
    returnData, unsigned int rowVectorLength)
{
  unsigned int j;
  unsigned int i = get_global_id(0);
  double temp;
  unsigned int start, end, col;
  while (i < rowVectorLength-1)
  {
    start = matRow[i];
    end = matRow[i+1];
    temp = matData[start] * vecData[matCol[start]];
    for (j = start+1; j < end; j += 1)
    {
      col = matCol[j];
      temp += matData[j] * vecData[col];
    }
    returnData[i] = temp;
    i += get_global_size(0);
  }
}

__kernel void SparseMatrixVectorFD(__global float * matData,
    __global unsigned int * matCol, __global unsigned int *
    matRow, __global double * vecData, __global float *
    returnData, unsigned int rowVectorLength)
{
  int a = 0;
}

#endif
```

10.1.2 Band matrix times vector kernel

```
#include <KernelHeaders.h>
__kernel void BandMatrixVectorFF(__global float * matData,
    __global float * vecData, __global float * returnData,
    unsigned int width, unsigned int bandwidth, unsigned int
    length)
{
  int j;
  int i = get_global_id(0);
  int low = i - bandwidth;
  int high = i + bandwidth;
  float temp = 0.0;
  for (j = low; j <= high; j += 1)
```

```
  {
    if ((j ≥ 0) && (j < width))
    {
      temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
    }
  }
  returnData[i] = temp;
}

#ifdef __DOUBLE_ALLOWED__

__kernel void BandMatrixVectorFD(__global float * matData,
    __global double * vecData, __global double * returnData,
    unsigned int width, unsigned int bandwidth, unsigned int
    length)
{
  int j;
  int i = get_global_id(0);
  int low = i - bandwidth;
  int high = i + bandwidth;
  double temp = 0.0;
  for (j = low; j ≤ high; j += 1)
  {
    if ((j ≥ 0) && (j < width))
    {
      temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
    }
  }
  returnData[i] = temp;
}

__kernel void BandMatrixVectorDD(__global double * matData,
    __global double * vecData, __global double * returnData,
    unsigned int width, unsigned int bandwidth, unsigned int
    length)
{
  int j;
  int i = get_global_id(0);
  int low = i - bandwidth;
  int high = i + bandwidth;
  double temp = 0.0;
  for (j = low; j ≤ high; j += 1)
  {
    if ((j ≥ 0) && (j < width))
    {
      temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
    }
  }
  returnData[i] = temp;
```

```
}

__kernel void BandMatrixVectorDF(__global double * matData,
    __global float * vecData, __global float * returnData,
    unsigned int width, unsigned int bandwidth, unsigned int
    length)
{
  int j;
  int i = get_global_id(0);
  int low = i - bandwidth;
  int high = i + bandwidth;
  double temp = 0.0;
  for (j = low; j ≤ high; j += 1)
  {
    if ((j ≥ 0) && (j < width))
    {
      temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
    }
  }
  returnData[i] = temp;
}

#endif
```

10.1.3 Upper diagonal matrix times vector kernel

```
#include <KernelHeaders.h>
__kernel void BandMatrixVectorFF(__global float * matData,
    __global float * vecData, __global float * returnData,
    unsigned int width, unsigned int bandwidth, unsigned int
    length)
{
  int j;
  int i = get_global_id(0);
  int low = i - bandwidth;
  int high = i + bandwidth;
  float temp = 0.0;
  for (j = low; j ≤ high; j += 1)
  {
    if ((j ≥ 0) && (j < width))
    {
      temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
    }
  }
  returnData[i] = temp;
}
```

```

#ifdef __DOUBLE_ALLOWED__

__kernel void BandMatrixVectorFD(__global float * matData, __global double * vecData, __global double * returnData, unsigned int width, unsigned int bandwidth, unsigned int length)
{
  int j;
  int i = get_global_id(0);
  int low = i - bandwidth;
  int high = i + bandwidth;
  double temp = 0.0;
  for (j = low; j <= high; j += 1)
  {
    if ((j >= 0) && (j < width))
    {
      temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
    }
  }
  returnData[i] = temp;
}

__kernel void BandMatrixVectorDD(__global double * matData, __global double * vecData, __global double * returnData, unsigned int width, unsigned int bandwidth, unsigned int length)
{
  int j;
  int i = get_global_id(0);
  int low = i - bandwidth;
  int high = i + bandwidth;
  double temp = 0.0;
  for (j = low; j <= high; j += 1)
  {
    if ((j >= 0) && (j < width))
    {
      temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
    }
  }
  returnData[i] = temp;
}

__kernel void BandMatrixVectorDF(__global double * matData, __global float * vecData, __global float * returnData, unsigned int width, unsigned int bandwidth, unsigned int length)
{
  int j;
```

```
  int i = get_global_id(0);
  int low = i - bandwidth;
  int high = i + bandwidth;
  double temp = 0.0;
  for (j = low; j <= high; j += 1)
  {
    if ((j >= 0) && (j < width))
    {
      temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
    }
  }
  returnData[i] = temp;
}

#endif
```

10.1.4 Vector times constant kernel

```
#include <KernelHeaders.h>
__kernel void VectorTimesConstantFF(__global float* vector,
    __global float* output, float element)
{
  int i = get_global_id(0);
  output[i] = vector[i] * element;
}


#ifdef __DOUBLE_ALLOWED__
__kernel void VectorTimesConstantFD(__global float* vector,
    __global float* output, double element)
{
  int i = get_global_id(0);
  output[i] = vector[i] * element;
}

__kernel void VectorTimesConstantDD(__global double* vector,
    __global double* output, double element)
{
  int i = get_global_id(0);
  output[i] = vector[i] * element;
}
#endif
```

10.1.5 Vector plus vector kernel

```
#include <KernelHeaders.h>
__kernel void VectorPlusVectorFF(__global float* vector1,
    __global float* vector2, __global float* output)
{
  int i = get_global_id(0);
  output[i] = vector1[i] + vector2[i];
}


#ifdef __DOUBLE_ALLOWED__
__kernel void VectorPlusVectorFD(__global float* vector1,
    __global double* vector2, __global float* output)
{
  int i = get_global_id(0);
  output[i] = vector1[i] + vector2[i];
}

__kernel void VectorPlusVectorDD(__global double* vector1,
    __global double* vector2, __global double* output)
{
  int i = get_global_id(0);
  output[i] = vector1[i] + vector2[i];
}

__kernel void VectorPlusVectorDF(__global double* vector1,
    __global float* vector2, __global double* output)
{
  int i = get_global_id(0);
  output[i] = vector1[i] + vector2[i];
}
#endif
```

10.1.6 Vector minus vector kernel

```
#include <KernelHeaders.h>
__kernel void VectorMinusVectorFF(__global float* vector1,
    __global float* vector2, __global float* output)
{
  int i = get_global_id(0);
  output[i] = vector1[i] - vector2[i];
}


#ifdef __DOUBLE_ALLOWED__
```

```
__kernel void VectorMinusVectorFD(__global float* vector1,
    __global double* vector2, __global float* output)
{
  int i = get_global_id(0);
  output[i] = vector1[i] - vector2[i];
}

__kernel void VectorMinusVectorDD(__global double* vector1,
    __global double* vector2, __global double* output)
{
  int i = get_global_id(0);
  output[i] = vector1[i] - vector2[i];
}

__kernel void VectorMinusVectorDF(__global double* vector1,
    __global float* vector2, __global double* output)
{
  int i = get_global_id(0);
  output[i] = vector1[i] - vector2[i];
}
#endif
```

10.1.7 Vector minus vector times constant kernel

```
#include <KernelHeaders.h>
__kernel void VectorMinusVectorConstantFF(__global float*
    vector1, __global float* vector2, __global float* output,
    float con, unsigned int length)
{
  int i = get_global_id(0);
  while (i < length)
  {
    output[i] = vector1[i] - con*vector2[i];
    i += get_global_size(0);
  }
}


#ifdef __DOUBLE_ALLOWED__
__kernel void VectorMinusVectorConstantFD(__global float*
    vector1, __global double* vector2, __global float* output,
     double con, unsigned int length)
{
  int i = get_global_id(0);
  while (i < length)
  {
```

```
    output[i] = vector1[i] - con*vector2[i];
    i += get_global_size(0);
  }
}

__kernel void VectorMinusVectorConstantDD(__global double*
    vector1, __global double* vector2, __global double* output
    , double con, unsigned int length)
{
  int i = get_global_id(0);
  while (i < length)
  {
    output[i] = vector1[i] - con*vector2[i];
    i += get_global_size(0);
  }
}

__kernel void VectorMinusVectorConstantDF(__global double*
    vector1, __global float* vector2, __global double* output,
     double con, unsigned int length)
{
  int i = get_global_id(0);
  while (i < length)
  {
    output[i] = vector1[i] - con*vector2[i];
    i += get_global_size(0);
  }
}
#endif
```

10.1.8 Vector sum kernel

```
#include <KernelHeaders.h>
__kernel void VectorReductionF(__global float* input,__global
    float* output, unsigned int threadSize, unsigned int
    problemSize)
{
  //Declare shared memory space:
  __local float sdata[512];
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0)*(threadSize*2) + tid;
  unsigned int gridSize = threadSize*2*get_num_groups(0);
  float tempCounter = 0.0;
  while (i < problemSize)
  {
    tempCounter += input[i];
```

```
    if (i + threadSize < problemSize)
    {
      tempCounter += input[i + threadSize];
    }
    i += gridSize;
  }
  sdata[tid] = tempCounter;
  barrier(CLK_LOCAL_MEM_FENCE);
  if (threadSize >= 512)
  {
    if (tid < 256)
    {
      sdata[tid] += sdata[tid + 256];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (threadSize >= 256)
  {
    if (tid < 128)
    {
      sdata[tid] += sdata[tid + 128];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (threadSize >= 128)
  {
    if (tid < 64)
    {
      sdata[tid] += sdata[tid + 64];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid < 32)
  {
    if (threadSize >= 64)
    {
      sdata[tid] += sdata[tid + 32];
    }
    if (threadSize >= 32)
    {
      sdata[tid] += sdata[tid + 16];
    }
    if (threadSize >= 16)
    {
      sdata[tid] += sdata[tid + 8];
    }
    if (threadSize >= 8)
    {
      sdata[tid] += sdata[tid + 4];
```

```
    }
    if (threadSize ≥ 4)
    {
      sdata[tid] += sdata[tid + 2];
    }
    if (threadSize ≥ 2)
    {
      sdata[tid] += sdata[tid + 1];
    }
  }
  if (tid == 0)
  {
    output[get_group_id(0)] = sdata[0];
  }
}

#ifdef __DOUBLE_ALLOWED__
__kernel void VectorReductionD(__global double* input, __global double* output, unsigned int threadSize, unsigned int problemSize)
{
  //Declare shared memory space:
  __local double sdata[512];
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0)*(threadSize*2) + tid;
  unsigned int gridSize = threadSize*2*get_num_groups(0);
  double tempCounter = 0.0;
  while (i < problemSize)
  {
    tempCounter += input[i];
    if (i + threadSize < problemSize)
    {
      tempCounter += input[i + threadSize];
    }
    i += gridSize;
  }
  sdata[tid] = tempCounter;
  barrier(CLK_LOCAL_MEM_FENCE);
  if (threadSize ≥ 512)
  {
    if (tid < 256)
    {
      sdata[tid] += sdata[tid + 256];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (threadSize ≥ 256)
  {
    if (tid < 128)
```

```
    {
      sdata[tid] += sdata[tid + 128];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (threadSize >= 128)
  {
    if (tid < 64)
    {
      sdata[tid] += sdata[tid + 64];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid < 32)
  {
    if (threadSize >= 64)
    {
      sdata[tid] += sdata[tid + 32];
    }
    if (threadSize >= 32)
    {
      sdata[tid] += sdata[tid + 16];
    }
    if (threadSize >= 16)
    {
      sdata[tid] += sdata[tid + 8];
    }
    if (threadSize >= 8)
    {
      sdata[tid] += sdata[tid + 4];
    }
    if (threadSize >= 4)
    {
      sdata[tid] += sdata[tid + 2];
    }
    if (threadSize >= 2)
    {
      sdata[tid] += sdata[tid + 1];
    }
  }
  if (tid == 0)
  {
    output[get_group_id(0)] = sdata[0];
  }
}
#endif
```

10.1.9 Vector 2-norm kernel

```
#include <KernelHeaders.h>
__kernel void Norm2F(__global float* input, __global float* output, unsigned int threadSize, unsigned int problemSize)
{
  //Declare shared memory space:
  __local float sdata[512];
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0)*(threadSize*2) + tid;
  unsigned int gridSize = threadSize*2*get_num_groups(0);
  float tempCounter = 0.0;
  while (i < problemSize)
  {
    tempCounter += input[i]*input[i];
    if (i + threadSize < problemSize)
    {
      tempCounter += input[i + threadSize]*input[i + threadSize];
    }
    i += gridSize;
  }
  sdata[tid] = tempCounter;
  barrier(CLK_LOCAL_MEM_FENCE);
  if (threadSize >= 512)
  {
    if (tid < 256)
    {
      sdata[tid] += sdata[tid + 256];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (threadSize >= 256)
  {
    if (tid < 128)
    {
      sdata[tid] += sdata[tid + 128];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (threadSize >= 128)
  {
    if (tid < 64)
    {
      sdata[tid] += sdata[tid + 64];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
```

```
  }
  if (tid < 32)
  {
    if (threadSize >= 64)
    {
      sdata[tid] += sdata[tid + 32];
    }
    if (threadSize >= 32)
    {
      sdata[tid] += sdata[tid + 16];
    }
    if (threadSize >= 16)
    {
      sdata[tid] += sdata[tid + 8];
    }
    if (threadSize >= 8)
    {
      sdata[tid] += sdata[tid + 4];
    }
    if (threadSize >= 4)
    {
      sdata[tid] += sdata[tid + 2];
    }
    if (threadSize >= 2)
    {
      sdata[tid] += sdata[tid + 1];
    }
  }
  if (tid == 0)
  {
    output[get_group_id(0)] = sdata[0];
  }
}


#ifdef __DOUBLE_ALLOWED__
__kernel void Norm2D(__global double* input, __global double* output, unsigned int threadSize, unsigned int problemSize)
{
  //Declare shared memory space:
  __local double sdata[512];
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0)*(threadSize*2) + tid;
  unsigned int gridSize = threadSize*2*get_num_groups(0);
  double tempCounter = 0.0;
  while (i < problemSize)
  {
    tempCounter += input[i]*input[i];
    if (i + threadSize < problemSize)
```

```
  {
    tempCounter += input[i + threadSize]*input[i + threadSize];
  }
  i += gridSize;
}
sdata[tid] = tempCounter;
barrier(CLK_LOCAL_MEM_FENCE);
if (threadSize >= 512)
{
  if (tid < 256)
  {
    sdata[tid] += sdata[tid + 256];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
if (threadSize >= 256)
{
  if (tid < 128)
  {
    sdata[tid] += sdata[tid + 128];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
if (threadSize >= 128)
{
  if (tid < 64)
  {
    sdata[tid] += sdata[tid + 64];
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
if (tid < 32)
{
  if (threadSize >= 64)
  {
    sdata[tid] += sdata[tid + 32];
  }
  if (threadSize >= 32)
  {
    sdata[tid] += sdata[tid + 16];
  }
  if (threadSize >= 16)
  {
    sdata[tid] += sdata[tid + 8];
  }
  if (threadSize >= 8)
  {
    sdata[tid] += sdata[tid + 4];
```

```
    }
    if (threadSize ≥ 4)
    {
      sdata[tid] += sdata[tid + 2];
    }
    if (threadSize ≥ 2)
    {
      sdata[tid] += sdata[tid + 1];
    }
  }
  if (tid == 0)
  {
    output[get_group_id(0)] = sdata[0];
  }
}
#endif
```

10.1.10 Vector ∞-norm kernel

```
#include <KernelHeaders.h>
__kernel void NormInfF(__global float* input, __global float*
    output, unsigned int threadSize, unsigned int problemSize)
{
  //Declare shared memory space:
  __local float sdata[512];
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0)*(threadSize*2) + tid;
  unsigned int gridSize = threadSize*2*get_num_groups(0);
  float tempCounter = 0.0;
  float tempNew;
  while (i < problemSize)
  {
    tempNew = input[i]; if (tempNew < 0) tempNew = -tempNew;
    if (tempNew > tempCounter) tempCounter = tempNew;
    if (i + threadSize < problemSize)
    {
      tempNew = input[i + threadSize]*input[i + threadSize];
          if (tempNew < 0) tempNew = -tempNew;
      if (tempNew > tempCounter) tempCounter = tempNew;
    }
    i += gridSize;
  }
  sdata[tid] = tempCounter;
  barrier(CLK_LOCAL_MEM_FENCE);
  if (threadSize ≥ 512)
  {
```

```
    if (tid < 256 && sdata[tid + 256] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 256];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (threadSize ≥ 256)
  {
    if (tid < 128 && sdata[tid + 128] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 128];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (threadSize ≥ 128)
  {
    if (tid < 64 && sdata[tid + 64] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 64];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid < 32)
  {
    if (threadSize ≥ 64 && sdata[tid + 32] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 32];
    }
    if (threadSize ≥ 32 && sdata[tid + 16] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 16];
    }
    if (threadSize ≥ 16 && sdata[tid + 8] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 8];
    }
    if (threadSize ≥ 8 && sdata[tid + 4] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 4];
    }
    if (threadSize ≥ 4 && sdata[tid + 2] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 2];
    }
    if (threadSize ≥ 2 && sdata[tid + 1] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 1];
    }
  }
```

```
  if (tid == 0)
  {
    output[get_group_id(0)] = sdata[0];
  }
}

#ifdef __DOUBLE_ALLOWED__
__kernel void NormInfD(__global double* input, __global double* output, unsigned int threadSize, unsigned int problemSize)
{
  //Declare shared memory space:
  __local double sdata[512];
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0)*(threadSize*2) + tid;
  unsigned int gridSize = threadSize*2*get_num_groups(0);
  double tempCounter = 0.0;
  double tempNew;
  while (i < problemSize)
  {
    tempNew = input[i]; if (tempNew < 0) tempNew = -tempNew;
    if (tempNew > tempCounter) tempCounter = tempNew;
    if (i + threadSize < problemSize)
    {
      tempNew = input[i + threadSize]*input[i + threadSize]; if (tempNew < 0) tempNew = -tempNew;
      if (tempNew > tempCounter) tempCounter = tempNew;
    }
    i += gridSize;
  }
  sdata[tid] = tempCounter;
  barrier(CLK_LOCAL_MEM_FENCE);
  if (threadSize >= 512)
  {
    if (tid < 256 && sdata[tid + 256] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 256];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (threadSize >= 256)
  {
    if (tid < 128 && sdata[tid + 128] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 128];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (threadSize >= 128)
```

```
  {
    if (tid < 64 && sdata[tid + 64] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 64];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid < 32)
  {
    if (threadSize ≥ 64 && sdata[tid + 32] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 32];
    }
    if (threadSize ≥ 32 && sdata[tid + 16] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 16];
    }
    if (threadSize ≥ 16 && sdata[tid + 8] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 8];
    }
    if (threadSize ≥ 8 && sdata[tid + 4] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 4];
    }
    if (threadSize ≥ 4 && sdata[tid + 2] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 2];
    }
    if (threadSize ≥ 2 && sdata[tid + 1] > sdata[tid])
    {
      sdata[tid] = sdata[tid + 1];
    }
  }
  if (tid == 0)
  {
    output[get_group_id(0)] = sdata[0];
  }
}

#endif
```

10.1.11 Jacobi method kernel

```
#include <KernelHeaders.h>
```

```
__kernel void JacobiMethodF(__global float * output, __global float * matData, __global float * rhsData, unsigned int width, unsigned int height, float h)
{
  int j = get_global_id(1)+1;
  int i = get_global_id(0)+1;
  if (i < width-1)
  {
    float temp = (matData[i + (j-1)*width] + matData[i + (j+1)*width] + matData[i-1 + j*width] + matData[i+1 + j*width] - rhsData[i + j*width]*h*h)/4.0;
    output[i + j*width] = temp;
  }
}

#ifdef __DOUBLE_ALLOWED__

__kernel void JacobiMethodD(__global double * output, __global double * matData, __global double * rhsData, unsigned int width, unsigned int height, double h)
{
  int j = get_global_id(1)+1;
  int i = get_global_id(0)+1;
  if (i < width-1)
  {
    double temp = (matData[i + (j-1)*width] + matData[i + (j+1)*width] + matData[i-1 + j*width] + matData[i+1 + j*width] - rhsData[i + j*width]*h*h)/4.0;
    output[i + j*width] = temp;
  }
}

#endif
```

10.1.12 RBGS - Red kernel

```
#include <KernelHeaders.h>
__kernel void JacobiMethodEvenF(__global float * matData, __global float * rhsData, unsigned int width, unsigned int height, float h)
{
  int j = get_global_id(1)+1;
  int i = get_global_id(0)*2+1+j%2;
  if (i < width-1)
  {
```

```
        float temp = (matData[i + (j-1)*width] + matData[i + (j+1)*width] + matData[i-1 + j*width] + matData[i+1 + j*width] - rhsData[i + j*width]*h*h)/4.0;
        matData[i + j*width] = temp;
    }
}

#ifdef __DOUBLE_ALLOWED__

__kernel void JacobiMethodEvenD(__global double * matData, __global double * rhsData, unsigned int width, unsigned int height, double h)
{
    int j = get_global_id(1)+1;
    int i = get_global_id(0)*2+1+j%2;
    if (i < width-1)
    {
        double temp = (matData[i + (j-1)*width] + matData[i + (j+1)*width] + matData[i-1 + j*width] + matData[i+1 + j*width] - rhsData[i + j*width]*h*h)/4.0;
        matData[i + j*width] = temp;
    }
}

#endif
```

10.1.13 RBGS - Black kernel

```
#include <KernelHeaders.h>
__kernel void JacobiMethodOddF(__global float * matData, __global float * rhsData, unsigned int width, unsigned int height, float h)
{
    int j = get_global_id(1)+1;
    int i = get_global_id(0)*2+1+(j+1)%2;
    if (i < width-1)
    {
        float temp = (matData[i + width*(j-1)] + matData[i + width*(j+1)] + matData[i-1 + j*width] + matData[i+1 + width*j] - rhsData[i + width*j]*h*h)/4.0;
        matData[i + j*width] = temp;
    }
}


#ifdef __DOUBLE_ALLOWED__
```

```

__kernel void JacobiMethodOddD(__global double * matData,
    __global double * rhsData, unsigned int width, unsigned
    int height, double h)
{
  int j = get_global_id(1)+1;
  int i = get_global_id(0)*2+1+(j+1)%2;
  if (i < width-1)
  {
    double temp = (matData[i + width*(j-1)] + matData[i +
        width*(j+1)] + matData[i-1 + j*width] + matData[i+1 +
        width*j] - rhsData[i + width*j]*h*h)/4.0;
    matData[i + j*width] = temp;
  }
}

#endif
```

10.1.14 Poisson defect kernel

```
#include <KernelHeaders.h>
__kernel void JacobiCalcDefectF(__global float * matData,
    __global float * rhsData, __global float * defectData,
    unsigned int width, unsigned int height, float h)
{
  int i = get_global_id(0);
  int j = get_global_id(1);
  float temp = 0.0;
  if ( (i == 0) || (j == 0) || (i == get_global_size(0)-1) ||
      (j == get_global_size(1)-1) )
  {
    //nothing
  }
  else
  {
    temp = -rhsData[i + j*width]*h*h - ( 4*matData[i + j*width
        ] - matData[i-1 + j*width] - matData[i+1 + j*width] -
        matData[i + (j+1)*width] - matData[i + (j-1)*width] );
  }
  defectData[i + j*width] = temp;
}


#ifdef __DOUBLE_ALLOWED__

```

```
__kernel void JacobiCalcDefectD(__global double * matData,
    __global double * rhsData, __global double * defectData,
    unsigned int width, unsigned int height, double h)
{
  int i = get_global_id(0);
  int j = get_global_id(1);
  double temp = 0.0;
  if ( (i == 0) || (j == 0) || (i == get_global_size(0)-1) ||
      (j == get_global_size(1)-1) )
  {
    //nothing
  }
  else
  {
    temp = -rhsData[i + j*width]*h*h - ( 4*matData[i + j*width
        ] - matData[i-1 + j*width] - matData[i+1 + j*width] -
        matData[i + (j+1)*width] - matData[i + (j-1)*width] );
  }
  defectData[i + j*width] = temp;
}

#endif
```

10.1.15 Interpolate kernel

```
#include <KernelHeaders.h>
__kernel void RefineCTFFF(__global float* fine, __global float
    * coarse, unsigned int corWidth)
{
  int i = get_global_id(0);
  int j = get_global_id(1);
  float tempData = 0.0;

  //load part of corGrid to shared mem:


  if ((i == 0) || (j == 0) || (i >= get_global_size(0)-1) || (j
      >= get_global_size(1)-1))
  {
    //Nothing, tempData = 0.0;
  }
  else if ( (i%2 == 0) && (j%2 == 0) )
  {
    tempData += 4*coarse[i/2 + j/2*corWidth];
  }
  else if (i%2 == 1 && j%2 == 0)
```

```
  {
    tempData += 2*coarse[i/2 + j/2*corWidth];
    tempData += 2*coarse[i/2+1 + j/2*corWidth];
  }
  else if (i%2 == 0 && j%2 == 1)
  {
    tempData += 2*coarse[i/2 + j/2*corWidth];
    tempData += 2*coarse[i/2 + j/2*corWidth+corWidth];
  }
  else if (i%2 == 1 && j%2 == 1)
  {
    tempData += coarse[i/2 + j/2*corWidth];
    tempData += coarse[i/2 + j/2*corWidth+corWidth];
    tempData += coarse[i/2+1 + j/2*corWidth];
    tempData += coarse[i/2+1 + j/2*corWidth+corWidth];
  }

  fine[i+j*(2*corWidth-1)] = tempData/4.0;
}


#ifdef __DOUBLE_ALLOWED__
__kernel void RefineCTFFD(__global double* fine, __global float* coarse, unsigned int corWidth)
{
  int i = get_global_id(0);
  int j = get_global_id(1);
  double tempData = 0.0;

  if ((i == 0) || (j == 0) || (i == get_global_size(0)-1) || (j == get_global_size(1)-1))
  {
    //Nothing, tempData = 0.0;
  }
  else if ( (i%2 == 0) && (j%2 == 0) )
  {
    tempData += 4*coarse[i/2 + j/2*corWidth];
  }
  else if (i%2 == 1 && j%2 == 0)
  {
    tempData += 2*coarse[i/2 + j/2*corWidth];
    tempData += 2*coarse[i/2+1 + j/2*corWidth];
  }
  else if (i%2 == 0 && j%2 == 1)
  {
    tempData += 2*coarse[i/2 + j/2*corWidth];
    tempData += 2*coarse[i/2 + j/2*corWidth+corWidth];
  }
  else if (i%2 == 1 && j%2 == 1)
```

```
    {
        tempData += coarse[i/2 + j/2*corWidth];
        tempData += coarse[i/2 + j/2*corWidth+corWidth];
        tempData += coarse[i/2+1 + j/2*corWidth];
        tempData += coarse[i/2+1 + j/2*corWidth+corWidth];
    }

    fine[i+j*(2*corWidth-1)] = tempData/4.0;
}

__kernel void RefineCTFDD(__global double* fine, __global double* coarse, unsigned int corWidth)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    double tempData = 0.0;

    if ((i == 0) || (j == 0) || (i == get_global_size(0)-1) || (j == get_global_size(1)-1))
    {
        //Nothing, tempData = 0.0;
    }
    else if ( (i%2 == 0) && (j%2 == 0) )
    {
        tempData += 4*coarse[i/2 + j/2*corWidth];
    }
    else if (i%2 == 1 && j%2 == 0)
    {
        tempData += 2*coarse[i/2 + j/2*corWidth];
        tempData += 2*coarse[i/2+1 + j/2*corWidth];
    }
    else if (i%2 == 0 && j%2 == 1)
    {
        tempData += 2*coarse[i/2 + j/2*corWidth];
        tempData += 2*coarse[i/2 + j/2*corWidth+corWidth];
    }
    else if (i%2 == 1 && j%2 == 1)
    {
        tempData += coarse[i/2 + j/2*corWidth];
        tempData += coarse[i/2 + j/2*corWidth+corWidth];
        tempData += coarse[i/2+1 + j/2*corWidth];
        tempData += coarse[i/2+1 + j/2*corWidth+corWidth];
    }

    fine[i+j*(2*corWidth-1)] = tempData/4.0;
}
#endif
```

10.1.16 Restriction kernel

```
#include <KernelHeaders.h>
__kernel void RefineFTCFF(__global float* fine, __global float* coarse, unsigned int fineWidth)
{
  int i = get_global_id(0);
  int j = get_global_id(1);
  float tempData = 0.0;
  if ( (i == 0) || (j == 0) || (i == get_global_size(0)-1) || (j == get_global_size(1)-1) )
  {
    //nothing
  }
  else
  {
    tempData +=     fine[2*i-1 + 2*j*fineWidth - fineWidth];
    tempData += 2 * fine[2*i   + 2*j*fineWidth - fineWidth];
    tempData +=     fine[2*i+1 + 2*j*fineWidth - fineWidth];
    tempData += 2 * fine[2*i-1 + 2*j*fineWidth];
    tempData += 4 * fine[2*i   + 2*j*fineWidth];
    tempData += 2 * fine[2*i+1 + 2*j*fineWidth];
    tempData +=     fine[2*i-1 + 2*j*fineWidth + fineWidth];
    tempData += 2 * fine[2*i   + 2*j*fineWidth + fineWidth];
    tempData +=     fine[2*i+1 + 2*j*fineWidth + fineWidth];
  }

  coarse[i+j*(fineWidth/2+1)] = tempData/16.0;
}



#ifdef __DOUBLE_ALLOWED__
__kernel void RefineFTCDF(__global double* fine, __global float* coarse, unsigned int fineWidth)
{
  int i = get_global_id(0);
  int j = get_global_id(1);
  float tempData = 0.0;
  if ( (i == 0) || (j == 0) || (i == get_global_size(0)-1) || (j == get_global_size(1)-1) )
  {
    //nothing
  }
  else
  {
    tempData +=     fine[2*i-1 + 2*j*fineWidth - fineWidth];
```

```
      tempData += 2 * fine[2*i   + 2*j*fineWidth - fineWidth];
      tempData +=     fine[2*i+1 + 2*j*fineWidth - fineWidth];
      tempData += 2 * fine[2*i-1 + 2*j*fineWidth];
      tempData += 4 * fine[2*i   + 2*j*fineWidth];
      tempData += 2 * fine[2*i+1 + 2*j*fineWidth];
      tempData +=     fine[2*i-1 + 2*j*fineWidth + fineWidth];
      tempData += 2 * fine[2*i   + 2*j*fineWidth + fineWidth];
      tempData +=     fine[2*i+1 + 2*j*fineWidth + fineWidth];
    }

    coarse[i+j*(fineWidth/2+1)] = tempData/16.0;
}

__kernel void RefineFTCDD(__global double* fine, __global
      double* coarse, unsigned int fineWidth)
{
    int i = get_global_id(0);
    int j = get_global_id(1);
    double tempData = 0.0;
    if ( (i == 0) || (j == 0) || (i == get_global_size(0)-1) ||
         (j == get_global_size(1)-1) )
    {
      //nothing
    }
    else
    {
      tempData +=     fine[2*i-1 + 2*j*fineWidth - fineWidth];
      tempData += 2 * fine[2*i   + 2*j*fineWidth - fineWidth];
      tempData +=     fine[2*i+1 + 2*j*fineWidth - fineWidth];
      tempData += 2 * fine[2*i-1 + 2*j*fineWidth];
      tempData += 4 * fine[2*i   + 2*j*fineWidth];
      tempData += 2 * fine[2*i+1 + 2*j*fineWidth];
      tempData +=     fine[2*i-1 + 2*j*fineWidth + fineWidth];
      tempData += 2 * fine[2*i   + 2*j*fineWidth + fineWidth];
      tempData +=     fine[2*i+1 + 2*j*fineWidth + fineWidth];
    }

    coarse[i+j*(fineWidth/2+1)] = tempData/16.0;
}
#endif
```

10.1.17 KernelHeaders

```
#ifdef __DOUBLE_ALLOWED__
#pragma OPENCL EXTENSION cl_khr_fp64 : enable
#endif
```

10.2 Mex Bindings

All MEX bindings are provided here:

10.2.1 MexInitOpenCL.cpp

```
#include "mex.h"
#include "OpenCLManager.h"

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  if (nrhs != 0)
  {
    mexPrintf("Failed to init OpenCL\n");
    return;
  }
  void * tempMng = mxMalloc(sizeof(OpenCLManager));
  OpenCLManager * __OpenCLManager__ = new(tempMng)
      OpenCLManager();
  mexMakeMemoryPersistent(tempMng);
  const mwSize rows = 1;
  plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL
      );
  size_t * data = (size_t *) mxGetData(plhs[0]);
  *data = (size_t)tempMng;
}
```

10.2.2 MexReleaseOpenCL.cpp

```
#include "mex.h"
#include "OpenCLManager.h"

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  if (nrhs != 1)
  {
    mexPrintf("Handle not provided");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
```

```
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
  __OpenCLManager__->ReleaseOpenCL();
  mxFree(__OpenCLManager__);
}
```

10.2.3 MexSetGPU.cpp

```
#include "mex.h"
#include "OpenCLManager.h"

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  if (nrhs != 2 && nrhs != 3)
  {
    mexPrintf("Handle not provided");
    return;
  }
  //Get OpenCL handle:
  uintptr_t * tempPtr = (uintptr_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
  //Get value:
  double * tempPtrVal = (double *)mxGetData(prhs[1]);
  double type = *tempPtrVal + 0.1;
  if (nrhs == 3)
  {
    double * tempPtrRes = (double *)mxGetData(prhs[2]);
    double res = *tempPtrRes + 0.1;
    if (((unsigned int)res) == 1)
    {
      __OpenCLManager__->AllowDouble();
    }
  }
  __OpenCLManager__->SetActiveGPU((unsigned int)type);
}
```

10.2.4 MexPrintGPU.cpp

```
#include "mex.h"
#include "OpenCLManager.h"

```

```
void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
  __OpenCLManager__->PrintGPUs();
}
```

10.2.5 MexHandleMatrix.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"

//
//Function for getting data from the GPU vectors to the cpu
    vectors.
//

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  if (nrhs != 2)
  {
    mexPrintf("array handle not provided. Input: Handle,
        Approximation, rhs\n");
    return;
  }

  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  //get vector handle:
  tempPtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempPtr[1];

  if (type == 0) //float
  {
    Matrix<float> * uh = (Matrix<float> *)((void *)tempPtr[0])
        ;

```

```
        plhs[0] = mxCreateDoubleMatrix(uh->GetWidth(),uh->
            GetHeight(),mxREAL);
        uh->SyncBuffers();
        double * data = (double *) mxGetData(plhs[0]);
        float * input = uh->GetData();
        for (unsigned int i = 0; i < uh->GetLength(); i += 1)
        {
            data[i] = (double)input[i];
        }

    }
    else if (type == 1) //double
    {
        Matrix<double> * uh = (Matrix<double> *)((void *)tempPtr
            [0]);
        uh->SyncBuffers();
        plhs[0] = mxCreateDoubleMatrix(uh->GetWidth(),uh->
            GetHeight(),mxREAL);
        double * data = (double *) mxGetData(plhs[0]);
        double * input = uh->GetData();
        for (unsigned int i = 0; i < uh->GetLength(); i += 1)
        {
            data[i] = input[i];
        }
    }
}
```

10.2.6 MexMatrix.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"

//
//Get a matrix from matlab and put it on the gpu.
//

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
    if (nrhs != 2)
    {
        mexPrintf("Handle or vector not provided");
        return;
    }
```

```
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  if (mxIsDouble(prhs[1]))
  {
    int M = mxGetM(prhs[1]);
    int N = mxGetN(prhs[1]);
    double * vecPtr = (double *)mxGetPr(prhs[1]);

    //populate gpu:
    void * tempMat = mxMalloc(sizeof(Matrix<double>));
    Matrix<double> * matrix = (Matrix<double> *)tempMat;
    matrix->InitMatrix(__OpenCLManager__,M, N, (double *)vecPtr);
    matrix->Type = 1;
    mexMakeMemoryPersistent(tempMat);


    //return handle to gpu vector:
    const mwSize rows = 2;
    plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
    size_t * data = (size_t *) mxGetData(plhs[0]);
    data[0] = (size_t)tempMat;
    data[1] = 1; //double
  }
  else if (mxIsSingle(prhs[1]))
  {
    int N = mxGetN(prhs[1]);
    int M = mxGetM(prhs[1]);

    float * vecPtr = (float*)mxGetPr(prhs[1]);

    //populate gpu:
    void * tempMat = mxMalloc(sizeof(Matrix<float>));
    Matrix<float> * matrix = (Matrix<float> *)tempMat;
    matrix->InitMatrix(__OpenCLManager__,M, N, (float *)vecPtr);
    matrix->Type = 0;
    mexMakeMemoryPersistent(tempMat);

    //return handle to gpu vector:
    const mwSize rows = 2;
    plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
    size_t * data = (size_t *) mxGetData(plhs[0]);
    data[0] = (size_t)tempMat;
    data[1] = 0; //float
```

```
  }
}
```

10.2.7 MexReleaseMatrix.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  if (nrhs != 2)
  {
    mexPrintf("matrix not provided");
    return;
  }
  //Get OpenCL handle:
  uintptr_t * handlePtr = (uintptr_t *)mxGetData(prhs[0]);
  void * temp = (void*)*handlePtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
  //get vector handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempPtr[1];
  if (type == 0) //float
  {
    Matrix<float> * mat = (Matrix<float> *)((void *)tempPtr
        [0]);
    __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
    mxFree(mat);
  }
  else if (type == 1) //double
  {
    Matrix<double> * mat = (Matrix<double> *)((void *)tempPtr
        [0]);
    __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
    mxFree(mat);
  }
}
```

10.2.8 MexBandMatrix.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "BandMatrix.h"

//
//Get a matrix from matlab and put it on the gpu.
//

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const mxArray *prhs[ ])
{
  if (nrhs != 2)
  {
    mexPrintf("Handle or vector not provided");
    return;
  }
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  if (mxIsDouble(prhs[1]))
  {
    int M = mxGetM(prhs[1]);
    int N = mxGetN(prhs[1]);
    double * vecPtr = (double *)mxGetPr(prhs[1]);
    mexPrintf("N %u, M %u\n", N, M);

    //populate gpu:
    void * tempMat = mxMalloc(sizeof(BandMatrix<double>));
    BandMatrix<double> * matrix = (BandMatrix<double> *)tempMat;
    matrix->InitMatrix(__OpenCLManager__,M, M, N/2, (double *)vecPtr);
    mexMakeMemoryPersistent(tempMat);


    //return handle to gpu vector:
    const mwSize rows = 2;
    plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
    size_t * data = (size_t *) mxGetData(plhs[0]);
    data[0] = (size_t)tempMat;
    data[1] = 1; //double
  }
  else if (mxIsSingle(prhs[1]))
  {
    int N = mxGetN(prhs[1]);
    int M = mxGetM(prhs[1]);
```

```

    float * vecPtr = (float*)mxGetPr(prhs[1]);

    //populate gpu:
    void * tempMat = mxMalloc(sizeof(BandMatrix<float>));
    BandMatrix<float> * matrix = (BandMatrix<float> *)tempMat;
    matrix->InitMatrix(__OpenCLManager__,M, N, N/2, (float *)
        vecPtr);
    mexMakeMemoryPersistent(tempMat);

    //return handle to gpu vector:
    const mwSize rows = 2;
    plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
        mxREAL);
    size_t * data = (size_t *) mxGetData(plhs[0]);
    data[0] = (size_t)tempMat;
    data[1] = 0; //float
  }
}
```

10.2.9 MexReleaseBandMatrix.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "BandMatrix.h"

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  if (nrhs != 2)
  {
    mexPrintf("matrix not provided");
    return;
  }
  //Get OpenCL handle:
  uintptr_t * handlePtr = (uintptr_t *)mxGetData(prhs[0]);
  void * temp = (void*)*handlePtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
  //get vector handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempPtr[1];
  if (type == 0) //float
  {
    BandMatrix<float> * mat = (BandMatrix<float> *)((void *)
        tempPtr[0]);
```

```
        __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
        mxFree(mat);
    }
    else if (type == 1) //double
    {
        BandMatrix<double> * mat = (BandMatrix<double> *)((void *)
            tempPtr[0]);
        __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
        mxFree(mat);
    }
}
```

10.2.10 MexSparseMatrix.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "SparseMatrix.h"

//
//Get a matrix from matlab and put it on the gpu.
//

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
    if (nrhs < 2)
    {
        mexPrintf("Handle or vector not provided");
        return;
    }
    size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
    size_t * rows = mxGetIr(prhs[1]);
    unsigned int nnz = (unsigned int)mxGetNzmax(prhs[1]);
    size_t * columns = mxGetJc(prhs[1]);
    void * temp = (void*)*tempPtr;
    int M = mxGetM(prhs[1]);
    int N = mxGetN(prhs[1]);
    OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

    bool isDouble = true;
    if (nrhs > 2)
    {
        double * isDoublePtr = (double *)mxGetData(prhs[2]);
        isDouble = (*isDoublePtr + 0.1) < 1;
    }
    if (isDouble)
```

```
  {
    double * vecPtr = (double *)mxGetPr(prhs[1]);

    //populate gpu:
    void * tempMat = mxMalloc(sizeof(SparseMatrix<double>));
    SparseMatrix<double> * matrix = (SparseMatrix<double> *)tempMat;
    matrix->InitMatrix(__OpenCLManager__,M, N, nnz, rows, columns, (double *)vecPtr);
    mexMakeMemoryPersistent(tempMat);


    //return handle to gpu vector:
    const mwSize rows = 2;
    plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS, mxREAL);
    size_t * data = (size_t *) mxGetData(plhs[0]);
    data[0] = (size_t)tempMat;
    data[1] = 1; //double
  }
  else
  {
    double * vecPtr = (double*)mxGetPr(prhs[1]);
    float * vecFloatPtr = (float *)malloc(sizeof(float)*nnz);
    for (unsigned int i = 0; i < nnz; i += 1)
    {
      vecFloatPtr[i] = (float)vecPtr[i];
    }

    //populate gpu:
    void * tempMat = mxMalloc(sizeof(SparseMatrix<float>));
    SparseMatrix<float> * matrix = (SparseMatrix<float> *)tempMat;
    matrix->InitMatrix(__OpenCLManager__,M, N, nnz, rows, columns, (float *)vecFloatPtr);
    mexMakeMemoryPersistent(tempMat);

    //return handle to gpu vector:
    const mwSize rows = 2;
    plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS, mxREAL);
    size_t * data = (size_t *) mxGetData(plhs[0]);
    data[0] = (size_t)tempMat;
    data[1] = 0; //float
  }
}
```

10.2.11 MexReleaseSparseMatrix.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "SparseMatrix.h"

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  if (nrhs != 2)
  {
    mexPrintf("matrix not provided");
    return;
  }
  //Get OpenCL handle:
  uintptr_t * handlePtr = (uintptr_t *)mxGetData(prhs[0]);
  void * temp = (void*)*handlePtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
  //get vector handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempPtr[1];
  if (type == 0) //float
  {
    SparseMatrix<float> * mat = (SparseMatrix<float> *)((void
        *)tempPtr[0]);
    __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
    __OpenCLManager__->DeleteIndex(mat->GetRowControl());
    __OpenCLManager__->DeleteIndex(mat->GetColumnControl());
    mxFree(mat);
  }
  else if (type == 1) //double
  {
    SparseMatrix<double> * mat = (SparseMatrix<double> *)((
        void *)tempPtr[0]);
    __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
    __OpenCLManager__->DeleteIndex(mat->GetRowControl());
    __OpenCLManager__->DeleteIndex(mat->GetColumnControl());
    mxFree(mat);
  }
}
```

10.2.12 MexBandMatrixVector.cpp

```
#include "mex.h"
```

```
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"
#include "BandMatrix.h"

//
//Function for using jacobi and gauss-seidel smoothers.
//Arguments:
//0: OpenCL handle.
//1: uh
//2: RHS
//3: h (optional).
//
void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
     mxArray *prhs[ ])
{
  float time;
  cl_event event;
  double h = -1.0;
  if (nrhs < 2 || nrhs > 6)
  {
    mexPrintf("Handle or arrays not provided. Input: Handle,
        Approximation, rhs\n");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  //Check if RPT and TPG are provided:
  if (nrhs >= 6)
  {
    size_t RowsPerThread = (size_t)(*mxGetPr(prhs[4])+0.1);
    size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[5])+0.1);
    __OpenCLManager__->SetBandMatrixVectorRowsPerThread(
        RowsPerThread);
    __OpenCLManager__->SetBandMatrixVectorThreadsPerGroup(
        ThreadsPerGroup);
  }


  //get vector handle:
  size_t * tempBandPtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempBandPtr[1];

  size_t * tempVecPtr = (size_t *)mxGetData(prhs[2]);
  size_t typeRHS = tempVecPtr[1];

```

```
if (type == 0 && typeRHS == 0) //float
{
  BandMatrix<float> * band = (BandMatrix<float> *)((void *)
      tempBandPtr[0]);
  Matrix<float> * vec = (Matrix<float> *)((void *)tempVecPtr
      [0]);

  Matrix<float> * out = vec;
  if (nrhs ≥ 3)
  {
    size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
    out = (Matrix<float> *)((void *)tempOutPtr[0]);
  }
    else if (nlhs == 1) //need to create a matrix to send
        out:
    {
       //populate gpu:
      out = (Matrix<float> * )mxMalloc(sizeof(Matrix<float>)
          );
    out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec
        ->GetHeight(), (float*)NULL);
       mexMakeMemoryPersistent(out);

       //return handle to gpu vector:
       const mwSize rows = 2;
       plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS
           ,mxREAL);
       size_t * data = (size_t *)mxGetData(plhs[0]);
       data[0] = (size_t)out;
       data[1] = 0; //float
    }

  __OpenCLManager__->BandMatrixVectorFF(band->GetDataBuffer
      (), vec->GetDataBuffer(), out->GetDataBuffer(), band->
      GetHeight(), band->GetBandwidth(), band->GetLength(),
      &event);
}
else if (type == 1 && typeRHS == 1) //doubles
{
  BandMatrix<double> * band = (BandMatrix<double> *)((void
      *)tempBandPtr[0]);
  Matrix<double> * vec = (Matrix<double> *)((void *)
      tempVecPtr[0]);

  Matrix<double> * out = vec;
  if (nrhs ≥ 3)
  {
    size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
```

```
        out = (Matrix<double> *)((void *)tempOutPtr[0]);
    }
        else if (nlhs == 1) //need to create a matrix to send
            out:
        {
          //populate gpu:
            out = (Matrix<double> * )mxMalloc(sizeof(Matrix<
                double>));
        out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec
            ->GetHeight(), (double*)NULL);
            mexMakeMemoryPersistent(out);

            //return handle to gpu vector:
            const mwSize rows = 2;
            plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS
                ,mxREAL);
            size_t * data = (size_t *)mxGetData(plhs[0]);
            data[0] = (size_t)out;
            data[1] = 1; //double
        }

    __OpenCLManager__->BandMatrixVectorDD(band->GetDataBuffer
        (), vec->GetDataBuffer(), out->GetDataBuffer(), band->
        GetHeight(), band->GetBandwidth(), vec->GetLength(), &
        event);
  }

  //Outgoing matrix provided and we return time instead.
  if (nlhs == 1 && nrhs >= 3)
  {
    __OpenCLManager__->WaitForCPU();
    float time = __OpenCLManager__->GetExecutionTime(&event);
      //return handle to gpu vector:
      const mwSize rows = 1;
      plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,
          mxREAL);
      double * data = (double *)mxGetData(plhs[0]);
      *data = (double)time;
  }
  clReleaseEvent(event);
}
```

10.2.13 MexSparseMatrixVector.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
```

```c
#include "MathVector.h"
#include "Matrix.h"
#include "SparseMatrix.h"

//
//Function for using jacobi and gauss-seidel smoothers.
//Arguments:
//0: OpenCL handle.
//1: uh
//2: RHS
//3: h (optional).
//
void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const mxArray *prhs[ ])
{
  double h = -1.0;
  if (nrhs < 3 || nrhs > 7)
  {
    mexPrintf("Handle or arrays not provided. Input: Handle, Approximation, rhs\n");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  //get vector handle:
  size_t * tempSparsePtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempSparsePtr[1];

  size_t * tempVecPtr = (size_t *)mxGetData(prhs[2]);
  size_t typeRHS = tempVecPtr[1];

  if (nrhs >= 6)
  {
    size_t RowsPerThread = (size_t)(*mxGetPr(prhs[4])+0.1);
    size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[5])+0.1);
    __OpenCLManager__->SetSparseMatrixVectorRowsPerThread(RowsPerThread);
    __OpenCLManager__->SetSparseMatrixVectorThreadsPerGroup(ThreadsPerGroup);
  }


  cl_event event;
  if (type == 0 && typeRHS == 0) //float
  {
```

```
    SparseMatrix<float> * sparse = (SparseMatrix<float> *)((void *)tempSparsePtr[0]);
    Matrix<float> * vec = (Matrix<float> *)((void *)tempVecPtr[0]);

    //populate gpu:
    Matrix<float> * out;
    if (nrhs>= 4)
    {
      size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
      out = (Matrix<float> *)((void *)tempOutPtr[0]);
    }
    else
    {
        out = (Matrix<float> * )mxMalloc(sizeof(Matrix<float>));
      out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec->GetHeight(), (float*)NULL);
        mexMakeMemoryPersistent(out);
      }

    __OpenCLManager__->SparseMatrixVectorFF(sparse->GetDataBuffer(), sparse->GetColumnIndexBuffer(), sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out->GetDataBuffer(), sparse->GetHeight(), sparse->GetWidth(), sparse->GetLength(), sparse->GetRowVectorLength(), &event);
    __OpenCLManager__->WaitForCPU();

    if (nrhs<4)
    {
        if (nlhs == 1) //need to create a matrix to send out:
        {
          //return handle to gpu vector:
           const mwSize rows = 2;
           plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
           size_t * data = (size_t *)mxGetData(plhs[0]);
           data[0] = (size_t)out;
           data[1] = 0; //float
        }
        else
        {
          __OpenCLManager__->DeleteMemory(vec->GetMemoryControl());
            vec->SetMemoryControl(out->GetMemoryControl());
          mxFree(out);
        }
      }
```

```
        else if (nlhs == 1) //send out timing:
        {
          //return handle to gpu vector:
           const mwSize rows = 1;
           plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS
               ,mxREAL);
           double * data = (double *)mxGetData(plhs[0]);
           __OpenCLManager__->WaitForCPU();
        float time = __OpenCLManager__->GetExecutionTime(&event)
            ;
           data[0] = time;
            }

    }
    else if (type == 1 && typeRHS == 1) //doubles
    {
      SparseMatrix<double> * sparse = (SparseMatrix<double> *)((
          void *)tempSparsePtr[0]);
      Matrix<double> * vec = (Matrix<double> *)((void *)
          tempVecPtr[0]);

      Matrix<double> * out;
      if (nrhs>= 4)
      {
        size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
        out = (Matrix<double> *)((void *)tempOutPtr[0]);
      }
      else
      {
          out = (Matrix<double> * )mxMalloc(sizeof(Matrix<double
              >));
        out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec
            ->GetHeight(), (double*)NULL);
          mexMakeMemoryPersistent(out);
        }

      __OpenCLManager__->SparseMatrixVectorDD(sparse->
          GetDataBuffer(), sparse->GetColumnIndexBuffer(),
          sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out
          ->GetDataBuffer(), sparse->GetHeight(), sparse->
          GetWidth(), sparse->GetLength(), sparse->
          GetRowVectorLength(), &event);
      __OpenCLManager__->WaitForCPU();
      if (nrhs<4)
      {
          if (nlhs == 1) //need to create a matrix to send out:
            {
              //return handle to gpu vector:
              const mwSize rows = 2;
```

```
                plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
                size_t * data = (size_t *)mxGetData(plhs[0]);
                data[0] = (size_t)out;
                data[1] = 1; //double
            }
            else
            {
                //Clean up:
                __OpenCLManager__->DeleteMemory(vec->GetMemoryControl());
                vec->SetMemoryControl(out->GetMemoryControl());
                mxFree(out);
            }
        }
        else if (nlhs == 1) //send out timing:
          {
            //return handle to gpu vector:
             const mwSize rows = 1;
             plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,mxREAL);
             double * data = (double *)mxGetData(plhs[0]);
          float time = __OpenCLManager__->GetExecutionTime(&event);
             data[0] = time;
              }

    }
    else if (type == 1 && typeRHS == 0) //double sparse and float vector given. Return should be double.
    {
        SparseMatrix<double> * sparse = (SparseMatrix<double> *)((void *)tempSparsePtr[0]);
        Matrix<float> * vec = (Matrix<float> *)((void *)tempVecPtr[0]);

        Matrix<double> * out;
        if (nrhs >= 4)
        {
            size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
            out = (Matrix<double> *)((void *)tempOutPtr[0]);
        }
        else
        {
              out = (Matrix<double> * )mxMalloc(sizeof(Matrix<double>));
            out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec->GetHeight(), (double*)NULL);
              mexMakeMemoryPersistent(out);
```

```
        }

      __OpenCLManager__->SparseMatrixVectorDF(sparse->GetDataBuffer(), sparse->GetColumnIndexBuffer(), sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out->GetDataBuffer(), sparse->GetHeight(), sparse->GetWidth(), sparse->GetLength(), sparse->GetRowVectorLength(), &event);
      __OpenCLManager__->WaitForCPU();
      if (nrhs<4)
      {
          if (nlhs == 1) //need to create a matrix to send out:
            {
              //return handle to gpu vector:
              const mwSize rows = 2;
              plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
              size_t * data = (size_t *)mxGetData(plhs[0]);
              data[0] = (size_t)out;
              data[1] = 1; //double
            }
      }
      else if (nlhs == 1) //send out timing:
        {
          //return handle to gpu vector:
           const mwSize rows = 1;
           plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,mxREAL);
           double * data = (double *)mxGetData(plhs[0]);
        float time = __OpenCLManager__->GetExecutionTime(&event);
           data[0] = time;
            }

    }
    else if (type == 0 && typeRHS == 1) //float sparse and double vector given. Return should be float.
    {
      SparseMatrix<float> * sparse = (SparseMatrix<float> *)((void *)tempSparsePtr[0]);
      Matrix<double> * vec = (Matrix<double> *)((void *)tempVecPtr[0]);

      Matrix<float> * out;
      if (nrhs>= 4)
      {
        size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
        out = (Matrix<float> *)((void *)tempOutPtr[0]);
      }
```

```
    else
    {
        out = (Matrix<float> * )mxMalloc(sizeof(Matrix<float>));
      out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec->GetHeight(), (float*)NULL);
        mexMakeMemoryPersistent(out);
      }

    __OpenCLManager__->SparseMatrixVectorFD(sparse->GetDataBuffer(), sparse->GetColumnIndexBuffer(), sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out->GetDataBuffer(), sparse->GetHeight(), sparse->GetWidth(), sparse->GetLength(), sparse->GetRowVectorLength(), &event);
    __OpenCLManager__->WaitForCPU();
    if (nrhs<4)
    {
        if (nlhs == 1) //need to create a matrix to send out:
          {
            //return handle to gpu vector:
            const mwSize rows = 2;
            plhs[0] = mxCreateNumericArray(1,&rows, mxUINT64_CLASS,mxREAL);
            size_t * data = (size_t *)mxGetData(plhs[0]);
            data[0] = (size_t)out;
            data[1] = 0; //double
          }
    }
    else if (nlhs == 1) //send out timing:
      {
        //return handle to gpu vector:
          const mwSize rows = 1;
          plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,mxREAL);
          double * data = (double *)mxGetData(plhs[0]);

        float time = __OpenCLManager__->GetExecutionTime(&event);
          data[0] = time;
          }

  }
  clReleaseEvent(event);
}
```

10.2.14 MexSparseMatrixVector.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"
#include "SparseMatrix.h"

//
//Function for using jacobi and gauss-seidel smoothers.
//Arguments:
//0: OpenCL handle.
//1: uh
//2: RHS
//3: h (optional).
//
void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const mxArray *prhs[ ])
{
  double h = -1.0;
  if (nrhs < 3 || nrhs > 7)
  {
    mexPrintf("Handle or arrays not provided. Input: Handle, Approximation, rhs\n");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  //get vector handle:
  size_t * tempSparsePtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempSparsePtr[1];

  size_t * tempVecPtr = (size_t *)mxGetData(prhs[2]);
  size_t typeRHS = tempVecPtr[1];

  if (nrhs ≥ 6)
  {
    size_t RowsPerThread = (size_t)(*mxGetPr(prhs[4])+0.1);
    size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[5])+0.1);
    __OpenCLManager__->SetSparseMatrixVectorRowsPerThread(RowsPerThread);
    __OpenCLManager__->SetSparseMatrixVectorThreadsPerGroup(ThreadsPerGroup);
  }


  cl_event event;
```

```
if (type == 0 && typeRHS == 0) //float
{
  SparseMatrix<float> * sparse = (SparseMatrix<float> *)((void *)tempSparsePtr[0]);
  Matrix<float> * vec = (Matrix<float> *)((void *)tempVecPtr[0]);

  //populate gpu:
  Matrix<float> * out;
  if (nrhs>= 4)
  {
    size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
    out = (Matrix<float> *)((void *)tempOutPtr[0]);
  }
  else
  {
      out = (Matrix<float> * )mxMalloc(sizeof(Matrix<float>));
    out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec->GetHeight(), (float*)NULL);
      mexMakeMemoryPersistent(out);
    }

  __OpenCLManager__->SparseMatrixVectorFF(sparse->GetDataBuffer(), sparse->GetColumnIndexBuffer(), sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out->GetDataBuffer(), sparse->GetHeight(), sparse->GetWidth(), sparse->GetLength(), sparse->GetRowVectorLength(), &event);
  __OpenCLManager__->WaitForCPU();

  if (nrhs<4)
  {
      if (nlhs == 1) //need to create a matrix to send out:
      {
        //return handle to gpu vector:
         const mwSize rows = 2;
         plhs[0] = mxCreateNumericArray(1,&rows, mxUINT64_CLASS,mxREAL);
         size_t * data = (size_t *)mxGetData(plhs[0]);
         data[0] = (size_t)out;
         data[1] = 0; //float
      }
      else
      {
        __OpenCLManager__->DeleteMemory(vec->GetMemoryControl());
          vec->SetMemoryControl(out->GetMemoryControl());
        mxFree(out);
```

```
          }
        }
        else if (nlhs == 1) //send out timing:
        {
          //return handle to gpu vector:
           const mwSize rows = 1;
           plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,mxREAL);
           double * data = (double *)mxGetData(plhs[0]);
           __OpenCLManager__->WaitForCPU();
        float time = __OpenCLManager__->GetExecutionTime(&event);
           data[0] = time;
            }

  }
  else if (type == 1 && typeRHS == 1) //doubles
  {
    SparseMatrix<double> * sparse = (SparseMatrix<double> *)((void *)tempSparsePtr[0]);
    Matrix<double> * vec = (Matrix<double> *)((void *)tempVecPtr[0]);

    Matrix<double> * out;
    if (nrhs>= 4)
    {
      size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
      out = (Matrix<double> *)((void *)tempOutPtr[0]);
    }
    else
    {
        out = (Matrix<double> * )mxMalloc(sizeof(Matrix<double>));
      out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec->GetHeight(), (double*)NULL);
        mexMakeMemoryPersistent(out);
      }

    __OpenCLManager__->SparseMatrixVectorDD(sparse->GetDataBuffer(), sparse->GetColumnIndexBuffer(), sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out->GetDataBuffer(), sparse->GetHeight(), sparse->GetWidth(), sparse->GetLength(), sparse->GetRowVectorLength(), &event);
    __OpenCLManager__->WaitForCPU();
    if (nrhs<4)
    {
        if (nlhs == 1) //need to create a matrix to send out:
          {
```

```
            //return handle to gpu vector:
            const mwSize rows = 2;
            plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
            size_t * data = (size_t *)mxGetData(plhs[0]);
            data[0] = (size_t)out;
            data[1] = 1; //double
          }
          else
          {
            //Clean up:
            __OpenCLManager__->DeleteMemory(vec->GetMemoryControl());
            vec->SetMemoryControl(out->GetMemoryControl());
            mxFree(out);
          }
    }
    else if (nlhs == 1) //send out timing:
      {
        //return handle to gpu vector:
         const mwSize rows = 1;
         plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,mxREAL);
         double * data = (double *)mxGetData(plhs[0]);
      float time = __OpenCLManager__->GetExecutionTime(&event);
         data[0] = time;
          }

  }
  else if (type == 1 && typeRHS == 0) //double sparse and float vector given. Return should be double.
  {
    SparseMatrix<double> * sparse = (SparseMatrix<double> *)((void *)tempSparsePtr[0]);
    Matrix<float> * vec = (Matrix<float> *)((void *)tempVecPtr[0]);

    Matrix<double> * out;
    if (nrhs>= 4)
    {
      size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
      out = (Matrix<double> *)((void *)tempOutPtr[0]);
    }
    else
    {
        out = (Matrix<double> * )mxMalloc(sizeof(Matrix<double>));
```

```
        out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec->GetHeight(), (double*)NULL);
          mexMakeMemoryPersistent(out);
        }

      __OpenCLManager__->SparseMatrixVectorDF(sparse->GetDataBuffer(), sparse->GetColumnIndexBuffer(), sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out->GetDataBuffer(), sparse->GetHeight(), sparse->GetWidth(), sparse->GetLength(), sparse->GetRowVectorLength(), &event);
      __OpenCLManager__->WaitForCPU();
      if (nrhs<4)
      {
          if (nlhs == 1) //need to create a matrix to send out:
            {
              //return handle to gpu vector:
              const mwSize rows = 2;
              plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
              size_t * data = (size_t *)mxGetData(plhs[0]);
              data[0] = (size_t)out;
              data[1] = 1; //double
            }
      }
      else if (nlhs == 1) //send out timing:
        {
          //return handle to gpu vector:
           const mwSize rows = 1;
           plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,mxREAL);
           double * data = (double *)mxGetData(plhs[0]);
        float time = __OpenCLManager__->GetExecutionTime(&event);
           data[0] = time;
            }

    }
    else if (type == 0 && typeRHS == 1) //float sparse and double vector given. Return should be float.
    {
      SparseMatrix<float> * sparse = (SparseMatrix<float> *)((void *)tempSparsePtr[0]);
      Matrix<double> * vec = (Matrix<double> *)((void *)tempVecPtr[0]);

      Matrix<float> * out;
      if (nrhs>= 4)
      {
```

```
        size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
        out = (Matrix<float> *)((void *)tempOutPtr[0]);
      }
      else
      {
          out = (Matrix<float> * )mxMalloc(sizeof(Matrix<float>)
              );
        out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec
            ->GetHeight(), (float*)NULL);
          mexMakeMemoryPersistent(out);
        }

      __OpenCLManager__->SparseMatrixVectorFD(sparse->
          GetDataBuffer(), sparse->GetColumnIndexBuffer(),
          sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out
          ->GetDataBuffer(), sparse->GetHeight(), sparse->
          GetWidth(), sparse->GetLength(), sparse->
          GetRowVectorLength(), &event);
      __OpenCLManager__->WaitForCPU();
      if (nrhs<4)
      {
          if (nlhs == 1) //need to create a matrix to send out:
            {
              //return handle to gpu vector:
              const mwSize rows = 2;
              plhs[0] = mxCreateNumericArray(1,&rows,
                  mxUINT64_CLASS,mxREAL);
              size_t * data = (size_t *)mxGetData(plhs[0]);
              data[0] = (size_t)out;
              data[1] = 0; //double
            }
      }
      else if (nlhs == 1) //send out timing:
        {
          //return handle to gpu vector:
            const mwSize rows = 1;
            plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS
                ,mxREAL);
            double * data = (double *)mxGetData(plhs[0]);

        float time = __OpenCLManager__->GetExecutionTime(&event)
            ;
            data[0] = time;
            }

  }
  clReleaseEvent(event);
}
```

10.2.15 MexVectorMinusVector.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"

//Not done, needs DD and FD

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  cl_event event;
  if (nrhs < 3 || nrhs > 7)
  {
    mexPrintf("Handle or arrays not provided. Input: Handle,
        vector1, vector2\n");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  //get vector1 handle:
  size_t * tempPtr1 = (size_t *)mxGetData(prhs[1]);
  size_t type1 = tempPtr1[1];

  //get vector2 handle:
  size_t * tempPtr2 = (size_t *)mxGetData(prhs[2]);
  size_t type2 = tempPtr2[1];

  if (nrhs >= 6)
  {
    size_t RowsPerThread = (size_t)(*mxGetPr(prhs[4])+0.1);
    size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[5])+0.1);
    __OpenCLManager__->SetVectorAndVectorRowsPerThread(
        RowsPerThread);
    __OpenCLManager__->SetVectorAndVectorThreadsPerGroup(
        ThreadsPerGroup);
  }

  if (type1 == 0 && type2 == 0) //float & float
  {
    Matrix<float> * vec1 = (Matrix<float> *)((void *)tempPtr1
        [0]);
```

```
    Matrix<float> * vec2 = (Matrix<float> *)((void *)tempPtr2
        [0]);
    Matrix<float> * out = vec1;
    if (nrhs > 3)
    {
      size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
      out = (Matrix<float> *)((void *)tempOutPtr[0]);
    }
    else if (nlhs == 1) //need to create a matrix to send out:
    {
      //populate gpu:
      void * tempMat = mxMalloc(sizeof(Matrix<float>));
      out = new(tempMat) Matrix<float>(__OpenCLManager__, vec1
          ->GetWidth(), vec1->GetHeight());
      mexMakeMemoryPersistent(tempMat);

      //return handle to gpu vector:
      const mwSize rows = 2;
      plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
          mxREAL);
      size_t * data = (size_t *)mxGetData(plhs[0]);
      data[0] = (size_t)tempMat;
      data[1] = 0; //float
    }
    __OpenCLManager__->VectorMinusVectorFF(vec1->GetDataBuffer
        (), vec2->GetDataBuffer(), out->GetDataBuffer(), vec1
        ->GetLength(), &event);
  }
  else if (type1 == 1 && type2 == 0) //double & float
  {
    Matrix<double> * vec1 = (Matrix<double> *)((void *)
        tempPtr1[0]);
    Matrix<float> * vec2 = (Matrix<float> *)((void *)tempPtr2
        [0]);
    Matrix<double> * out = vec1;
    if (nrhs > 3)
    {
      size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
      out = (Matrix<double> *)((void *)tempOutPtr[0]);
    }
    else if (nlhs == 1) //need to create a matrix to send out:
    {
      //populate gpu:
      void * tempMat = mxMalloc(sizeof(Matrix<double>));
      out = new(tempMat) Matrix<double>(__OpenCLManager__,
          vec1->GetWidth(), vec1->GetHeight());
      mexMakeMemoryPersistent(tempMat);

      //return handle to gpu vector:
```

```
      const mwSize rows = 2;
      plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
      size_t * data = (size_t *)mxGetData(plhs[0]);
      data[0] = (size_t)tempMat;
      data[1] = 1; //double
    }
    __OpenCLManager__->VectorMinusVectorDF(vec1->GetDataBuffer(), vec2->GetDataBuffer(), out->GetDataBuffer(), vec1->GetLength(), &event);
  }
  else if (type1 == 1 && type2 == 1) //double & double
  {
    Matrix<double> * vec1 = (Matrix<double> *)((void *)tempPtr1[0]);
    Matrix<double> * vec2 = (Matrix<double> *)((void *)tempPtr2[0]);
    Matrix<double> * out = vec1;

    if (nrhs > 3)
    {
      size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
      out = (Matrix<double> *)((void *)tempOutPtr[0]);
    }
    else if (nlhs == 1) //need to create a matrix to send out:
    {
      //populate gpu:
      void * tempMat = mxMalloc(sizeof(Matrix<double>));
      out = new(tempMat) Matrix<double>(__OpenCLManager__, vec1->GetWidth(), vec1->GetHeight());
      mexMakeMemoryPersistent(tempMat);

      //return handle to gpu vector:
      const mwSize rows = 2;
      plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
      size_t * data = (size_t *)mxGetData(plhs[0]);
      data[0] = (size_t)tempMat;
      data[1] = 1; //double
    }
    __OpenCLManager__->VectorMinusVectorDD(vec1->GetDataBuffer(), vec2->GetDataBuffer(), out->GetDataBuffer(), vec1->GetLength(), &event);
  }
  __OpenCLManager__->WaitForCPU();
  //if an outgoing matrix is provided already:
  if (nlhs >= 1 && nrhs >= 4)
  {
    float time = __OpenCLManager__->GetExecutionTime(&event);
```

```
    const mwSize rows = 1;
    plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,
        mxREAL);
    float * data = (float *) mxGetData(plhs[0]);
    *data = (float)time;
  }
  clReleaseEvent(event);
}
```

10.2.16 MexVectorMinusVectorConstant.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"

//Not done, needs DD and FD

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  cl_event event;
  if (nrhs < 4 || nrhs > 7)
  {
    mexPrintf("Handle or arrays not provided. Input: Handle,
        vector1, vector2\n");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  //get vector1 handle:
  size_t * tempPtr1 = (size_t *)mxGetData(prhs[1]);
  size_t type1 = tempPtr1[1];

  //get vector2 handle:
  size_t * tempPtr2 = (size_t *)mxGetData(prhs[2]);
  size_t type2 = tempPtr2[1];

  //get contant:
  double * tempConst = (double *)mxGetData(prhs[3]);
  double con = *tempConst;

  if (nrhs >= 7)
```

```
{
  size_t RowsPerThread = (size_t)(*mxGetPr(prhs[5])+0.1);
  size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[6])+0.1);
  __OpenCLManager__->SetVectorAndVectorRowsPerThread(RowsPerThread);
  __OpenCLManager__->SetVectorAndVectorThreadsPerGroup(ThreadsPerGroup);
}

if (type1 == 0 && type2 == 0) //float & float
{
  Matrix<float> * vec1 = (Matrix<float> *)((void *)tempPtr1[0]);
  Matrix<float> * vec2 = (Matrix<float> *)((void *)tempPtr2[0]);
  Matrix<float> * out = vec1;
  if (nrhs > 3)
  {
    size_t * tempOutPtr = (size_t *)mxGetData(prhs[4]);
    out = (Matrix<float> *)((void *)tempOutPtr[0]);
  }
  else if (nlhs == 1) //need to create a matrix to send out:
  {
    //populate gpu:
    void * tempMat = mxMalloc(sizeof(Matrix<float>));
    out = new(tempMat) Matrix<float>(__OpenCLManager__, vec1->GetWidth(), vec1->GetHeight());
    mexMakeMemoryPersistent(tempMat);

    //return handle to gpu vector:
    const mwSize rows = 2;
    plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
    size_t * data = (size_t *)mxGetData(plhs[0]);
    data[0] = (size_t)tempMat;
    data[1] = 0; //float
  }
  __OpenCLManager__->VectorMinusVectorConstantFF(vec1->GetDataBuffer(), vec2->GetDataBuffer(), out->GetDataBuffer(), (float) con, vec1->GetLength(), &event);
}
else if (type1 == 1 && type2 == 0) //double & float
{
  Matrix<double> * vec1 = (Matrix<double> *)((void *)tempPtr1[0]);
  Matrix<float> * vec2 = (Matrix<float> *)((void *)tempPtr2[0]);
  Matrix<double> * out = vec1;
```

```
    if (nrhs > 3)
    {
      size_t * tempOutPtr = (size_t *)mxGetData(prhs[4]);
      out = (Matrix<double> *)((void *)tempOutPtr[0]);
    }
    else if (nlhs == 1) //need to create a matrix to send out:
    {
      //populate gpu:
      void * tempMat = mxMalloc(sizeof(Matrix<double>));
      out = new(tempMat) Matrix<double>(__OpenCLManager__,
          vec1->GetWidth(), vec1->GetHeight());
      mexMakeMemoryPersistent(tempMat);

      //return handle to gpu vector:
      const mwSize rows = 2;
      plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
          mxREAL);
      size_t * data = (size_t *)mxGetData(plhs[0]);
      data[0] = (size_t)tempMat;
      data[1] = 1; //double
    }
    __OpenCLManager__->VectorMinusVectorConstantDF(vec1->
        GetDataBuffer(), vec2->GetDataBuffer(), out->
        GetDataBuffer(), con, vec1->GetLength(), &event);
  }
  else  if (type1 == 1 && type2 == 1) //double & double
  {
    Matrix<double> * vec1 = (Matrix<double> *)((void *)
        tempPtr1[0]);
    Matrix<double> * vec2 = (Matrix<double> *)((void *)
        tempPtr2[0]);
    Matrix<double> * out = vec1;

    if (nrhs > 3)
    {
      size_t * tempOutPtr = (size_t *)mxGetData(prhs[4]);
      out = (Matrix<double> *)((void *)tempOutPtr[0]);
    }
    else if (nlhs == 1) //need to create a matrix to send out:
    {
      //populate gpu:
      void * tempMat = mxMalloc(sizeof(Matrix<double>));
      out = new(tempMat) Matrix<double>(__OpenCLManager__,
          vec1->GetWidth(), vec1->GetHeight());
      mexMakeMemoryPersistent(tempMat);

      //return handle to gpu vector:
      const mwSize rows = 2;
```

```
        plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS, mxREAL);
        size_t * data = (size_t *)mxGetData(plhs[0]);
        data[0] = (size_t)tempMat;
        data[1] = 1; //double
      }
      __OpenCLManager__->VectorMinusVectorConstantDD(vec1->GetDataBuffer(), vec2->GetDataBuffer(), out->GetDataBuffer(), con, vec1->GetLength(), &event);
    }

    //if an outgoing matrix is provided already:
    __OpenCLManager__->WaitForCPU();
    if (nlhs >= 1 && nrhs >= 5)
    {
      float time = __OpenCLManager__->GetExecutionTime(&event);
      const mwSize rows = 1;
      plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS, mxREAL);
      float * data = (float *) mxGetData(plhs[0]);
      *data = (float)time;
    }
    clReleaseEvent(event);
}
```

10.2.17 MexVectorTimesConstant.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"

//Not done, needs DD and FD

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const mxArray *prhs[ ])
{
    cl_event event;
      if (nrhs > 7 || nrhs < 3)
      {
          mexPrintf("Handle or arrays not provided. Input: Handle, matrix, element\n");
          return;
      }
      //get opencl handle:
      size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
```

```
    void * temp = (void*)*tempPtr;
    OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  if (nrhs ≥ 6)
  {
    size_t RowsPerThread = (size_t)(*mxGetPr(prhs[4])+0.1);
    size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[5])+0.1);
    __OpenCLManager__->SetVectorConstantRowsPerThread(RowsPerThread);
    __OpenCLManager__->SetVectorConstantThreadsPerGroup(ThreadsPerGroup);
  }

    //get vector handle:
    tempPtr = (size_t *)mxGetData(prhs[1]);
    size_t type = tempPtr[1];

    double * tempPtrEl = (double *)mxGetData(prhs[2]);
    double element = *tempPtrEl;
    if (type == 0) //float
    {
        Matrix<float> * uh = (Matrix<float> *)((void *)tempPtr[0]);
        Matrix<float> * out = uh;

          if (nrhs > 3)
      {
        size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
        out = (Matrix<float> *)((void *)tempOutPtr[0]);
      }
        else if (nlhs == 1) //need to create a matrix to send out:
        {
            //populate gpu:
            void * tempMat = mxMalloc(sizeof(Matrix<float>));
            Matrix<float> * tempGPUvec = new(tempMat) Matrix<float>(__OpenCLManager__, uh->GetWidth(), uh->GetHeight());
            mexMakeMemoryPersistent(tempMat);
            out = tempGPUvec;

            //return handle to gpu vector:
            const mwSize rows = 2;
            plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
            size_t * data = (size_t *)mxGetData(plhs[0]);
            data[0] = (size_t)tempMat;
            data[1] = 0; //float
        }
```

```
            __OpenCLManager__->VectorTimesConstantFF(uh->GetDataBuffer(), out->GetDataBuffer(), (float)element, uh->GetLength(), &event);
    }
    else if (type == 1) //double
    {
        Matrix<double> * uh = (Matrix<double> *)((void *)tempPtr[0]);
        Matrix<double> * out = uh;

          if (nrhs >   3)
      {
        size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
        out = (Matrix<double> *)((void *)tempOutPtr[0]);
      }
        else if (nlhs == 1) //need to create a matrix to send out:
        {
            //populate gpu:
            void * tempMat = mxMalloc(sizeof(Matrix<double>));
            Matrix<double> * tempGPUvec = new(tempMat) Matrix<double>(__OpenCLManager__, uh->GetWidth(), uh->GetHeight());
            mexMakeMemoryPersistent(tempMat);
            out = tempGPUvec;

            //return handle to gpu vector:
            const mwSize rows = 2;
            plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL);
            size_t * data = (size_t *)mxGetData(plhs[0]);
            data[0] = (size_t)tempMat;
            data[1] = 1; //double
        }
        __OpenCLManager__->VectorTimesConstantDD(uh->GetDataBuffer(), out->GetDataBuffer(), (double)element, uh->GetLength(), &event);
    }

  //if an outgoing matrix is provided already:
  __OpenCLManager__->WaitForCPU();
  if (nlhs >= 1 && nrhs >= 4)
  {
    float time = __OpenCLManager__->GetExecutionTime(&event);
    const mwSize rows = 1;
    plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,mxREAL);
    float * data = (float *) mxGetData(plhs[0]);
    *data = (float)time;
```

```
  }
  clReleaseEvent(event);

}
```

10.2.18 MexSum.cpp

```
#include "mex.h"
#include "OpenCLHigh.h"

//
//Norm Inf:
//Arguments:
//0: OpenCL handle
//1: Matrix/vector to operate on.
//
void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  if (nrhs != 2 && nrhs != 3)
  {
    mexPrintf("MexSum: Handle or array not provided\n");
    return;
  }
  //timing:
  float time;

  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  //get vector handle:
  tempPtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempPtr[1];

  //get optional size handle:
  unsigned int size = 256;
  if (nrhs >= 3)
  {
    double * tempSizePtr = (double *)mxGetData(prhs[2]);
    size = (unsigned int)(*tempSizePtr + 0.1);
  }
  if (type == 0) //float
  {
```

```
    MathVector<float> * mat = (MathVector<float> *)((void *)
        tempPtr[0]);
    float tempsum = sum(__OpenCLManager__, *mat, size, &time);

    //return sum:
    const mwSize rows = 1;
    plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,
        mxREAL);
    float * data = (float *) mxGetData(plhs[0]);
    *data = (float)tempsum;
  }
  else if (type == 1) //double
  {
    MathVector<double> * mat = (MathVector<double> *)((void *)
        tempPtr[0]);
    double tempsum = sum(__OpenCLManager__, *mat, size, &time)
        ;

    //return sum:
    const mwSize rows = 1;
    plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,
        mxREAL);
    double * data = (double *) mxGetData(plhs[0]);
    *data = (double)tempsum;
  }
  __OpenCLManager__->WaitForCPU();
  if (nlhs > 1)
  {
    const mwSize rows = 1;
    plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,
        mxREAL);
    float * data = (float *) mxGetData(plhs[0]);
    *data = (float)time;
  }


}
```

10.2.19 MexNorm2.cpp

```
#include "mex.h"
#include "OpenCLHigh.h"

//
//Norm Inf:
//Arguments:
```

```
//0: OpenCL handle
//1: Matrix/vector to operate on.
//
void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const mxArray *prhs[ ])
{
  float time;
  if (nrhs != 2 && nrhs != 3)
  {
    mexPrintf("MexNorm2: Handle or array not provided\n");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  //get vector handle:
  tempPtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempPtr[1];

  unsigned int size = 256;
  if (nrhs ≥ 3)
  {
    double * tempSizePtr = (double *)mxGetData(prhs[2]);
    size = (unsigned int)(*tempSizePtr + 0.1);
  }
  if (type == 0) //float
  {
    MathVector<float> * mat = (MathVector<float> *)((void *)tempPtr[0]);
    float tempsum = Norm2(__OpenCLManager__, *mat, size, &time);

    //return sum:
    const mwSize rows = 1;
    plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,mxREAL);
    float * data = (float *) mxGetData(plhs[0]);
    *data = (float)tempsum;
  }
  else if (type == 1) //double
  {
    MathVector<double> * mat = (MathVector<double> *)((void *)tempPtr[0]);;
    double tempsum = Norm2(__OpenCLManager__, *mat, size, &time);

    //return sum:
```

```
    const mwSize rows = 1;
    plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,
        mxREAL);
    double * data = (double *) mxGetData(plhs[0]);
    *data = (double)tempsum;
  }
  __OpenCLManager__->WaitForCPU();
  if (nlhs > 1)
  {
    const mwSize rows = 1;
    plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,
        mxREAL);
    float * data = (float *) mxGetData(plhs[0]);
    *data = (float)time;
  }


}
```

10.2.20 MexNormInf.cpp

```
#include "mex.h"
#include "OpenCLHigh.h"

//
//Norm Inf:
//Arguments:
//0: OpenCL handle
//1: Matrix/vector to operate on.
//
void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  float time;
  if (nrhs != 2 && nrhs != 3)
  {
    mexPrintf("MexNormInf: Handle or array not provided\n");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  //get vector handle:
  tempPtr = (size_t *)mxGetData(prhs[1]);
```

```
    size_t type = tempPtr[1];

    //Get optional size handle:
    unsigned int size = 256;
    if (nrhs >= 3)
    {
      double * tempSizePtr = (double *)mxGetData(prhs[2]);
      size = (unsigned int)(*tempSizePtr + 0.1);
    }
    if (type == 0) //float
    {
      MathVector<float> * mat = (MathVector<float> *)((void *)
          tempPtr[0]);
      float tempsum = NormInf(__OpenCLManager__, *mat, size, &
          time);

      //return sum:
      const mwSize rows = 1;
      plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,
          mxREAL);
      float * data = (float *) mxGetData(plhs[0]);
      *data = (float)tempsum;
    }
    else if (type == 1) //double
    {
      MathVector<double> * mat = (MathVector<double> *)((void *)
          tempPtr[0]);;
      double tempsum = NormInf(__OpenCLManager__, *mat, size, &
          time);

      //return sum:
      const mwSize rows = 1;
      plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,
          mxREAL);
      double * data = (double *) mxGetData(plhs[0]);
      *data = (double)tempsum;
    }
    __OpenCLManager__->WaitForCPU();
    if (nlhs > 1)
    {
      const mwSize rows = 1;
      plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,
          mxREAL);
      float * data = (float *) mxGetData(plhs[0]);
      *data = (float)time;
    }

}
```

10.2.21 MexCoarseToFine.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"

//
//Function for turning a fine grid into a coarser grid.
//Arguments:
//0: OpenCL handle
//1: Fine matrix handle
//2: type of returned matrix (0: float, 1: double)
//output:
//0: Coarse matrix handle
//
void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  cl_event event;
  if (nrhs < 3)
  {
    mexPrintf("Handle or arrays not provided. Input: Handle,
        Approximation, rhs\n");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  __OpenCLManager__->SetCTFRowsPerThread(64);
  __OpenCLManager__->SetCTFThreadsPerGroup(64);

  //get matrix handle:
  tempPtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempPtr[1];

  //Determine return type:
  size_t * tempFinePtr = (size_t *)mxGetData(prhs[2]);
  size_t typeFineType = tempFinePtr[1];

  if (type == 0 && typeFineType == 0) //float and return float
      .
  {
    Matrix<float> * coarse = (Matrix<float> *)((void *)tempPtr
        [0]);
```

```
    Matrix<float> * fine = (Matrix<float> *)((void *)
        tempFinePtr[0]);
    __OpenCLManager__->CoarseToFineFF(fine->GetDataBuffer(),
        coarse->GetDataBuffer(), fine->GetWidth(), fine->
        GetHeight(), &event);
  }
  else if (type == 1 && typeFineType == 1) //double
  {
    Matrix<double> * coarse = (Matrix<double> *)((void *)
        tempPtr[0]);
    Matrix<double> * fine = (Matrix<double> *)((void *)
        tempFinePtr[0]);
    __OpenCLManager__->CoarseToFineDD(fine->GetDataBuffer(),
        coarse->GetDataBuffer(), fine->GetWidth(), fine->
        GetHeight(), &event);
  }
  else if (type == 0 && typeFineType == 1) //float to double
  {
    Matrix<float> * coarse = (Matrix<float> *)((void *)tempPtr
        [0]);
    Matrix<double> * fine = (Matrix<double> *)((void *)
        tempFinePtr[0]);
    __OpenCLManager__->CoarseToFineFD(fine->GetDataBuffer(),
        coarse->GetDataBuffer(), fine->GetWidth(), fine->
        GetHeight(), &event);
  }

  if (nlhs == 1) //send out timing:
   {
    //return handle to gpu vector:
      const mwSize rows = 1;
      plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,
          mxREAL);
      double * data = (double *)mxGetData(plhs[0]);
      __OpenCLManager__->WaitForCPU();
    float time = __OpenCLManager__->GetExecutionTime(&event);
      data[0] = time;
   }

  clReleaseEvent(event);
}
```

10.2.22 MexFineToCoarse.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
```

```
#include "MathVector.h"
#include "Matrix.h"

//
//Function for turning a fine grid into a coarser grid.
//Arguments:
//0: OpenCL handle
//1: Fine matrix handle
//2: type of returned matrix (0: float, 1: double)
//output:
//0: Coarse matrix handle
//
void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  cl_event event;
  if (nrhs < 3)
  {
    mexPrintf("Handle or arrays not provided. Input: Handle,
        Approximation, rhs\n");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  __OpenCLManager__->SetFTCRowsPerThread(64);
  __OpenCLManager__->SetFTCThreadsPerGroup(64);

  //get matrix handle:
  tempPtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempPtr[1];

  //get matrix handle:
  size_t * tempCorPtr = (size_t *)mxGetData(prhs[2]);
  size_t typeCorType = tempCorPtr[1];

  if (type == 0 && typeCorType == 0) //float and return float.
  {
    Matrix<float> * fine = (Matrix<float> *)((void *)tempPtr
        [0]);
    Matrix<float> * coarse = (Matrix<float> *)((void *)
        tempCorPtr[0]);
    __OpenCLManager__->FineToCoarseFF(fine->GetDataBuffer(),
        coarse->GetDataBuffer(), coarse->GetWidth(), coarse->
        GetHeight(), &event);
  }
  else if (type == 1 && typeCorType == 1) //double
```

```
  {
    Matrix<double> * fine = (Matrix<double> *)((void *)tempPtr[0]);
    Matrix<double> * coarse = (Matrix<double> *)((void *)tempCorPtr[0]);
    __OpenCLManager__->FineToCoarseDD(fine->GetDataBuffer(), coarse->GetDataBuffer(), coarse->GetWidth(), coarse->GetHeight(), &event);
  }
  else if (type == 1 && typeCorType == 0) //double to float
  {
    Matrix<double> * fine = (Matrix<double> *)((void *)tempPtr[0]);
    Matrix<float> * coarse = (Matrix<float> *)((void *)tempCorPtr[0]);
    __OpenCLManager__->FineToCoarseDF(fine->GetDataBuffer(), coarse->GetDataBuffer(), coarse->GetWidth(), coarse->GetHeight(), &event);
  }

  if (nlhs == 1) //send out timing:
   {
    //return handle to gpu vector:
      const mwSize rows = 1;
      plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS, mxREAL);
      double * data = (double *)mxGetData(plhs[0]);
      __OpenCLManager__->WaitForCPU();
    float time = __OpenCLManager__->GetExecutionTime(&event);
      data[0] = time;
   }

  clReleaseEvent(event);
}
```

10.2.23 MexRBGS.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"

//
//Function for using jacobi and gauss-seidel smoothers.
//Arguments:
//0: OpenCL handle.
```

```
//1: uh
//2: RHS
//3: h (optional).
//
void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const mxArray *prhs[ ])
{
  cl_event event;
  double h = -1.0;
  if (nrhs < 3 || nrhs > 8)
  {
    mexPrintf("Handle or arrays not provided. Input: Handle, Approximation, rhs\n");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  //get vector handle:
  tempPtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempPtr[1];

  size_t * tempPtrRHS = (size_t *)mxGetData(prhs[2]);
  size_t typeRHS = tempPtrRHS[1];

  __OpenCLManager__->SetJacobiRowsPerThread(8);
  __OpenCLManager__->SetJacobiThreadsPerGroup(8);

  //Get h if given:
  if (nrhs >= 4)
  {
    double * tempHPtr = (double *)mxGetData(prhs[3]);
    h = *tempHPtr;
  }

  //if nmax is given:
  unsigned int nmax = 1;
  if (nrhs >= 5)
  {
    double * tempNmaxPtr = (double*)mxGetData(prhs[4]);
    nmax = (unsigned int)(*tempNmaxPtr + 0.1);
  }

  unsigned int grid = 8;
  if (nrhs >= 7)
  {
    double * tempRPTPtr = (double*)mxGetData(prhs[6]);
```

```
    size_t RPT = (size_t)(*tempRPTPtr + 0.1);
    double * tempTPGPtr = (double*)mxGetData(prhs[7]);
    size_t TPG = (size_t)(*tempTPGPtr + 0.1);
    __OpenCLManager__->SetJacobiThreadsPerGroup(TPG);
    __OpenCLManager__->SetJacobiRowsPerThread(RPT);
  }

  if (type == 0 && typeRHS == 0) //float
  {
    Matrix<float> * uh = (Matrix<float> *)((void *)tempPtr[0]);
    Matrix<float> * rhs = (Matrix<float> *)((void *)tempPtrRHS[0]);
    Matrix<float> * output;
    if (nrhs>= 6) //swap matrix given.
    {
      size_t * tempOutPtr = (size_t *)mxGetData(prhs[5]);
      output = (Matrix<float> *)((void *)tempOutPtr[0]);
    }
    else
    {
        output = (Matrix<float> * )mxMalloc(sizeof(Matrix<float>));
      output->InitMatrix(__OpenCLManager__, uh->GetWidth(), uh->GetHeight(), (float*)NULL);
        //mexMakeMemoryPersistent(output);
    }
    if (h < 0) //h not given:
    {
      h = 1.0/(uh->GetWidth()-1);
    }
    Matrix<float> * pOut = output, * pIn = uh, * pTemp;
    for (unsigned int i = 0; i < nmax; i++)
    {
      __OpenCLManager__->JacobiF(pOut->GetDataBuffer(), pIn->GetDataBuffer(), rhs->GetDataBuffer(), uh->GetWidth(), uh->GetHeight(), (float)h, grid, &event);
      pTemp = pOut;
      pOut = pIn;
      pIn = pTemp;
    }
    //Make sure we keep the right one:
    __MemoryControl__<float> * memOut = pIn->GetMemoryControl(), * memIn = pOut->GetMemoryControl();
    uh->SetMemoryControl(memOut);
    output->SetMemoryControl(memIn);
    //delete output;
  }
  else if (type == 1 && typeRHS == 1) //double
```

```
{
    Matrix<double> * uh = (Matrix<double> *)((void *)tempPtr[0]);
    Matrix<double> * rhs = (Matrix<double> *)((void *)tempPtrRHS[0]);
    Matrix<double> * output;
    if (nrhs>= 6) //swap matrix given.
    {
      size_t * tempOutPtr = (size_t *)mxGetData(prhs[5]);
      output = (Matrix<double> *)((void *)tempOutPtr[0]);
    }
    else
    {
        output = (Matrix<double> * )mxMalloc(sizeof(Matrix<double>));
      output->InitMatrix(__OpenCLManager__, uh->GetWidth(), uh->GetHeight(), (double*)NULL);
        //mexMakeMemoryPersistent(output);
    }
    if (h < 0) //h not given:
    {
      h = 1.0/(uh->GetWidth()-1);
    }
    Matrix<double> * pOut = output, * pIn = uh, * pTemp;
    for (unsigned int i = 0; i < nmax; i++)
    {
      __OpenCLManager__->JacobiD(pOut->GetDataBuffer(), pIn->GetDataBuffer(), rhs->GetDataBuffer(), uh->GetWidth(), uh->GetHeight(), (double)h, grid, &event);
      pTemp = pOut;
      pOut = pIn;
      pIn = pTemp;
    }
    //Make sure we keep the right one:
    __MemoryControl__<double> * memOut = pIn->GetMemoryControl(), * memIn = pOut->GetMemoryControl();
    uh->SetMemoryControl(memOut);
    output->SetMemoryControl(memIn);
    //delete output;
}
    if (nlhs == 1) //send out timing:
{
    //return handle to gpu vector:
      const mwSize rows = 1;
      plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS, mxREAL);
    double * data = (double *)mxGetData(plhs[0]);
    __OpenCLManager__->WaitForCPU();
    float time = __OpenCLManager__->GetExecutionTime(&event);
```

```
            data[0] = time;
        }
    clReleaseEvent(event);
}
```

10.2.24 MexJacobi.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"

//
//Function for using jacobi and gauss-seidel smoothers.
//Arguments:
//0: OpenCL handle.
//1: uh
//2: RHS
//3: h (optional).
//
void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const mxArray *prhs[ ])
{
    cl_event event1;
    cl_event event2;
    double h = -1.0;
    if (nrhs < 3 || nrhs > 7)
    {
        mexPrintf("Handle or arrays not provided. Input: Handle, Approximation, rhs\n");
        return;
    }
    //get opencl handle:
    size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
    void * temp = (void*)*tempPtr;
    OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

    //get vector handle:
    tempPtr = (size_t *)mxGetData(prhs[1]);
    size_t type = tempPtr[1];

    size_t * tempPtrRHS = (size_t *)mxGetData(prhs[2]);
    size_t typeRHS = tempPtrRHS[1];

    //Get h if given:
    if (nrhs >= 4)
```

```
{
  double * tempHPtr = (double *)mxGetData(prhs[3]);
  h = *tempHPtr;
}

//if nmax is given:
unsigned int nmax = 1;
if (nrhs ≥ 5)
{
  double * tempNmaxPtr = (double*)mxGetData(prhs[4]);
  nmax = (unsigned int)(*tempNmaxPtr + 0.1);
}
unsigned int grid = 8;
if (nrhs ≥ 7)
{
  double * tempRPTPtr = (double*)mxGetData(prhs[5]);
  size_t RPT = (size_t)(*tempRPTPtr + 0.1);
  double * tempTPGPtr = (double*)mxGetData(prhs[6]);
  size_t TPG = (size_t)(*tempTPGPtr + 0.1);
  __OpenCLManager__->SetRBGSThreadsPerGroup(TPG);
  __OpenCLManager__->SetRBGSRowsPerThread(RPT);
}

if (type == 0 && typeRHS == 0) //float
{
  Matrix<float> * uh = (Matrix<float> *)((void *)tempPtr[0])
      ;
  Matrix<float> * rhs = (Matrix<float> *)((void *)tempPtrRHS
      [0]);
  if (h < 0) //h not given:
  {
    h = 1.0/(uh->GetWidth()-1);
  }
  for (unsigned int i = 0; i < nmax; i++)
  {
    __OpenCLManager__->JacobiMethodOddF(uh->GetDataBuffer(),
        rhs->GetDataBuffer(), uh->GetWidth(), uh->GetHeight
        (), (float)h, grid, &event1);
    __OpenCLManager__->WaitForCPU();
    __OpenCLManager__->JacobiMethodEvenF(uh->GetDataBuffer()
        , rhs->GetDataBuffer(), rhs->GetWidth(), rhs->
        GetHeight(), (float)h, grid, &event2);
  }
}
else if (type == 1 && typeRHS == 1) //double
{
  Matrix<double> * uh = (Matrix<double> *)((void *)tempPtr
      [0]);
```

```
    Matrix<double> * rhs = (Matrix<double> *)((void *)
        tempPtrRHS[0]);
    if (h < 0) //h not given:
    {
      h = 1.0/(uh->GetWidth()-1);
    }
    for (unsigned int i = 0; i < nmax; i++)
    {
      __OpenCLManager__->JacobiMethodOddD(uh->GetDataBuffer(),
          rhs->GetDataBuffer(), uh->GetWidth(), uh->GetHeight
          (), (double)h, grid, &event1);
      __OpenCLManager__->WaitForCPU();
      __OpenCLManager__->JacobiMethodEvenD(uh->GetDataBuffer()
          , rhs->GetDataBuffer(), uh->GetWidth(), uh->
          GetHeight(), (double)h, grid, &event2);
    }
  }
  if (nlhs == 1) //send out timing:
   {
    //return handle to gpu vector:
      const mwSize rows = 1;
      plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,
          mxREAL);
      double * data = (double *)mxGetData(plhs[0]);
      __OpenCLManager__->WaitForCPU();
    float time = __OpenCLManager__->GetExecutionTime(&event1)
        + __OpenCLManager__->GetExecutionTime(&event2);
      data[0] = time;
   }
  clReleaseEvent(event1);
  clReleaseEvent(event2);
}
```

10.2.25 MexPoissonDefect.cpp

```
#include "mex.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include "Matrix.h"

void mexFunction(int nlhs, mxArray *plhs[ ],int nrhs, const
    mxArray *prhs[ ])
{
  cl_event event;
  double h = -1.0;
  if (nrhs < 3 || nrhs > 8)
```

```
  {
    mexPrintf("Handle or arrays not provided. Input: Handle, Approximation, rhs\n");
    return;
  }
  //get opencl handle:
  size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
  void * temp = (void*)*tempPtr;
  OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;

  //get uh handle:
  tempPtr = (size_t *)mxGetData(prhs[1]);
  size_t type = tempPtr[1];

  //Get RHS handle:
  size_t * tempPtrRHS = (size_t *)mxGetData(prhs[2]);
  size_t typeRHS = tempPtrRHS[1];

    //Get h if given:
  if (nrhs >= 4)
  {
    double * tempHPtr = (double *)mxGetData(prhs[3]);
    h = *tempHPtr;
  }
  if (nrhs >= 7)
  {
    double * tempRPTPtr = (double*)mxGetData(prhs[5]);
    size_t RPT = (size_t)(*tempRPTPtr + 0.1);
    double * tempTPGPtr = (double*)mxGetData(prhs[6]);
    size_t TPG = (size_t)(*tempTPGPtr + 0.1);
    __OpenCLManager__->SetDefectThreadsPerGroup(TPG);
    __OpenCLManager__->SetDefectRowsPerThread(RPT);
  }

  if (type == 0 && typeRHS == 0) //float
  {
    Matrix<float> * uh = (Matrix<float> *)((void *)tempPtr[0]);
    Matrix<float> * rhs = (Matrix<float> *)((void *)tempPtrRHS[0]);
    Matrix<float> * defect;
    if (nlhs == 1 && nrhs < 5) //need to create a matrix to send out:
    {
      //populate gpu:
      defect = (Matrix<float> * )mxMalloc(sizeof(Matrix<float>));
      defect->InitMatrix(__OpenCLManager__, uh->GetWidth(), uh->GetHeight(), (float*)NULL);
```

```
            mexMakeMemoryPersistent(defect);

            //return handle to gpu vector:
            const mwSize rows = 2;
            plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
                mxREAL);
            size_t * data = (size_t *)mxGetData(plhs[0]);
            data[0] = (size_t)defect;
            data[1] = 0; //float
        }
        else //Matrix provided:
        {
            size_t * tempPtrDef = (size_t *)mxGetData(prhs[4]);
            defect = (Matrix<float> *)((void *)tempPtrDef[0]);
        }
        if (h < 0) //h not given:
        {
            h = 1.0/(uh->GetWidth()-1);
        }
        __OpenCLManager__->JacobiDefectF(uh->GetDataBuffer(), rhs
            ->GetDataBuffer(), defect->GetDataBuffer(), uh->
            GetWidth(), uh->GetHeight(), (float)h, &event);
    }
    else if (type == 1 && typeRHS == 1) //double
    {
        Matrix<double> * uh = (Matrix<double> *)((void *)tempPtr
            [0]);
        Matrix<double> * rhs = (Matrix<double> *)((void *)
            tempPtrRHS[0]);
        Matrix<double> * defect;
        if (nlhs == 1 && nrhs < 5) //need to create a matrix to
            send out:
        {
            //populate gpu:
            defect = (Matrix<double> * )mxMalloc(sizeof(Matrix<
                double>));
            defect->InitMatrix(__OpenCLManager__, uh->GetWidth(), uh
                ->GetHeight(), (double*)NULL);
            mexMakeMemoryPersistent(defect);

            //return handle to gpu vector:
            const mwSize rows = 2;
            plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
                mxREAL);
            size_t * data = (size_t *)mxGetData(plhs[0]);
            data[0] = (size_t)defect;
            data[1] = 1; //double
        }
        else //Matrix provided:
```

```
      {
        size_t * tempPtrDef = (size_t *)mxGetData(prhs[4]);
        defect = (Matrix<double> *)((void *)tempPtrDef[0]);
      }
      if (h < 0) //h not given:
      {
        h = 1.0/(uh->GetWidth()-1);
      }
      __OpenCLManager__->JacobiDefectD(uh->GetDataBuffer(), rhs->GetDataBuffer(), defect->GetDataBuffer(), uh->GetWidth(), uh->GetHeight(), (double)h, &event);
    }
      if (nlhs == 1 && nrhs > 5) //send out timing:
    {
      //return handle to gpu vector:
        const mwSize rows = 1;
        plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS, mxREAL);
      double * data = (double *)mxGetData(plhs[0]);
      __OpenCLManager__->WaitForCPU();
      float time = __OpenCLManager__->GetExecutionTime(&event);
        data[0] = time;
     }
    clReleaseEvent(event);
}
```

10.3 API code

The non-kernel code of the API can be found here.

10.3.1 OpenCLManager.h

```
#include "MemoryControl.h"
#include "preprocessor.h"
#include <CL/cl.h>

#ifndef __OPENCLMANAGER___
#define __OPENCLMANAGER___
class OpenCLManager
{
public:
    OpenCLManager();
    ¬OpenCLManager();
    void ReleaseOpenCL();
    void PrintGPUs();
```

```
void SetActiveGPU(unsigned int index);
void AddSource( char * name );
void AllowDouble();

//Memory management:
__MemoryControl__<float> * AllocateMemory(float * real,
    unsigned int size);
__MemoryControl__<double> * AllocateMemory(double * real,
    unsigned int size);
__IndexControl__ * AllocateIndex(unsigned int * index,
    unsigned int size);

//vector standard operations:
void VectorTimesConstantD(cl_kernel kernel, cl_mem & vector,
     cl_mem & output, double constant, unsigned int
    vectorSize, cl_event * event);
void VectorTimesConstantFF(cl_mem & vector, cl_mem & output,
     float constant, unsigned int vectorSize, cl_event *
    event);
void VectorTimesConstantFD(cl_mem & vector, cl_mem & output,
     double constant, unsigned int vectorSize, cl_event *
    event);
void VectorTimesConstantDD(cl_mem & vector, cl_mem & output,
     double constant, unsigned int vectorSize, cl_event *
    event);

void VectorOperatorVector(cl_kernel kernel, cl_mem & vector1
    , cl_mem & vector2, cl_mem & output, unsigned int length
    , cl_event * event);
void VectorMinusVectorFF(cl_mem & vector1, cl_mem & vector2,
     cl_mem & output, unsigned int length, cl_event * event)
    ;
void VectorMinusVectorFD(cl_mem & vector1, cl_mem & vector2,
     cl_mem & output, unsigned int length, cl_event * event)
    ;
void VectorMinusVectorDF(cl_mem & vector1, cl_mem & vector2,
     cl_mem & output, unsigned int length, cl_event * event)
    ;
void VectorMinusVectorDD(cl_mem & vector1, cl_mem & vector2,
     cl_mem & output, unsigned int length, cl_event * event)
    ;
void VectorPlusVectorFF(cl_mem & vector1, cl_mem & vector2,
    cl_mem & output, unsigned int length, cl_event * event);
void VectorPlusVectorFD(cl_mem & vector1, cl_mem & vector2,
    cl_mem & output, unsigned int length, cl_event * event);
void VectorPlusVectorDF(cl_mem & vector1, cl_mem & vector2,
    cl_mem & output, unsigned int length, cl_event * event);
void VectorPlusVectorDD(cl_mem & vector1, cl_mem & vector2,
    cl_mem & output, unsigned int length, cl_event * event);
```

```

void VectorOperatorVectorConstant(cl_kernel kernel, cl_mem &
    vector1, cl_mem & vector2, cl_mem & output, double con,
    unsigned int length, cl_event * event);
void VectorMinusVectorConstantFF(cl_mem & vector1, cl_mem &
    vector2, cl_mem & output, float con, unsigned int length
    , cl_event * event);
void VectorMinusVectorConstantFD(cl_mem & vector1, cl_mem &
    vector2, cl_mem & output, double con, unsigned int
    length, cl_event * event);
void VectorMinusVectorConstantDF(cl_mem & vector1, cl_mem &
    vector2, cl_mem & output, double con, unsigned int
    length, cl_event * event);
void VectorMinusVectorConstantDD(cl_mem & vector1, cl_mem &
    vector2, cl_mem & output, double con, unsigned int
    length, cl_event * event);


//sum operations:
void Norm(cl_kernel kernel, cl_mem & input, cl_mem & output,
    unsigned int threadsize, unsigned int problemsize,
    cl_event * event);
void ParallelSumReductionF(cl_mem & input, cl_mem & output,
    unsigned int threadsize, unsigned int problemsize,
    cl_event * event);
void ParallelSumReductionD(cl_mem & input, cl_mem & output,
    unsigned int threadsize, unsigned int problemsize,
    cl_event * event);
void Norm2F(cl_mem & input, cl_mem & output, unsigned int
    threadsize, unsigned int problemsize, cl_event * event);
void Norm2D(cl_mem & input, cl_mem & output, unsigned int
    threadsize, unsigned int problemsize, cl_event * event);
void NormInfF(cl_mem & input, cl_mem & output, unsigned int
    threadsize, unsigned int problemsize, cl_event * event);
void NormInfD(cl_mem & input, cl_mem & output, unsigned int
    threadsize, unsigned int problemsize, cl_event * event);

//Matrix vector operations:
void SparseMatrixVector(cl_kernel kernel, cl_mem & matData,
    cl_mem & matCol, cl_mem & matRow, cl_mem & vecData,
    cl_mem & returnData, unsigned int height, unsigned int
    width,                  unsigned int numIndexes,
    unsigned int rowVectorLength, cl_event * event);
void SparseMatrixVectorDF(cl_mem & matData, cl_mem & matCol,
    cl_mem & matRow, cl_mem & vecData, cl_mem & returnData,
    unsigned int height, unsigned int width,
        unsigned int numIndexes, unsigned int
    rowVectorLength, cl_event * event);
```

```
void SparseMatrixVectorFF(cl_mem & matData, cl_mem & matCol, cl_mem & matRow, cl_mem & vecData, cl_mem & returnData, unsigned int height, unsigned int width, unsigned int numIndexes, unsigned int rowVectorLength, cl_event * event);
void SparseMatrixVectorFD(cl_mem & matData, cl_mem & matCol, cl_mem & matRow, cl_mem & vecData, cl_mem & returnData, unsigned int height, unsigned int width, unsigned int numIndexes, unsigned int rowVectorLength, cl_event * event);
void SparseMatrixVectorDD(cl_mem & matData, cl_mem & matCol, cl_mem & matRow, cl_mem & vecData, cl_mem & returnData, unsigned int height, unsigned int width, unsigned int numIndexes, unsigned int rowVectorLength, cl_event * event);
void BandMatrixVector(cl_kernel kernel, cl_mem & matData, cl_mem & vecData, cl_mem & returnData, unsigned int height, unsigned int bandwidth,
                      unsigned int length, cl_event * event);
void BandMatrixVectorFF(cl_mem & matData, cl_mem & vecData, cl_mem & returnData, unsigned int height, unsigned int bandwidth,
                      unsigned int length, cl_event * event);
void BandMatrixVectorFD(cl_mem & matData, cl_mem & vecData, cl_mem & returnData, unsigned int height, unsigned int bandwidth,
                      unsigned int length, cl_event * event);
void BandMatrixVectorDF(cl_mem & matData, cl_mem & vecData, cl_mem & returnData, unsigned int height, unsigned int bandwidth,
                      unsigned int length, cl_event * event);
void BandMatrixVectorDD(cl_mem & matData, cl_mem & vecData, cl_mem & returnData, unsigned int height, unsigned int bandwidth,
                      unsigned int length, cl_event * event);

//Coarse-Fine operations:
void FineToCoarse(cl_kernel kernel, cl_mem & fineData, cl_mem & corData, unsigned int corWidth, unsigned int corHeight, cl_event * event);
void FineToCoarseFF(cl_mem & fineData, cl_mem & corData, unsigned int corWidth, unsigned int corHeight, cl_event * event);
```

```
void FineToCoarseDF(cl_mem & fineData, cl_mem & corData, unsigned int corWidth, unsigned int corHeight, cl_event * event);
void FineToCoarseDD(cl_mem & fineData, cl_mem & corData, unsigned int corWidth, unsigned int corHeight, cl_event * event);
void CoarseToFine(cl_kernel kernel, cl_mem & fineData, cl_mem & corData, unsigned int fineWidth, unsigned int fineHeight, cl_event * event);
void CoarseToFineFF(cl_mem & fineData, cl_mem & corData, unsigned int fineWidth, unsigned int fineHeight, cl_event * event);
void CoarseToFineFD(cl_mem & fineData, cl_mem & corData, unsigned int fineWidth, unsigned int fineHeight, cl_event * event);
void CoarseToFineDD(cl_mem & fineData, cl_mem & corData, unsigned int fineWidth, unsigned int fineHeight, cl_event * event);

//Jacobi method:
void JacobiD(cl_mem & output, cl_mem & input, cl_mem & rightData, unsigned int width, unsigned int height, double spacing, unsigned int grid, cl_event * event);
void JacobiF(cl_mem & output, cl_mem & input, cl_mem & rightData, unsigned int width, unsigned int height, float spacing, unsigned int grid, cl_event * event);
void JacobiMethodF(cl_kernel kernel, cl_mem & leftData, cl_mem & rightData, unsigned int width, unsigned int height, float spacing, unsigned int grid, cl_event * event);
void JacobiMethodD(cl_kernel kernel, cl_mem & leftData, cl_mem & rightData, unsigned int width, unsigned int height, double spacing, unsigned int grid, cl_event * event);
void JacobiMethodOddF(cl_mem & leftData, cl_mem & rightData, unsigned int width, unsigned int height, float spacing, unsigned int grid, cl_event * event);
void JacobiMethodOddD(cl_mem & leftData, cl_mem & rightData, unsigned int width, unsigned int height, double spacing, unsigned int grid, cl_event * event);
void JacobiMethodEvenF(cl_mem & leftData, cl_mem & rightData, unsigned int width, unsigned int height, float spacing, unsigned int grid, cl_event * event);
void JacobiMethodEvenD(cl_mem & leftData, cl_mem & rightData, unsigned int width, unsigned int height, double spacing, unsigned int grid, cl_event * event);
void JacobiDefectF(cl_mem & leftData, cl_mem & rightData, cl_mem & defect, unsigned int width, unsigned int height, float spacing, cl_event * event);
```

```
void JacobiDefectD(cl_mem & leftData, cl_mem & rightData,
    cl_mem & defect, unsigned int width, unsigned int height
    , double spacing, cl_event * event);

//Memory swapping:
void SwapGPUBufferData(const cl_mem & buffer, void * ptr,
    unsigned int size, size_t sizeType);
void WriteGPUBufferData(const cl_mem & buffer, void * ptr,
    unsigned int size, size_t sizeType);

//Memory resizing:
void ResizeGPUBuffer( __MemoryControl__<float> * control,
    unsigned int size);
void ResizeGPUBuffer( __MemoryControl__<double> * control,
    unsigned int size);
void ResizeGPUBuffer( __IndexControl__ * control, unsigned
    int size);

//Memory leak control:
void DeleteMemory(__MemoryControl__<float> * mem);
void DeleteMemory(__MemoryControl__<double> * mem);
void DeleteIndex(__IndexControl__ * mem);


//Autotuning:
void SetSparseMatrixVectorRowsPerThread(size_t);
void SetSparseMatrixVectorThreadsPerGroup(size_t);
void SetBandMatrixVectorRowsPerThread(size_t);
void SetBandMatrixVectorThreadsPerGroup(size_t);
void SetNormRowsPerThread(size_t);
void SetNormThreadsPerGroup(size_t);
void SetVectorAndVectorRowsPerThread(size_t);
void SetVectorAndVectorThreadsPerGroup(size_t);
void SetVectorConstantRowsPerThread(size_t);
void SetVectorConstantThreadsPerGroup(size_t);
void SetJacobiRowsPerThread(size_t);
void SetJacobiThreadsPerGroup(size_t);
void SetRBGSRowsPerThread(size_t);
void SetRBGSThreadsPerGroup(size_t);
void SetDefectRowsPerThread(size_t);
void SetDefectThreadsPerGroup(size_t);
void SetFTCRowsPerThread(size_t);
void SetFTCThreadsPerGroup(size_t);
void SetCTFRowsPerThread(size_t);
void SetCTFThreadsPerGroup(size_t);
void WaitForCPU();
float GetExecutionTime(cl_event * event);
```

```
private:
  //Shortcut functions:
  cl_kernel CreateKernel(char * name);
  //Platform and Device control:
  cl_platform_id * vectorPlatforms;
  unsigned int numPlatforms;
  cl_device_id ** vectorDevices;
  unsigned int * numDevices;
  cl_platform_id platform;
  cl_device_id device;

  //Program control:
  cl_program program;
  cl_context context;
  char ** vectorSourceFiles;
  unsigned int numSourceFiles;
  cl_command_queue queue;
  bool EnableDouble;
  char ** program_strings;
  size_t * program_sizes;

  //Autotuning constants:
  size_t SparseMatrixVectorRowsPerThread;
  size_t SparseMatrixVectorThreadsPerGroup;
  size_t BandMatrixVectorRowsPerThread;
  size_t BandMatrixVectorThreadsPerGroup;
  size_t NormRowsPerThread;
  size_t NormThreadsPerGroup;
  size_t VectorAndVectorRowsPerThread;
  size_t VectorAndVectorThreadsPerGroup;
  size_t VectorConstantRowsPerThread;
  size_t VectorConstantThreadsPerGroup;
  size_t JacobiRowsPerThread;
  size_t JacobiThreadsPerGroup;
  size_t RBGSRowsPerThread;
  size_t RBGSThreadsPerGroup;
  size_t DefectRowsPerThread;
  size_t DefectThreadsPerGroup;
  size_t FTCRowsPerThread;
  size_t FTCThreadsPerGroup;
  size_t CTFRowsPerThread;
  size_t CTFThreadsPerGroup;



  //Memory Control:
  __MemoryControl__<float> ** vectorMemoryF;
  __MemoryControl__<double> ** vectorMemoryD;
  unsigned int numMemoryF;
```

```
unsigned int numMemoryD;
unsigned int capMemoryF;
unsigned int capMemoryD;
__IndexControl__ ** vectorIndex;
unsigned int numIndex;
unsigned int capIndex;

//Kernels:
cl_kernel kernelReductionF;
cl_kernel kernelReductionD;
cl_kernel kernelSparseMatrixVectorFF;
cl_kernel kernelSparseMatrixVectorDF;
cl_kernel kernelSparseMatrixVectorDD;
cl_kernel kernelSparseMatrixVectorFD;
cl_kernel kernelBandMatrixVectorFF;
cl_kernel kernelBandMatrixVectorFD;
cl_kernel kernelBandMatrixVectorDF;
cl_kernel kernelBandMatrixVectorDD;
cl_kernel kernelJacobiMethodOddF;
cl_kernel kernelJacobiMethodOddD;
cl_kernel kernelJacobiMethodEvenF;
cl_kernel kernelJacobiMethodEvenD;
cl_kernel kernelJacobiDefectF;
cl_kernel kernelJacobiDefectD;
cl_kernel kernelJacobiF;
cl_kernel kernelJacobiD;
cl_kernel kernelRefineCTFFF;
cl_kernel kernelRefineCTFFD;
cl_kernel kernelRefineCTFDD;
cl_kernel kernelRefineFTCFF;
cl_kernel kernelRefineFTCDF;
cl_kernel kernelRefineFTCDD;
cl_kernel kernelVectorTimesConstantFF;
cl_kernel kernelVectorTimesConstantFD;
cl_kernel kernelVectorTimesConstantDD;
cl_kernel kernelVectorPlusVectorFF;
cl_kernel kernelVectorPlusVectorFD;
cl_kernel kernelVectorPlusVectorDF;
cl_kernel kernelVectorPlusVectorDD;
cl_kernel kernelVectorMinusVectorFF;
cl_kernel kernelVectorMinusVectorFD;
cl_kernel kernelVectorMinusVectorDF;
cl_kernel kernelVectorMinusVectorDD;
cl_kernel kernelVectorMinusVectorConstantFF;
cl_kernel kernelVectorMinusVectorConstantFD;
cl_kernel kernelVectorMinusVectorConstantDF;
cl_kernel kernelVectorMinusVectorConstantDD;
cl_kernel kernelNormInfF;
cl_kernel kernelNormInfD;
```

```
    cl_kernel kernelNorm2F;
    cl_kernel kernelNorm2D;

    //conditional kernel statements:
    bool NVIDIA;

    //functions:
    void ResetContext();
    void ResetProgram();
    void PushBack( __MemoryControl__<float> * mem);
    void PushBack( __MemoryControl__<double> * mem);
    void PushBack( __IndexControl__ * mem);

    //error function:
    void WriteError(cl_int err);
};

#endif
```

10.3.2 OpenCLManager.cpp

```
#include "preprocessor.h"
#include "OpenCLManager.h"
#include <iostream>
#include <fstream>
#include "mex.h"

//Definition of singleton:
OpenCLManager * __OpenCLManager__;

void OpenCLManager::PushBack( __MemoryControl__<float> * mem)
{
   numMemoryF++;
   if (numMemoryF == capMemoryF)
   {
      capMemoryF *= 2;
      __MemoryControl__<float> ** temp = (__MemoryControl__<float> **)mxMalloc(sizeof(__MemoryControl__<float>*) * capMemoryF);
      mexMakeMemoryPersistent(temp);
      for (unsigned int i = 0; i < numMemoryF-1; i += 1)
      {
         temp[i] = vectorMemoryF[i];
      }
      mxFree( vectorMemoryF );
      vectorMemoryF = temp;
```

```
  }
  vectorMemoryF[numMemoryF-1] = mem;
}
void OpenCLManager::PushBack( __MemoryControl__<double> * mem)
{
  numMemoryD++;
  if (numMemoryD == capMemoryD)
  {
    capMemoryD *= 2;
    __MemoryControl__<double> ** temp = (__MemoryControl__<double> **)mxMalloc(sizeof(__MemoryControl__<double>*) * capMemoryD);
    mexMakeMemoryPersistent(temp);
    for (unsigned int i = 0; i < numMemoryD-1; i += 1)
    {
      temp[i] = vectorMemoryD[i];
    }
    mxFree( vectorMemoryD );
    vectorMemoryD = temp;
  }
  vectorMemoryD[numMemoryD-1] = mem;
}
void OpenCLManager::PushBack( __IndexControl__ * mem)
{
  numIndex++;
  if (numIndex == capIndex)
  {
    capIndex *= 2;
    __IndexControl__ ** temp = (__IndexControl__ **)mxMalloc(sizeof(__IndexControl__*) * capIndex);
    mexMakeMemoryPersistent(temp);
    for (unsigned int i = 0; i < numIndex-1; i += 1)
    {
      temp[i] = vectorIndex[i];
    }
    mxFree( vectorIndex );
    vectorIndex = temp;
  }
  vectorIndex[numIndex-1] = mem;
}

//
//Print Commands:
//
void OpenCLManager::PrintGPUs()
{
  unsigned int tempCounter = 0;
  char name[64];
  char ext[4096];
```

```
  size_t size = 64;
  size_t ext_size = 4096;
  char platformname[64];
  for (unsigned int i = 0; i < numPlatforms; i += 1)
  {
    clGetPlatformInfo(vectorPlatforms[i], CL_PLATFORM_NAME,
        size, platformname, NULL);
    mexPrintf("Platform %u: %s\n",i,platformname);
    for (unsigned int j = 0; j < numDevices[i]; j += 1)
    {

      clGetDeviceInfo(vectorDevices[i][j], CL_DEVICE_NAME,
          size, name , NULL);
      clGetDeviceInfo(vectorDevices[i][j],
          CL_DEVICE_EXTENSIONS, ext_size, ext , NULL);
      mexPrintf("GPU %u: %s supports: %s\n",tempCounter, name,
          ext);
      tempCounter++;
    }
  }
}

//
//GPU buffer commands:
//
void OpenCLManager::SwapGPUBufferData(const cl_mem & buffer,
    void * ptr, unsigned int size, size_t sizeType)
{
  cl_int err;
  err = clEnqueueReadBuffer(queue, buffer, CL_TRUE, 0, size *
      sizeType, ptr, NULL, NULL, NULL);
  #ifdef __DEBUG__
    if (err < 0)
    {
      std::cout << "Failed to swap GPU buffer Data, code: " <<
          err << std::endl;
    }
  #endif
}

void OpenCLManager::WriteGPUBufferData(const cl_mem & buffer,
    void * ptr, unsigned int size, size_t sizeType)
{
  cl_int err;
  err = clEnqueueWriteBuffer(queue, buffer, CL_TRUE, 0, size *
      sizeType, ptr, NULL, NULL, NULL);
  #ifdef __DEBUG__
    if (err < 0)
    {
```

```
            std::cout << "Failed to swap GPU buffer Data, code: " << err << std::endl;
        }
    #endif
}

void OpenCLManager::ResizeGPUBuffer( __MemoryControl__<float> * control, unsigned int size)
{
    cl_int err;
    clReleaseMemObject(control->buffer);
    control->buffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, size * sizeof( float ), &(control->data[0]), &err);
    #ifdef __DEBUG__
        if (err < 0)
        {
            std::cout << "Failed to resize GPU buffer, code: " << err << std::endl;
        }
    #endif
    control->BuffersMatch = true;
}
void OpenCLManager::ResizeGPUBuffer( __MemoryControl__<double> * control, unsigned int size)
{
    cl_int err;
    clReleaseMemObject(control->buffer);
    control->buffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, size * sizeof( double ), &(control->data[0]), &err);
    #ifdef __DEBUG__
        if (err < 0)
        {
            std::cout << "Failed to resize GPU buffer, code: " << err << std::endl;
        }
    #endif
    control->BuffersMatch = true;
}
void OpenCLManager::ResizeGPUBuffer( __IndexControl__ * control, unsigned int size)
{
    cl_int err;
    clReleaseMemObject(control->buffer);
    control->buffer = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, size * sizeof( unsigned int ), &(control->data[0]), &err);
    #ifdef __DEBUG__
```

```
    if (err < 0)
    {
      std::cout << "Failed to resize GPU buffer, code; " <<
          err << std::endl;
    }
  #endif
  control->BuffersMatch = true;
}

__IndexControl__ * OpenCLManager::AllocateIndex(unsigned int *
    index, unsigned int size)
{
  cl_int err;
  //declare new memorycontroller:
  __IndexControl__ * temp = (__IndexControl__ *)mxMalloc(
      sizeof(__IndexControl__));
  mexMakeMemoryPersistent(temp);
  temp->size = size;
  temp->data = (unsigned int *)mxMalloc( sizeof( unsigned int
      ) * size );
  mexMakeMemoryPersistent(temp->data);

  if (index != NULL)
  {
    //populate array:
    temp->buffer = clCreateBuffer(context, CL_MEM_READ_WRITE |
        CL_MEM_COPY_HOST_PTR, size*sizeof( unsigned int ),
        index, &err);
    #ifdef __DEBUG__
      if (err < 0)
      {
        std::cout << "Error: Tried to create new memory object
            on GPU. Code: " << err << std::endl;
      }
    #endif
    temp->BuffersMatch = true;
  }
  else
  {
    temp->buffer = NULL;
    temp->BuffersMatch = false;
  }
  temp->RefCount = 1;

  //PushBack( temp );
  return temp;
}
```

```
__MemoryControl__<float> * OpenCLManager:: AllocateMemory(
    float * data, unsigned int size)
{
  //openCL error var:
  cl_int err;


  //declare new memorycontroller:
  __MemoryControl__<float> * temp = (__MemoryControl__<float>
      *)mxMalloc(sizeof(__MemoryControl__<float>));
  //temp = new(temp) __MemoryControl__<float>();
  mexMakeMemoryPersistent(temp);
  temp->size = size;
  temp->data = (float *)mxMalloc( sizeof( float ) * size );
  mexMakeMemoryPersistent(temp->data);

  if (data != NULL)
  {
    temp->buffer = clCreateBuffer(context, CL_MEM_READ_WRITE |
        CL_MEM_COPY_HOST_PTR, size*sizeof(float), data, &err)
        ;
    #ifdef __DEBUG__
      if (err < 0)
      {
        std::cout << "Error: Tried to create new memory object
            on GPU. Code: " << err << std::endl;
      }
    #endif
    temp->BuffersMatch = true;
  }
  else
  {
    temp->buffer = clCreateBuffer(context, CL_MEM_READ_WRITE,
        size*sizeof(float), NULL, &err);
    #ifdef __DEBUG__
      if (err < 0)
      {
        std::cout << "Error: Tried to create new memory object
            on GPU. Code: " << err << std::endl;
      }
    #endif
    temp->BuffersMatch = true;
  }
  temp->RefCount = 1;
  //PushBack( temp );
  return temp;
}
__MemoryControl__<double> * OpenCLManager::AllocateMemory(
    double * data, unsigned int size)
```

```
{
  //openCL error var:
  cl_int err;


  //declare new memorycontroller:
  __MemoryControl__<double> * temp = (__MemoryControl__<double
     > *)mxMalloc(sizeof(__MemoryControl__<double>));
  //temp = new(temp) __MemoryControl__<float>();
  mexMakeMemoryPersistent(temp);
  temp->size = size;
  temp->data = (double *)mxMalloc( sizeof( double ) * size );
  mexMakeMemoryPersistent(temp->data);

  if (data != NULL)
  {
    temp->buffer = clCreateBuffer(context, CL_MEM_READ_WRITE |
         CL_MEM_COPY_HOST_PTR, size*sizeof(double), data, &err
        );
    #ifdef __DEBUG__
      if (err < 0)
      {
        std::cout << "Error: Tried to create new memory object
             on GPU. Code: " << err << std::endl;
      }
    #endif
    temp->BuffersMatch = true;
  }
  else
  {
    temp->buffer = clCreateBuffer(context, CL_MEM_READ_WRITE,
        size*sizeof(double), NULL, &err);
    #ifdef __DEBUG__
      if (err < 0)
      {
        std::cout << "Error: Tried to create new memory object
             on GPU. Code: " << err << std::endl;
      }
    #endif
    temp->BuffersMatch = true;
  }
  temp->RefCount = 1;
  //PushBack( temp );
  return temp;
}

//
//Linear algebra components:
//
```

```


//Multiply by constant:
void OpenCLManager::VectorTimesConstantFF(cl_mem & vector, cl_mem & output, float constant, unsigned int vectorSize, cl_event * event)
{
  cl_int err;
  clSetKernelArg( kernelVectorTimesConstantFF, 0, sizeof( cl_mem ), &vector);
  clSetKernelArg( kernelVectorTimesConstantFF, 1, sizeof( cl_mem ), &output);
  clSetKernelArg( kernelVectorTimesConstantFF, 2, sizeof( float ), &constant);
  clSetKernelArg( kernelVectorTimesConstantFF, 3, sizeof( unsigned int), &vectorSize);

  size_t global_size;
  if (vectorSize % (VectorConstantThreadsPerGroup * VectorConstantRowsPerThread) == 0)
  {
    global_size = vectorSize/(VectorConstantRowsPerThread);
  }
  else
  {
    global_size = (vectorSize/(VectorConstantThreadsPerGroup * VectorConstantRowsPerThread)+1) * VectorConstantThreadsPerGroup;
  }
  size_t local_size = VectorConstantThreadsPerGroup;
  err = clEnqueueNDRangeKernel(queue, kernelVectorTimesConstantFF, 1, NULL, &global_size, &local_size, 0, NULL, event);
  #ifdef __DEBUG__
    if (err < 0)
    {
      std::cout << "Failed to enque VectorTimesConstant, code: " << err << std::endl;
    }
  #endif
}
void OpenCLManager::VectorTimesConstantD(cl_kernel kernel, cl_mem & vector, cl_mem & output, double constant, unsigned int vectorSize, cl_event * event)
{
  cl_int err;
  clSetKernelArg( kernel, 0, sizeof( cl_mem ), &vector);
  clSetKernelArg( kernel, 1, sizeof( cl_mem ), &output);
  clSetKernelArg( kernel, 2, sizeof( double ), &constant);
```

```
  clSetKernelArg( kernel, 3, sizeof( unsigned int), &vectorSize);

  size_t global_size;
  if (vectorSize % (VectorConstantThreadsPerGroup * VectorConstantRowsPerThread) == 0)
  {
    global_size = vectorSize/(VectorConstantRowsPerThread);
  }
  else
  {
    global_size = (vectorSize/(VectorConstantThreadsPerGroup * VectorConstantRowsPerThread)+1) * VectorConstantThreadsPerGroup;
  }
  size_t local_size = VectorConstantThreadsPerGroup;
  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size, &local_size, 0, NULL, event);
  #ifdef __DEBUG__
    if (err < 0)
    {
      std::cout << "Failed to enque VectorTimesConstant, code: " << err << std::endl;
    }
  #endif
}
void OpenCLManager::VectorTimesConstantDD(cl_mem & vector, cl_mem & output, double constant, unsigned int vectorSize, cl_event * event)
{
  VectorTimesConstantD(kernelVectorTimesConstantDD, vector, output, constant, vectorSize, event);
}
void OpenCLManager::VectorTimesConstantFD(cl_mem & vector, cl_mem & output, double constant, unsigned int vectorSize, cl_event * event)
{
  VectorTimesConstantD(kernelVectorTimesConstantFD, vector, output, constant, vectorSize, event);
}

//
//Vector Minus Vector:
//
void OpenCLManager::VectorOperatorVector(cl_kernel kernel, cl_mem & vector1, cl_mem & vector2, cl_mem & output, unsigned int length, cl_event * event)
{
  cl_int err;
```

```
  clSetKernelArg( kernel, 0, sizeof( cl_mem ), &vector1);
  clSetKernelArg( kernel, 1, sizeof( cl_mem ), &vector2);
  clSetKernelArg( kernel, 2, sizeof( cl_mem ), &output);
  clSetKernelArg( kernel, 3, sizeof( unsigned int ), &length);

  size_t global_size;
  if (length % (VectorAndVectorThreadsPerGroup *
      VectorAndVectorRowsPerThread) == 0)
  {
    global_size = length/(VectorAndVectorRowsPerThread);
  }
  else
  {
    global_size = (length/(VectorAndVectorThreadsPerGroup *
        VectorAndVectorRowsPerThread)+1) *
        VectorAndVectorThreadsPerGroup;
  }
  size_t local_size = VectorAndVectorThreadsPerGroup;

  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
      global_size, &local_size, 0, NULL, event);
  #ifdef __DEBUG__
    if (err < 0)
    {
      std::cout << "Failed to enque VectorAndVector, code: "
          << err << std::endl;
    }
  #endif
}
void OpenCLManager::VectorMinusVectorFF(cl_mem & vector1,
     cl_mem & vector2, cl_mem & output, unsigned int length,
     cl_event * event)
{
  VectorOperatorVector(kernelVectorMinusVectorFF, vector1,
      vector2, output, length, event);
}
void OpenCLManager::VectorMinusVectorFD(cl_mem & vector1,
     cl_mem & vector2, cl_mem & output, unsigned int length,
     cl_event * event)
{
  VectorOperatorVector(kernelVectorMinusVectorFD, vector1,
      vector2, output, length, event);
}
void OpenCLManager::VectorMinusVectorDF(cl_mem & vector1,
     cl_mem & vector2, cl_mem & output, unsigned int length,
     cl_event * event)
{
  VectorOperatorVector(kernelVectorMinusVectorDF, vector1,
      vector2, output, length, event);
```

```
}
void OpenCLManager::VectorMinusVectorDD(cl_mem & vector1,
    cl_mem & vector2, cl_mem & output, unsigned int length,
    cl_event * event)
{
  VectorOperatorVector(kernelVectorMinusVectorDD, vector1,
      vector2, output, length, event);
}
void OpenCLManager::VectorPlusVectorFF(cl_mem & vector1,
    cl_mem & vector2, cl_mem & output, unsigned int length,
    cl_event * event)
{
  VectorOperatorVector(kernelVectorPlusVectorFF, vector1,
      vector2, output, length, event);
}
void OpenCLManager::VectorPlusVectorFD(cl_mem & vector1,
    cl_mem & vector2, cl_mem & output, unsigned int length,
    cl_event * event)
{
  VectorOperatorVector(kernelVectorPlusVectorFD, vector1,
      vector2, output, length, event);
}
void OpenCLManager::VectorPlusVectorDF(cl_mem & vector1,
    cl_mem & vector2, cl_mem & output, unsigned int length,
    cl_event * event)
{
  VectorOperatorVector(kernelVectorPlusVectorDF, vector1,
      vector2, output, length, event);
}
void OpenCLManager::VectorPlusVectorDD(cl_mem & vector1,
    cl_mem & vector2, cl_mem & output, unsigned int length,
    cl_event * event)
{
  VectorOperatorVector(kernelVectorPlusVectorDD, vector1,
      vector2, output, length, event  );
}


//Vector times vector constant
void OpenCLManager::VectorOperatorVectorConstant(cl_kernel
    kernel, cl_mem & vector1, cl_mem & vector2, cl_mem &
    output, double con, unsigned int length, cl_event * event)
{
  cl_int err;
  clSetKernelArg( kernel, 0, sizeof( cl_mem ), &vector1);
  clSetKernelArg( kernel, 1, sizeof( cl_mem ), &vector2);
  clSetKernelArg( kernel, 2, sizeof( cl_mem ), &output);
  clSetKernelArg( kernel, 3, sizeof( double ), &con);
  clSetKernelArg( kernel, 4, sizeof( unsigned int ), &length);
```

```

  size_t global_size;
  if (length % (VectorAndVectorThreadsPerGroup *
      VectorAndVectorRowsPerThread) == 0)
  {
    global_size = length/(VectorAndVectorRowsPerThread);
  }
  else
  {
    global_size = (length/(VectorAndVectorThreadsPerGroup *
        VectorAndVectorRowsPerThread)+1) *
        VectorAndVectorThreadsPerGroup;
  }
  size_t local_size = VectorAndVectorThreadsPerGroup;

  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
      global_size, &local_size, 0, NULL, event);
  #ifdef __DEBUG__
    if (err < 0)
    {
      std::cout << "Failed to enque VectorAndVectorConstant,
          code: " << err << std::endl;
    }
  #endif
}

void OpenCLManager::VectorMinusVectorConstantFF(cl_mem &
    vector1, cl_mem & vector2, cl_mem & output, float con,
    unsigned int length, cl_event * event)
{
  cl_int err;
  clSetKernelArg( kernelVectorMinusVectorConstantFF, 0, sizeof
      ( cl_mem ), &vector1);
  clSetKernelArg( kernelVectorMinusVectorConstantFF, 1, sizeof
      ( cl_mem ), &vector2);
  clSetKernelArg( kernelVectorMinusVectorConstantFF, 2, sizeof
      ( cl_mem ), &output);
  clSetKernelArg( kernelVectorMinusVectorConstantFF, 3, sizeof
      ( float ), &con);
  clSetKernelArg( kernelVectorMinusVectorConstantFF, 4, sizeof
      ( unsigned int ), &length);

  size_t global_size;
  if (length % (VectorAndVectorThreadsPerGroup *
      VectorAndVectorRowsPerThread) == 0)
  {
    global_size = length/(VectorAndVectorRowsPerThread);
  }
  else
```

```
  {
    global_size = (length/(VectorAndVectorThreadsPerGroup *
        VectorAndVectorRowsPerThread)+1) *
        VectorAndVectorThreadsPerGroup;
  }
  size_t local_size = VectorAndVectorThreadsPerGroup;

  err = clEnqueueNDRangeKernel(queue,
      kernelVectorMinusVectorConstantFF, 1, NULL, &global_size
      , &local_size, 0, NULL, event);
  #ifdef __DEBUG__
    if (err < 0)
    {
      std::cout << "Failed to enque VectorAndVectorConstant,
          code: " << err << std::endl;
    }
  #endif
}

void OpenCLManager::VectorMinusVectorConstantFD(cl_mem &
    vector1, cl_mem & vector2, cl_mem & output, double con,
    unsigned int length, cl_event * event)
{
  VectorOperatorVectorConstant(
      kernelVectorMinusVectorConstantFD, vector1, vector2,
      output, con, length, event);
}
void OpenCLManager::VectorMinusVectorConstantDF(cl_mem &
    vector1, cl_mem & vector2, cl_mem & output, double con,
    unsigned int length, cl_event * event)
{
  VectorOperatorVectorConstant(
      kernelVectorMinusVectorConstantDF, vector1, vector2,
      output, con, length, event);
}
void OpenCLManager::VectorMinusVectorConstantDD(cl_mem &
    vector1, cl_mem & vector2, cl_mem & output, double con,
    unsigned int length, cl_event * event)
{
  VectorOperatorVectorConstant(
      kernelVectorMinusVectorConstantDD, vector1, vector2,
      output, con, length, event);
}

//
//Norm:
//
void OpenCLManager::Norm(cl_kernel kernel, cl_mem & input,
    cl_mem & output, unsigned int threadsize, unsigned int
```

```
    problemsize, cl_event * event)
{
  cl_int err;
  clSetKernelArg( kernel, 0, sizeof( cl_mem ), &input);
  clSetKernelArg( kernel, 1, sizeof( cl_mem ), &output);
  clSetKernelArg( kernel, 2, sizeof( unsigned int ), &threadsize);
  clSetKernelArg( kernel, 3, sizeof( unsigned int ), &problemsize);
  unsigned numWorkItems;
  if (problemsize % (threadsize*threadsize) == 0)
  {
    numWorkItems = problemsize/(threadsize*threadsize);
  }
  else
  {
    numWorkItems = problemsize/(threadsize*threadsize)+1;
  }
  const size_t global_size = numWorkItems*threadsize;
  const size_t local_size = threadsize;

  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size, &local_size, 0, NULL, event);
  #ifdef __DEBUG__
    if (err < 0)
    {
      std::cout << "Failed to enque Norm, code: " << err << std::endl;
    }
  #endif
}
void OpenCLManager::ParallelSumReductionF(cl_mem & input, cl_mem & output, unsigned int threadsize, unsigned int problemsize, cl_event * event)
{
  Norm(kernelReductionF, input, output, threadsize, problemsize, event);
}
void OpenCLManager::ParallelSumReductionD(cl_mem & input, cl_mem & output, unsigned int threadsize, unsigned int problemsize, cl_event * event)
{
  Norm(kernelReductionD, input, output, threadsize, problemsize, event);
}

void OpenCLManager::Norm2F(cl_mem & input, cl_mem & output, unsigned int threadsize, unsigned int problemsize, cl_event * event)
```

```
{
  Norm(kernelNorm2F, input, output, threadsize, problemsize,
      event);
}
void OpenCLManager::Norm2D(cl_mem & input, cl_mem & output,
    unsigned int threadsize, unsigned int problemsize,
    cl_event * event)
{
  Norm(kernelNorm2D, input, output, threadsize, problemsize,
      event);
}
void OpenCLManager::NormInfF(cl_mem & input, cl_mem & output,
    unsigned int threadsize, unsigned int problemsize,
    cl_event * event)
{
  Norm(kernelNormInfF, input, output, threadsize, problemsize,
      event);
}
void OpenCLManager::NormInfD(cl_mem & input, cl_mem & output,
    unsigned int threadsize, unsigned int problemsize,
    cl_event * event)
{
  Norm(kernelNormInfD, input, output, threadsize, problemsize,
      event);
}

//
//Matrix vector:
//
void OpenCLManager::SparseMatrixVector(cl_kernel kernel,
    cl_mem & matData, cl_mem & matCol, cl_mem & matRow, cl_mem
    & vecData, cl_mem & returnData, unsigned int height,
    unsigned int width, unsigned int numIndexes, unsigned int
    rowVectorLength, cl_event * event)
{
  cl_int err;
  clSetKernelArg( kernel, 0, sizeof( cl_mem ), &matData);
  clSetKernelArg( kernel, 1, sizeof( cl_mem ), &matCol);
  clSetKernelArg( kernel, 2, sizeof( cl_mem ), &matRow);
  clSetKernelArg( kernel, 3, sizeof( cl_mem ), &vecData);
  clSetKernelArg( kernel, 4, sizeof( cl_mem ), &returnData);
  clSetKernelArg( kernel, 5, sizeof( unsigned int ), &
      rowVectorLength);

  size_t global_size;
  if (rowVectorLength % (SparseMatrixVectorThreadsPerGroup *
      SparseMatrixVectorRowsPerThread) == 0)
  {
```

```
        global_size = rowVectorLength/(SparseMatrixVectorRowsPerThread);
    }
    else
    {
        global_size = (rowVectorLength/(SparseMatrixVectorRowsPerThread*SparseMatrixVectorThreadsPerGroup)+1) * SparseMatrixVectorThreadsPerGroup;
    }
    size_t local_size = SparseMatrixVectorThreadsPerGroup;

    err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &global_size, &local_size, 0, NULL, event);
    #ifdef __DEBUG__
        if (err < 0)
        {
            mexPrintf("Failed to enque SparseMatrixVector. Reason:");
            WriteError(err);
        }
    #endif
}

void OpenCLManager::SparseMatrixVectorFF(cl_mem & matData, cl_mem & matCol, cl_mem & matRow, cl_mem & vecData, cl_mem & returnData, unsigned int height, unsigned int width, unsigned int numIndexes, unsigned int rowVectorLength, cl_event * event)
{
    SparseMatrixVector(kernelSparseMatrixVectorFF, matData, matCol, matRow, vecData, returnData, height, width, numIndexes, rowVectorLength, event);
}

void OpenCLManager::SparseMatrixVectorFD(cl_mem & matData, cl_mem & matCol, cl_mem & matRow, cl_mem & vecData, cl_mem & returnData, unsigned int height, unsigned int width, unsigned int numIndexes, unsigned int rowVectorLength, cl_event * event)
{
    SparseMatrixVector(kernelSparseMatrixVectorFD, matData, matCol, matRow, vecData, returnData, height, width, numIndexes, rowVectorLength, event);
}

void OpenCLManager::SparseMatrixVectorDF(cl_mem & matData, cl_mem & matCol, cl_mem & matRow, cl_mem & vecData, cl_mem & returnData, unsigned int height, unsigned int width,
```

```
    unsigned int numIndexes, unsigned int rowVectorLength,
    cl_event * event)
{
  SparseMatrixVector(kernelSparseMatrixVectorDF, matData,
      matCol, matRow, vecData, returnData, height, width,
      numIndexes, rowVectorLength, event);
}

void OpenCLManager::SparseMatrixVectorDD(cl_mem & matData,
    cl_mem & matCol, cl_mem & matRow, cl_mem & vecData, cl_mem
     & returnData, unsigned int height, unsigned int width,
    unsigned int numIndexes, unsigned int rowVectorLength,
    cl_event * event)
{
  SparseMatrixVector(kernelSparseMatrixVectorDD, matData,
      matCol, matRow, vecData, returnData, height, width,
      numIndexes, rowVectorLength, event);
}

void OpenCLManager::BandMatrixVector(cl_kernel kernel, cl_mem
    & matData, cl_mem & vecData, cl_mem & returnData, unsigned
     int height, unsigned int bandwidth, unsigned int length,
    cl_event * event)
{
  cl_int err;
  clSetKernelArg( kernel, 0, sizeof( cl_mem ), &matData);
  clSetKernelArg( kernel, 1, sizeof( cl_mem ), &vecData);
  clSetKernelArg( kernel, 2, sizeof( cl_mem ), &returnData);
  clSetKernelArg( kernel, 3, sizeof( unsigned int ), &height);
  clSetKernelArg( kernel, 4, sizeof( unsigned int ), &
      bandwidth);
  clSetKernelArg( kernel, 5, sizeof( unsigned int ), &height);

  size_t global_size;
  if (height % (BandMatrixVectorThreadsPerGroup *
      BandMatrixVectorRowsPerThread) == 0)
  {
    global_size = height/(BandMatrixVectorRowsPerThread);
  }
  else
  {
    global_size = (height/(BandMatrixVectorRowsPerThread*
        BandMatrixVectorThreadsPerGroup)+1) *
        BandMatrixVectorThreadsPerGroup;
  }
  size_t local_size = BandMatrixVectorThreadsPerGroup;
  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
      global_size, &local_size, 0, NULL, event);
  #ifdef __DEBUG__
```

```
    if (err < 0)
    {
      std::cout << "Failed to enque BandMatrixVector, code: "
          << err << std::endl;
    }
  #endif
}
void OpenCLManager::BandMatrixVectorFF(cl_mem & matData,
    cl_mem & vecData, cl_mem & returnData, unsigned int height
    , unsigned int bandwidth, unsigned int length, cl_event *
    event)
{
  BandMatrixVector(kernelBandMatrixVectorFF, matData, vecData,
      returnData, height, bandwidth, length, event);
}
void OpenCLManager::BandMatrixVectorDD(cl_mem & matData,
    cl_mem & vecData, cl_mem & returnData, unsigned int height
    , unsigned int bandwidth, unsigned int length, cl_event *
    event)
{
  BandMatrixVector(kernelBandMatrixVectorDD, matData, vecData,
      returnData, height, bandwidth, length, event);
}
void OpenCLManager::BandMatrixVectorFD(cl_mem & matData,
    cl_mem & vecData, cl_mem & returnData, unsigned int height
    , unsigned int bandwidth, unsigned int length, cl_event *
    event)
{
  BandMatrixVector(kernelBandMatrixVectorFD, matData, vecData,
      returnData, height, bandwidth, length, event);
}
void OpenCLManager::BandMatrixVectorDF(cl_mem & matData,
    cl_mem & vecData, cl_mem & returnData, unsigned int height
    , unsigned int bandwidth, unsigned int length, cl_event *
    event)
{
  BandMatrixVector(kernelBandMatrixVectorDF, matData, vecData,
      returnData, height, bandwidth, length, event);
}


//Refinement methods:

void OpenCLManager::FineToCoarse(cl_kernel kernel, cl_mem &
    fineData, cl_mem & corData, unsigned int corWidth,
    unsigned int corHeight, cl_event * event)
{
  cl_int err;
  unsigned int fineWidth = 2*corWidth-1;
```

```
    clSetKernelArg( kernel, 0, sizeof( cl_mem ), &fineData);
    clSetKernelArg( kernel, 1, sizeof( cl_mem ), &corData);
    clSetKernelArg( kernel, 2, sizeof( unsigned int ), &
        fineWidth);
    clSetKernelArg( kernel, 3, sizeof( unsigned int ), &corWidth
        );

    size_t global_size;
    if (corWidth*corHeight % (FTCThreadsPerGroup *
        FTCRowsPerThread) == 0)
    {
      global_size = corWidth*corHeight/(FTCRowsPerThread);
    }
    else
    {
      global_size = (corWidth*corHeight/(FTCThreadsPerGroup *
          FTCRowsPerThread)+1) * FTCThreadsPerGroup;
    }
    size_t local_size = FTCThreadsPerGroup;
    err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
        global_size, &local_size, 0, NULL, event);
   #ifdef __DEBUG__
      if (err < 0)
      {
        std::cout << "Failed to enque FineToCoarse, code: " <<
            err << std::endl;
      }
   #endif
}
void OpenCLManager::FineToCoarseFF(cl_mem & fineData, cl_mem &
      corData, unsigned int corWidth, unsigned int corHeight,
      cl_event * event)
{
    FineToCoarse(kernelRefineFTCFF, fineData, corData, corWidth,
        corHeight, event);
}
void OpenCLManager::FineToCoarseDF(cl_mem & fineData, cl_mem &
      corData, unsigned int corWidth, unsigned int corHeight,
      cl_event * event)
{
    FineToCoarse(kernelRefineFTCDF, fineData, corData, corWidth,
        corHeight, event);
}
void OpenCLManager::FineToCoarseDD(cl_mem & fineData, cl_mem &
      corData, unsigned int corWidth, unsigned int corHeight,
      cl_event * event)
{
    FineToCoarse(kernelRefineFTCDD, fineData, corData, corWidth,
        corHeight, event);
```

```
}
void OpenCLManager::CoarseToFine(cl_kernel kernel, cl_mem &
    fineData, cl_mem & corData, unsigned int fineWidth,
    unsigned int fineHeight, cl_event * event)
{
  cl_int err;
  unsigned int corWidth = fineWidth/2+1;
  clSetKernelArg( kernel, 0, sizeof( cl_mem ), &fineData);
  clSetKernelArg( kernel, 1, sizeof( cl_mem ), &corData);
  clSetKernelArg( kernel, 2, sizeof( unsigned int ), &corWidth
      );
  clSetKernelArg( kernel, 3, sizeof( unsigned int ), &
      fineWidth);

  size_t global_size;
  if (fineWidth*fineHeight % (CTFThreadsPerGroup *
      CTFRowsPerThread) == 0)
  {
    global_size = fineWidth*fineHeight/(CTFRowsPerThread);
  }
  else
  {
    global_size = (fineWidth*fineHeight/(CTFThreadsPerGroup *
        CTFRowsPerThread)+1) * CTFThreadsPerGroup;
  }
  size_t local_size = CTFThreadsPerGroup;
  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
      global_size, &local_size, 0, NULL, event);
 #ifdef __DEBUG__
    if (err < 0)
    {
      std::cout << "Failed to enque CoarseToFine, code: " <<
          err << std::endl;
    }
 #endif
}
void OpenCLManager::CoarseToFineFF(cl_mem & fineData, cl_mem &
      corData, unsigned int fineWidth, unsigned int fineHeight,
      cl_event * event)
{
  CoarseToFine(kernelRefineCTFFF, fineData, corData, fineWidth
      , fineHeight, event);
}
void OpenCLManager::CoarseToFineFD(cl_mem & fineData, cl_mem &
      corData, unsigned int fineWidth, unsigned int fineHeight,
      cl_event * event)
{
  CoarseToFine(kernelRefineCTFFD, fineData, corData, fineWidth
      , fineHeight, event);
```

```
}
void OpenCLManager::CoarseToFineDD(cl_mem & fineData, cl_mem & corData, unsigned int fineWidth, unsigned int fineHeight, cl_event * event)
{
  CoarseToFine(kernelRefineCTFDD, fineData, corData, fineWidth, fineHeight, event);
}


//
//Jacobi methods:
//
void OpenCLManager::JacobiF(cl_mem & output, cl_mem & input, cl_mem & rightData, unsigned int width, unsigned int height, float spacing, unsigned int gridSize, cl_event * event)
{
  cl_int err;
  clSetKernelArg( kernelJacobiF, 0, sizeof( cl_mem ), &output);
  clSetKernelArg( kernelJacobiF, 1, sizeof( cl_mem ), &input);
  clSetKernelArg( kernelJacobiF, 2, sizeof( cl_mem ), &rightData);
  clSetKernelArg( kernelJacobiF, 3, sizeof( unsigned int ), &width);
  clSetKernelArg( kernelJacobiF, 4, sizeof( unsigned int ), &height);
  clSetKernelArg( kernelJacobiF, 5, sizeof( float ), &spacing);

  size_t global_size;
  if (width*height % (JacobiThreadsPerGroup * JacobiRowsPerThread) == 0)
  {
    global_size = width*height/(JacobiRowsPerThread);
  }
  else
  {
    global_size = (width*height/(JacobiThreadsPerGroup * JacobiRowsPerThread)+1) * JacobiThreadsPerGroup;
  }
  size_t local_size = JacobiThreadsPerGroup;
  err = clEnqueueNDRangeKernel(queue, kernelJacobiF, 1, NULL, &global_size, &local_size, 0, NULL, event);
 #ifdef __DEBUG__
  if (err < 0)
  {
```

```
        std::cout << "Failed to enqueue kernel JacobiF, code: " << err << std::endl;
    }
    #endif
}

void OpenCLManager::JacobiD(cl_mem & output, cl_mem & input, cl_mem & rightData, unsigned int width, unsigned int height, double spacing, unsigned int gridSize, cl_event * event)
{
    cl_int err;
    clSetKernelArg( kernelJacobiD, 0, sizeof( cl_mem ), &output);
    clSetKernelArg( kernelJacobiD, 1, sizeof( cl_mem ), &input);
    clSetKernelArg( kernelJacobiD, 2, sizeof( cl_mem ), &rightData);
    clSetKernelArg( kernelJacobiD, 3, sizeof( unsigned int ), &width);
    clSetKernelArg( kernelJacobiD, 4, sizeof( unsigned int ), &height);
    clSetKernelArg( kernelJacobiD, 5, sizeof( double ), &spacing);

    size_t global_size;
    if (width*height % (JacobiThreadsPerGroup * JacobiRowsPerThread) == 0)
    {
        global_size = width*height/(JacobiRowsPerThread);
    }
    else
    {
        global_size = (width*height/(JacobiThreadsPerGroup * JacobiRowsPerThread)+1) * JacobiThreadsPerGroup;
    }
    size_t local_size = JacobiThreadsPerGroup;
    err = clEnqueueNDRangeKernel(queue, kernelJacobiD, 1, NULL, &global_size, &local_size, 0, NULL, event);
    #ifdef __DEBUG__
    if (err < 0)
    {
        std::cout << "Failed to enqueue kernel JacobiF, code: " << err << std::endl;
    }
    #endif
}

void OpenCLManager::JacobiMethodF(cl_kernel kernel, cl_mem & leftData, cl_mem & rightData, unsigned int width, unsigned
```

```
    int height, float spacing, unsigned int gridSize,
    cl_event * event)
{
  cl_int err;
  clSetKernelArg( kernel, 0, sizeof( cl_mem ), &leftData);
  clSetKernelArg( kernel, 1, sizeof( cl_mem ), &rightData);
  clSetKernelArg( kernel, 2, sizeof( unsigned int ), &width);
  clSetKernelArg( kernel, 3, sizeof( unsigned int ), &height);
  clSetKernelArg( kernel, 4, sizeof( float ), &spacing);

  size_t global_size;
  if (width*height % (RBGSThreadsPerGroup * RBGSRowsPerThread)
      == 0)
  {
    global_size = width*height/(RBGSRowsPerThread);
  }
  else
  {
    global_size = (width*height/(RBGSThreadsPerGroup *
        RBGSRowsPerThread)+1) * RBGSThreadsPerGroup;
  }
  size_t local_size = RBGSThreadsPerGroup;
  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
      global_size, &local_size, 0, NULL, event);
  #ifdef __DEBUG__
  if (err < 0)
  {
    std::cout << "Failed to enqueue kernel JackkobiMethodOdd,
        code: " << err << std::endl;
  }
  #endif
}
void OpenCLManager::JacobiMethodD(cl_kernel kernel, cl_mem &
    leftData, cl_mem & rightData, unsigned int width, unsigned
    int height, double spacing, unsigned int gridSize,
    cl_event * event)
{
  cl_int err;
  clSetKernelArg( kernel, 0, sizeof( cl_mem ), &leftData);
  clSetKernelArg( kernel, 1, sizeof( cl_mem ), &rightData);
  clSetKernelArg( kernel, 2, sizeof( unsigned int ), &width);
  clSetKernelArg( kernel, 3, sizeof( unsigned int ), &height);
  clSetKernelArg( kernel, 4, sizeof( double ), &spacing);

  size_t global_size;
  if (width*height % (RBGSThreadsPerGroup * RBGSRowsPerThread)
      == 0)
  {
    global_size = width*height/(RBGSRowsPerThread);
```

```
  }
  else
  {
    global_size = (width*height/(RBGSThreadsPerGroup *
        RBGSRowsPerThread)+1) * RBGSThreadsPerGroup;
  }
  size_t local_size = RBGSThreadsPerGroup;

  err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
      global_size, &local_size, 0, NULL, event);
  #ifdef __DEBUG__
  if (err < 0)
  {
    std::cout << "Failed to enqueue kernel JacobiMethodOdd,
        code: " << err << std::endl;
  }
  #endif
}
void OpenCLManager::JacobiMethodOddD(cl_mem & leftData, cl_mem
     & rightData, unsigned int width, unsigned int height,
    double spacing, unsigned int gridSize, cl_event * event)
{
  JacobiMethodD(kernelJacobiMethodOddD, leftData, rightData,
      width, height, spacing, gridSize, event);
}
void OpenCLManager::JacobiMethodEvenD(cl_mem & leftData,
    cl_mem & rightData, unsigned int width, unsigned int
    height, double spacing, unsigned int gridSize, cl_event *
    event)
{
  JacobiMethodD(kernelJacobiMethodEvenD, leftData, rightData,
      width, height, spacing, gridSize, event);
}
void OpenCLManager::JacobiMethodOddF(cl_mem & leftData, cl_mem
     & rightData, unsigned int width, unsigned int height,
    float spacing, unsigned int gridSize, cl_event * event)
{
  JacobiMethodF(kernelJacobiMethodOddF, leftData, rightData,
      width, height, spacing, gridSize, event);
}
void OpenCLManager::JacobiMethodEvenF(cl_mem & leftData,
    cl_mem & rightData, unsigned int width, unsigned int
    height, float spacing, unsigned int gridSize, cl_event *
    event)
{
  JacobiMethodF(kernelJacobiMethodEvenF, leftData, rightData,
      width, height, spacing, gridSize, event);
}
```

```

void OpenCLManager::JacobiDefectF(cl_mem & leftData, cl_mem &
    rightData, cl_mem & defectData, unsigned int width,
    unsigned int height, float spacing, cl_event * event)
{
  cl_int err;
  clSetKernelArg(kernelJacobiDefectF, 0, sizeof( cl_mem ), &
      leftData);
  clSetKernelArg(kernelJacobiDefectF, 1, sizeof( cl_mem ), &
      rightData);
  clSetKernelArg(kernelJacobiDefectF, 2, sizeof( cl_mem ), &
      defectData);
  clSetKernelArg(kernelJacobiDefectF, 3, sizeof( unsigned int
      ), &width);
  clSetKernelArg(kernelJacobiDefectF, 4, sizeof( unsigned int
      ), &height);
  clSetKernelArg(kernelJacobiDefectF, 5, sizeof( float ), &
      spacing);

  size_t global_size;
  if (width*height % (DefectThreadsPerGroup *
      DefectRowsPerThread) == 0)
  {
    global_size = width*height/(DefectRowsPerThread);
  }
  else
  {
    global_size = (width*height/(DefectThreadsPerGroup *
        DefectRowsPerThread)+1) * DefectThreadsPerGroup;
  }
  size_t local_size = DefectThreadsPerGroup;
  err = clEnqueueNDRangeKernel(queue, kernelJacobiDefectF, 1,
      NULL, &global_size, &local_size, 0, NULL, event);
  #ifdef __DEBUG__
    if (err < 0)
    {
      std::cout << "Failed to enqueue kernel JacobiDefect" <<
          std::endl;
    }
  #endif
}
void OpenCLManager::JacobiDefectD(cl_mem & leftData, cl_mem &
    rightData, cl_mem & defectData, unsigned int width,
    unsigned int height, double spacing, cl_event * event)
{
  cl_int err;
  clSetKernelArg(kernelJacobiDefectD, 0, sizeof( cl_mem ), &
      leftData);
```

```
    clSetKernelArg(kernelJacobiDefectD, 1, sizeof( cl_mem ), &rightData);
    clSetKernelArg(kernelJacobiDefectD, 2, sizeof( cl_mem ), &defectData);
    clSetKernelArg(kernelJacobiDefectD, 3, sizeof( unsigned int ), &width);
    clSetKernelArg(kernelJacobiDefectD, 4, sizeof( unsigned int ), &height);
    clSetKernelArg(kernelJacobiDefectD, 5, sizeof( double ), &spacing);

    size_t global_size;
    if (width*height % (DefectThreadsPerGroup * DefectRowsPerThread) == 0)
    {
      global_size = width*height/(DefectRowsPerThread);
    }
    else
    {
      global_size = (width*height/(DefectThreadsPerGroup * DefectRowsPerThread)+1) * DefectThreadsPerGroup;
    }
    size_t local_size = DefectThreadsPerGroup;

    err = clEnqueueNDRangeKernel(queue, kernelJacobiDefectD, 1, NULL, &global_size, &local_size, 0, NULL, event);
   #ifdef __DEBUG__
      if (err < 0)
      {
        std::cout << "Failed to enqueue kernel JacobiDefect" << std::endl;
      }
   #endif
}

//
//Methods:
//
void OpenCLManager::AddSource( char * name )
{
    numSourceFiles++;
    char ** temp = (char **)mxMalloc( sizeof(char *) * numSourceFiles);
    mexMakeMemoryPersistent(temp);
    for (unsigned int i = 0; i < numSourceFiles-1; i += 1)
    {
      temp[i] = vectorSourceFiles[i];
    }
    temp[numSourceFiles-1] = name;
```

```
    if (vectorSourceFiles != NULL)
      mxFree( vectorSourceFiles );
    vectorSourceFiles = temp;
}

//
//Memory deletion:
//
void OpenCLManager::DeleteMemory(__MemoryControl__<float> *
    mem)
{
    clReleaseMemObject(mem->buffer);
    mxFree(mem->data);
    mxFree(mem);
}
void OpenCLManager::DeleteMemory(__MemoryControl__<double> *
    mem)
{
    clReleaseMemObject(mem->buffer);
    mxFree(mem->data);
    mxFree(mem);
}
void OpenCLManager::DeleteIndex(__IndexControl__ * mem)
{
    clReleaseMemObject(mem->buffer);
    mxFree(mem->data);
    mxFree(mem);
}
//
//OpenCL control flow:
//
void OpenCLManager::SetActiveGPU(unsigned int index)
{
    unsigned int DeviceIndex = 0;
    unsigned int PlatformIndex = 0;
    while (true)
    {
      if (PlatformIndex >= numPlatforms)
      {
        mexPrintf("Failed to set GPU: bad index\n");
        return;
      }
      if (index >= numDevices[PlatformIndex])
      {
        index -= numDevices[PlatformIndex];
        PlatformIndex++;
      }
      else
      {
```

```
    DeviceIndex = index;
    break;
  }
}

//find maker of card:
char platformname[64];
size_t platformnamesize = 64;
clGetPlatformInfo(vectorPlatforms[PlatformIndex], CL_PLATFORM_NAME, platformnamesize, platformname, NULL);
if ( (platformname[0] == 'A') && (platformname[1] == 'M') && (platformname[2] == 'D') )
{
  NVIDIA = false;
}

//Save current settings:
cl_int err;
platform = vectorPlatforms[PlatformIndex];
device = vectorDevices[PlatformIndex][DeviceIndex];

//create context:
context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);
#ifdef __DEBUG__
  if (err < 0)
  {
    std::cout << "Error creating program, code: " << err << std::endl;
  }
#endif

//Load all kernels:
program_strings = (char **)mxMalloc( sizeof(char *) * numSourceFiles);
mexMakeMemoryPersistent(program_strings);
program_sizes = (size_t *)mxMalloc( sizeof(size_t) * numSourceFiles);
mexMakeMemoryPersistent(program_sizes);
FILE * program_handle;

for (unsigned int i = 0; i < numSourceFiles; i += 1)
{
  program_handle = fopen(vectorSourceFiles[i], "r");
  fseek(program_handle, 0, SEEK_END);
  program_sizes[i] = ftell(program_handle);
  rewind(program_handle);
  program_strings[i] = (char *)mxMalloc(sizeof(char) * (program_sizes[i]+1));
```

```
    mexMakeMemoryPersistent(program_strings[i]);
    program_strings[i][program_sizes[i]] = '\0';
    fread(program_strings[i], sizeof(char), program_sizes[i], program_handle);
    fclose(program_handle);
  }
  //create program:
  program = clCreateProgramWithSource(context, numSourceFiles, (const char **)program_strings, program_sizes, &err);
  if (err < 0)
    mexPrintf("Fail...\n",err);
#ifdef __DEBUG__
    if (err < 0)
    {
      std::cout << "Error creating program, code: " << err << std::endl;
    }
#endif

  //build program:
  char * options;
  if (EnableDouble && NVIDIA)
  {
    options = "-D__DOUBLE_ALLOWED__ -IKernels -D__NVIDIA__"; //
  }
  else if (EnableDouble)
  {
    options = "-D__DOUBLE_ALLOWED__ -IKernels"; //
  }
  else
  {
    options = "-IKernels";
  }
  const char * optionsConst = options;
  err = clBuildProgram(program, 1, &device, optionsConst, NULL, NULL);
#ifdef __DEBUG__
    if (err < 0)
    {

      std::cout << "Error building program, code: ";
      WriteError(err);
    }
#endif

  //Program build info:
#ifdef __PROGRAM_BUILD_INFO__
    size_t program_log_size;
```

```
    clGetProgramBuildInfo(program, device,
        CL_PROGRAM_BUILD_LOG, 0, NULL, &program_log_size);
    char * program_log = (char *)mxMalloc(sizeof(char)*
        program_log_size);
    clGetProgramBuildInfo(program, device,
        CL_PROGRAM_BUILD_LOG, program_log_size, program_log,
        NULL);
    std::cout << program_log << std::endl;
#endif

  //create kernels:
  kernelReductionF = CreateKernel((char*)"VectorReductionF");
  kernelVectorTimesConstantFF = CreateKernel((char*)"
      VectorTimesConstantFF");
  kernelJacobiF = CreateKernel((char*)"JacobiMethodF");
  kernelJacobiMethodOddF = CreateKernel((char*)"
      JacobiMethodOddF");
  kernelJacobiMethodEvenF = CreateKernel((char*)"
      JacobiMethodEvenF");
  kernelJacobiDefectF = CreateKernel((char*)"JacobiCalcDefectF
      ");
  kernelRefineFTCFF = CreateKernel((char*)"RefineFTCFF");
  kernelRefineCTFFF = CreateKernel((char*)"RefineCTFFF");
  kernelVectorMinusVectorFF = CreateKernel((char*)"
      VectorMinusVectorFF");
  kernelVectorMinusVectorConstantFF = CreateKernel((char*)"
      VectorMinusVectorConstantFF");
  kernelNorm2F = CreateKernel((char*)"Norm2F");
  kernelNormInfF = CreateKernel((char*)"NormInfF");
  kernelVectorPlusVectorFF = CreateKernel((char*)"
      VectorPlusVectorFF");
   kernelReductionD = CreateKernel((char*)"VectorReductionD");
  kernelSparseMatrixVectorFF = CreateKernel((char*)"
      SparseMatrixVectorFF");
  kernelBandMatrixVectorFF = CreateKernel((char*)"
      BandMatrixVectorFF");
  if (EnableDouble)
  {
    kernelVectorTimesConstantFD = CreateKernel((char*)"
        VectorTimesConstantFD");
    kernelVectorTimesConstantDD = CreateKernel((char*)"
        VectorTimesConstantDD");
    kernelJacobiD = CreateKernel((char*)"JacobiMethodD");
    kernelJacobiMethodOddD = CreateKernel((char*)"
        JacobiMethodOddD");
    kernelJacobiMethodEvenD = CreateKernel((char*)"
        JacobiMethodEvenD");
    kernelJacobiDefectD = CreateKernel((char*)"
        JacobiCalcDefectD");
```

```
        kernelRefineFTCDF = CreateKernel((char*)"RefineFTCDF");
        kernelRefineFTCDD = CreateKernel((char*)"RefineFTCDD");
        kernelRefineCTFFD = CreateKernel((char*)"RefineCTFFD");
        kernelRefineCTFDD = CreateKernel((char*)"RefineCTFDD");
        kernelVectorMinusVectorFD = CreateKernel((char*)"VectorMinusVectorFD");
        kernelVectorMinusVectorDF = CreateKernel((char*)"VectorMinusVectorDF");
        kernelVectorMinusVectorDD = CreateKernel((char*)"VectorMinusVectorDD");
        kernelVectorMinusVectorConstantFD = CreateKernel((char*)"VectorMinusVectorConstantFD");
        kernelVectorMinusVectorConstantDF = CreateKernel((char*)"VectorMinusVectorConstantDF");
        kernelVectorMinusVectorConstantDD = CreateKernel((char*)"VectorMinusVectorConstantDD");
        kernelNorm2D = CreateKernel((char*)"Norm2D");
        kernelNormInfD = CreateKernel((char*)"NormInfD");
        kernelVectorPlusVectorFD = CreateKernel((char*)"VectorPlusVectorFF");
        kernelVectorPlusVectorDF = CreateKernel((char*)"VectorPlusVectorFF");
        kernelVectorPlusVectorDD = CreateKernel((char*)"VectorPlusVectorFF");
        kernelSparseMatrixVectorDF = CreateKernel((char*)"SparseMatrixVectorDF");
        kernelSparseMatrixVectorDD = CreateKernel((char*)"SparseMatrixVectorDD");
        kernelSparseMatrixVectorFD = CreateKernel((char*)"SparseMatrixVectorFD");
        kernelBandMatrixVectorFD = CreateKernel((char*)"BandMatrixVectorFD");
        kernelBandMatrixVectorDF = CreateKernel((char*)"BandMatrixVectorDF");
        kernelBandMatrixVectorDD = CreateKernel((char*)"BandMatrixVectorDD");
    }

    queue = clCreateCommandQueue(context, device,
        CL_QUEUE_PROFILING_ENABLE, &err);
    #ifdef __DEBUG__
        if (err < 0)
        {
            std::cout << "Error creating queue, code: " << err <<
                std::endl;
        }
    #endif
}
```

```

//
//Allocators:void
//
OpenCLManager::OpenCLManager()
{
   cl_int err;
   NVIDIA = true; //assume nvidia card.
   //Init variables:
   numSourceFiles = 0;
   numIndex = 0;
   numMemoryF = 0;
   numMemoryD = 0;
   capMemoryF = 1024;
   capMemoryD = 1024;
   capIndex = 1024;
   vectorSourceFiles = NULL;
   vectorIndex = (__IndexControl__**)mxMalloc(sizeof(
        __IndexControl__)*capIndex);
   vectorMemoryD = (__MemoryControl__<double> **)mxMalloc(
        sizeof(__MemoryControl__<double>)*capMemoryD);
   vectorMemoryF = (__MemoryControl__<float>  **)mxMalloc(
        sizeof(__MemoryControl__<float>)*capMemoryF);
   mexMakeMemoryPersistent(vectorIndex);
   mexMakeMemoryPersistent(vectorMemoryD);
   mexMakeMemoryPersistent(vectorMemoryF);
   EnableDouble = false;

   //Tuning stuff:
   SparseMatrixVectorRowsPerThread = 64;
   SparseMatrixVectorThreadsPerGroup = 64;
   BandMatrixVectorRowsPerThread = 64;
   BandMatrixVectorThreadsPerGroup = 64;
   NormRowsPerThread = 64;
   NormThreadsPerGroup = 64;
   VectorAndVectorRowsPerThread = 64;
   VectorAndVectorThreadsPerGroup = 64;
   VectorConstantRowsPerThread = 64;
   VectorConstantThreadsPerGroup = 64;
   JacobiRowsPerThread = 64;
   JacobiThreadsPerGroup = 64;
   RBGSRowsPerThread = 64;
   RBGSThreadsPerGroup = 64;
   DefectRowsPerThread = 64;
   DefectThreadsPerGroup = 64;
   FTCRowsPerThread = 64;
   FTCThreadsPerGroup = 64;
   CTFRowsPerThread = 64;
   CTFThreadsPerGroup = 64;
```

```

  //Init OpenCL Stuff:

  err = clGetPlatformIDs(0, NULL, &numPlatforms);
  vectorPlatforms = (cl_platform_id *)mxMalloc( sizeof(cl_platform_id) * numPlatforms);
  mexMakeMemoryPersistent(vectorPlatforms);
  err = clGetPlatformIDs(numPlatforms, vectorPlatforms, NULL);
  vectorDevices = (cl_device_id **)mxMalloc( sizeof(cl_device_id *) * numPlatforms);
  mexMakeMemoryPersistent(vectorDevices);
  numDevices = (unsigned int *)mxMalloc( sizeof(unsigned int) * numPlatforms);
  mexMakeMemoryPersistent(numDevices);

  for (unsigned int i = 0; i < numPlatforms; i += 1)
  {
    cl_uint testvar = 0;
    clGetDeviceIDs(vectorPlatforms[i], CL_DEVICE_TYPE_GPU, 0, NULL, &testvar);
    numDevices[i] = testvar;
    mexPrintf("number of devices: %u\n",numDevices[i]);
    vectorDevices[i] = (cl_device_id *)mxMalloc( sizeof(cl_device_id) * numDevices[i]);
    mexMakeMemoryPersistent(vectorDevices[i]);
    clGetDeviceIDs(vectorPlatforms[i], CL_DEVICE_TYPE_GPU, numDevices[i], vectorDevices[i], NULL);
  }
  AddSource((char*)"Kernels/VectorReduction.cl");
  AddSource((char*)"Kernels/BandMatrixVector.cl");
  AddSource((char*)"Kernels/SparseMatrixVector.cl");
  AddSource((char*)"Kernels/JacobiMethod.cl");
  AddSource((char*)"Kernels/JacobiMethodEven.cl");
  AddSource((char*)"Kernels/JacobiMethodOdd.cl");
  AddSource((char*)"Kernels/JacobiCalcDefect.cl");
  AddSource((char*)"Kernels/Norm2Smart.cl");
  AddSource((char*)"Kernels/NormInfSmart.cl");
  AddSource((char*)"Kernels/RefineFTC.cl");
  AddSource((char*)"Kernels/RefineCTF.cl");
  AddSource((char*)"Kernels/VectorTimesConstant.cl");
  AddSource((char*)"Kernels/VectorPlusVector.cl");
  AddSource((char*)"Kernels/VectorMinusVector.cl");
  AddSource((char*)"Kernels/VectorMinusVectorConstant.cl");
}



//
//Deallocators:
```

```
//
OpenCLManager::~OpenCLManager()
{
  ReleaseOpenCL();
}
void OpenCLManager::ReleaseOpenCL()
{
  #ifdef __DEBUG__
    unsigned int count = 0;
    for (unsigned int i = 0; i < numMemoryD; i += 1)
    {
      if (vectorMemoryD[i] != NULL)
        count++;
    }
    for (unsigned int i = 0; i < numMemoryF; i += 1)
    {
      if (vectorMemoryF[i] != NULL)
        count++;
    }
    mexPrintf("Number of memory spots assigned: %u, number of memory leaks: %u\n",numMemoryD+numMemoryF,count);
  #endif
  //Internal ordering:
  mxFree(vectorMemoryD);
  mxFree(vectorMemoryF);
  mxFree(vectorIndex);
  mxFree(vectorSourceFiles);
  //kernels:
  clReleaseKernel(kernelReductionF);
  clReleaseKernel(kernelJacobiMethodOddF);
  clReleaseKernel(kernelJacobiMethodEvenF);
  clReleaseKernel(kernelJacobiDefectF);
  clReleaseKernel(kernelRefineCTFFF);
  clReleaseKernel(kernelRefineFTCFF);
  clReleaseKernel(kernelVectorTimesConstantFF);
  clReleaseKernel(kernelVectorMinusVectorFF);
  clReleaseKernel(kernelVectorMinusVectorConstantFF);
  clReleaseKernel(kernelVectorPlusVectorFF);
  clReleaseKernel(kernelNormInfF);
  clReleaseKernel(kernelNorm2F);
  clReleaseKernel(kernelSparseMatrixVectorFF);

  if (EnableDouble)
  {
    clReleaseKernel(kernelReductionD);
    clReleaseKernel(kernelJacobiMethodOddD);
    clReleaseKernel(kernelJacobiMethodEvenD);
    clReleaseKernel(kernelJacobiDefectD);
    clReleaseKernel(kernelRefineCTFFD);
```

```
      clReleaseKernel(kernelRefineCTFDD);
      clReleaseKernel(kernelRefineFTCDF);
      clReleaseKernel(kernelRefineFTCDD);
      clReleaseKernel(kernelVectorTimesConstantFD);
      clReleaseKernel(kernelVectorTimesConstantDD);
      clReleaseKernel(kernelVectorMinusVectorFD);
      clReleaseKernel(kernelVectorMinusVectorDF);
      clReleaseKernel(kernelVectorMinusVectorDD);
      clReleaseKernel(kernelVectorMinusVectorConstantFD);
      clReleaseKernel(kernelVectorMinusVectorConstantDF);
      clReleaseKernel(kernelVectorMinusVectorConstantDD);
      clReleaseKernel(kernelVectorPlusVectorFD);
      clReleaseKernel(kernelVectorPlusVectorDF);
      clReleaseKernel(kernelVectorPlusVectorDD);
      clReleaseKernel(kernelNormInfD);
      clReleaseKernel(kernelNorm2D);
      clReleaseKernel(kernelSparseMatrixVectorDD);
      clReleaseKernel(kernelSparseMatrixVectorFD);
      clReleaseKernel(kernelSparseMatrixVectorDF);
    }
    //clean up:
    clReleaseCommandQueue(queue);
    clReleaseProgram(program);
    clReleaseContext(context);

    //clean up rest:
    mxFree(vectorPlatforms);
    for (unsigned int i = 0; i < numPlatforms; i++)
    {
      mxFree(vectorDevices[i]);
    }
    mxFree(vectorDevices);
    mxFree(numDevices);
    mxFree(program_strings);
    mxFree(program_sizes);
  }

  //Shortcuts:
  cl_kernel OpenCLManager::CreateKernel(char * name)
  {
    cl_int err;
    cl_kernel temp = clCreateKernel(program, name, &err);
    #ifdef __DEBUG__
      if (err < 0)
      {
        mexPrintf("Failed to create kernel %s\n",name);
      }
    #endif
    return temp;
```

```
}

void OpenCLManager::AllowDouble()
{
  EnableDouble = true;
}

void OpenCLManager::WaitForCPU()
{
  clFinish(queue);
}

float OpenCLManager::GetExecutionTime(cl_event * event)
{
  cl_ulong start, end;
  clGetEventProfilingInfo(*event, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &end, NULL);
  clGetEventProfilingInfo(*event, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &start, NULL);

  return (end - start) * 1.0e-3f;
}

//Autotuning functions:
void OpenCLManager::SetSparseMatrixVectorRowsPerThread(size_t newvalue)
{
  SparseMatrixVectorRowsPerThread = newvalue;
}
void OpenCLManager::SetSparseMatrixVectorThreadsPerGroup(size_t newvalue)
{
  SparseMatrixVectorThreadsPerGroup = newvalue;
}
void OpenCLManager::SetBandMatrixVectorRowsPerThread(size_t newvalue)
{
  BandMatrixVectorRowsPerThread = newvalue;
}
void OpenCLManager::SetBandMatrixVectorThreadsPerGroup(size_t newvalue)
{
  BandMatrixVectorThreadsPerGroup = newvalue;
}
void OpenCLManager::SetNormRowsPerThread(size_t newvalue)
{
  NormRowsPerThread = newvalue;
}
void OpenCLManager::SetNormThreadsPerGroup(size_t newvalue)
```

```
{
  NormThreadsPerGroup = newvalue;
}
void OpenCLManager::SetVectorAndVectorRowsPerThread(size_t
    newvalue)
{
  VectorAndVectorRowsPerThread = newvalue;
}
void OpenCLManager::SetVectorAndVectorThreadsPerGroup(size_t
    newvalue)
{
  VectorAndVectorThreadsPerGroup = newvalue;
}
void OpenCLManager::SetVectorConstantRowsPerThread(size_t
    newvalue)
{
  VectorConstantRowsPerThread = newvalue;
}
void OpenCLManager::SetVectorConstantThreadsPerGroup(size_t
    newvalue)
{
  VectorConstantThreadsPerGroup = newvalue;
}
void OpenCLManager::SetJacobiRowsPerThread(size_t newvalue)
{
  JacobiRowsPerThread = newvalue;
}
void OpenCLManager::SetJacobiThreadsPerGroup(size_t newvalue)
{
  JacobiThreadsPerGroup = newvalue;
}
void OpenCLManager::SetRBGSRowsPerThread(size_t newvalue)
{
  RBGSRowsPerThread = newvalue;
}
void OpenCLManager::SetRBGSThreadsPerGroup(size_t newvalue)
{
  RBGSThreadsPerGroup = newvalue;
}
void OpenCLManager::SetDefectRowsPerThread(size_t newvalue)
{
  DefectRowsPerThread = newvalue;
}
void OpenCLManager::SetDefectThreadsPerGroup(size_t newvalue)
{
  DefectThreadsPerGroup = newvalue;
}
void OpenCLManager::SetFTCRowsPerThread(size_t newvalue)
{
```

```
  FTCRowsPerThread = newvalue;
}
void OpenCLManager::SetFTCThreadsPerGroup(size_t newvalue)
{
  FTCThreadsPerGroup = newvalue;
}
void OpenCLManager::SetCTFRowsPerThread(size_t newvalue)
{
  CTFRowsPerThread = newvalue;
}
void OpenCLManager::SetCTFThreadsPerGroup(size_t newvalue)
{
  CTFThreadsPerGroup = newvalue;
}

void OpenCLManager::WriteError(cl_int err)
{
  switch (err)
  {
  case CL_INVALID_COMMAND_QUEUE:
    mexPrintf("invalid command queue.\n");
    break;
  case CL_INVALID_KERNEL:
    mexPrintf("invalid kernel.\n");
    break;
  case CL_INVALID_CONTEXT:
    mexPrintf("invalid context.\n");
    break;
  case CL_INVALID_KERNEL_ARGS:
    mexPrintf("invalid kernel arguments.\n");
    break;
  case CL_INVALID_WORK_DIMENSION:
    mexPrintf("invalid work dimension.\n");
    break;
  case CL_INVALID_WORK_GROUP_SIZE:
    mexPrintf("invalid work group size.\n");
    break;
  case CL_INVALID_WORK_ITEM_SIZE:
    mexPrintf("invalid work item size.\n");
    break;
  case CL_INVALID_GLOBAL_OFFSET:
    mexPrintf("invalid global offset.\n");
    break;
  case CL_OUT_OF_RESOURCES:
    mexPrintf("out of resources.\n");
    break;
  case CL_MEM_OBJECT_ALLOCATION_FAILURE:
    mexPrintf("failed to allocate memory object.\n");
    break;
```

```
    case CL_INVALID_EVENT_WAIT_LIST:
      mexPrintf("invalid event wait list.\n");
      break;
    case CL_OUT_OF_HOST_MEMORY:
      mexPrintf("out of host memory.\n");
      break;
    }
}
```

10.3.3 OpenCLHigh.h

```
#include "preprocessor.h"
#include "OpenCLManager.h"
#include "MathVector.h"
#include <cmath>
#include "mex.h"

#ifndef __OPENCLHIGH_H__
#define __OPENCLHIGH_H__

//Helper functions to simulate templates for c functionality:
void HelpNormInf(OpenCLManager * __OpenCLManager__, MathVector<float> * input, MathVector<float> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);
void HelpNormInf(OpenCLManager * __OpenCLManager__, MathVector<double> * input, MathVector<double> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);
void HelpSum(OpenCLManager * __OpenCLManager__, MathVector<float> * input, MathVector<float> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);
void HelpSum(OpenCLManager * __OpenCLManager__, MathVector<double> * input, MathVector<double> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);
void HelpNorm2(OpenCLManager * __OpenCLManager__, MathVector<float> * input, MathVector<float> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);
void HelpNorm2(OpenCLManager * __OpenCLManager__, MathVector<double> * input, MathVector<double> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);

//Helper functions for C++ api:
template <class T> T NormInf(MathVector<T> & input, unsigned int threadsize, float * time);
template <class T> T NormInf(MathVector<T> & input, float * time);
```

```
template <class T> T Norm2(MathVector<T> & input, unsigned int threadsize, float * time);
template <class T> T Norm2(MathVector<T> & input, float * time);
template <class T> T sum(MathVector<T> & input, unsigned int threadsize, float * time);
template <class T> T sum(MathVector<T> & input, float * time);


//Norms and sum:
template <class T> T NormInf(OpenCLManager * __OpenCLManager__, MathVector<T> & input, unsigned int threadsize, float * time);
template <class T> T Norm2(OpenCLManager * __OpenCLManager__, MathVector<T> & input, unsigned int threadsize, float * time);
template <class T> T sum(OpenCLManager * __OpenCLManager__, MathVector<T> & input, unsigned int threadsize, float * time);





//
//HELPER FUNCTIONS:
//
void HelpNormInf(OpenCLManager * __OpenCLManager__, MathVector<float> * input, MathVector<float> * output, unsigned int threadsize, unsigned int remLength, cl_event * event)
{
  __OpenCLManager__->NormInfF(input->GetDataBuffer(), output->GetDataBuffer(), threadsize, remLength, event);
}
void HelpNormInf(OpenCLManager * __OpenCLManager__, MathVector<double> * input, MathVector<double> * output, unsigned int threadsize, unsigned int remLength, cl_event * event)
{
  __OpenCLManager__->NormInfD(input->GetDataBuffer(), output->GetDataBuffer(), threadsize, remLength, event);
}
void HelpSum(OpenCLManager * __OpenCLManager__, MathVector<float> * input, MathVector<float> * output, unsigned int threadsize, unsigned int remLength, cl_event * event)
{
  __OpenCLManager__->ParallelSumReductionF(input->GetDataBuffer(), output->GetDataBuffer(), threadsize, remLength, event);
}
```

```
void HelpSum(OpenCLManager * __OpenCLManager__, MathVector<
    double> * input, MathVector<double> * output, unsigned int
     threadsize, unsigned int remLength, cl_event * event)
{
  __OpenCLManager__->ParallelSumReductionD(input->
      GetDataBuffer(), output->GetDataBuffer(), threadsize,
      remLength, event);
}
void HelpNorm2(OpenCLManager * __OpenCLManager__, MathVector<
    float> * input, MathVector<float> * output, unsigned int
    threadsize, unsigned int remLength, cl_event * event)
{
  __OpenCLManager__->Norm2F(input->GetDataBuffer(), output->
      GetDataBuffer(), threadsize, remLength, event);
}
void HelpNorm2(OpenCLManager * __OpenCLManager__, MathVector<
    double> * input, MathVector<double> * output, unsigned int
     threadsize, unsigned int remLength, cl_event * event)
{
  __OpenCLManager__->Norm2D(input->GetDataBuffer(), output->
      GetDataBuffer(), threadsize, remLength, event);
}

//
//C++ API:
//


//
//NORMS:
//
template <class T> T NormInf(OpenCLManager * __OpenCLManager__
    , MathVector<T> & input, unsigned int threadsize, float *
    time)
{
  *time = 0.0f;
   cl_event event;
  //Declare variables:
  T result = 0.0;
  unsigned int numIter = 0;
  unsigned int remWork = input.GetLength();
  //Find the number of iterations required:
  while (remWork > 1)
  {
    if (remWork % (threadsize*threadsize) == 0)
      remWork /= (threadsize*threadsize);
    else
      remWork = remWork/(threadsize*threadsize)+1;
    numIter++;
```

```
    }
    MathVector<T> * pInput, * pOut;
    pInput = &input;

    remWork = input.GetLength();
    unsigned int remLength = remWork;
    for (unsigned int i = 0; i < numIter; i += 1)
    {
        //Calculate the remaining work:
        if (remWork % (threadsize*threadsize) == 0)
            remWork /= (threadsize*threadsize);
        else
            remWork = remWork/(threadsize*threadsize)+1;
        //if at last iteration, create array for storage, otherwise keep on GPU:
        if (remWork == 1)
            pOut = new MathVector<T>(__OpenCLManager__, remWork, (T*)NULL);
        else
            pOut = new MathVector<T>(__OpenCLManager__, remWork);


        //Call the parallel sum reduction routine:
        HelpNormInf(__OpenCLManager__, pInput, pOut, threadsize, remLength, &event);
        remLength = remWork;
        *time += __OpenCLManager__->GetExecutionTime(&event);

        //Reset pointers:
        if (pInput != &input)
        {
            delete pInput;
        }
        pInput = pOut;

        //If at last iteration, swap buffers and delete temp buffer:
        if (remWork == 1)
        {
            __OpenCLManager__->SwapGPUBufferData(pOut->GetDataBuffer(), pOut->GetData(), remWork, sizeof(T));
            result = *(pOut->GetData());
            delete pOut;
        }
    }
    return result;
}

```

```
template <class T> T Norm2(OpenCLManager * __OpenCLManager__,
    MathVector<T> & input, unsigned int threadsize, float *
    time)
{
  //Declare variables:
  *time = 0.0f;
  cl_event event;
  T result = 0.0;
  unsigned int numIter = 0;
  unsigned int remWork = input.GetLength();

  //Find the number of iterations required:
  while (remWork > 1)
  {
    if (remWork % (threadsize*threadsize) == 0)
      remWork /= (threadsize*threadsize);
    else
      remWork = remWork/(threadsize*threadsize)+1;
    numIter++;
  }
  MathVector<T> * pInput, * pOut;
  pInput = &input;

  remWork = input.GetLength();
  unsigned int remLength = remWork;
  for (unsigned int i = 0; i < numIter; i += 1)
  {
    //Calculate the remaining work:
    if (remWork % (threadsize*threadsize) == 0)
      remWork /= (threadsize*threadsize);
    else
      remWork = remWork/(threadsize*threadsize)+1;
    //if at last iteration, create array for storage,
        otherwise keep on GPU:
    if (remWork == 1)
      pOut = new MathVector<T>(__OpenCLManager__, remWork, (T
          *)NULL);
    else
      pOut = new MathVector<T>(__OpenCLManager__, remWork);


    //Call the parallel sum reduction routine:
    if (i ==0)
      HelpNorm2(__OpenCLManager__, pInput, pOut, threadsize,
          remLength, &event);
    else
      HelpSum(__OpenCLManager__, pInput, pOut, threadsize,
          remLength, &event);
    remLength = remWork;
```

```
    *time += __OpenCLManager__->GetExecutionTime(&event);

    //Reset pointers:
    if (pInput != &input)
    {
      delete pInput;
    }
    pInput = pOut;

    //If at last iteration, swap buffers and delete temp
        buffer:
    if (remWork == 1)
    {
      __OpenCLManager__->SwapGPUBufferData(pOut->GetDataBuffer
          (), pOut->GetData(), remWork, sizeof(T));
      result = *(pOut->GetData());
      delete pOut;
    }
  }
  return sqrt(result);
}

template <class T> T sum(OpenCLManager * __OpenCLManager__,
    MathVector<T> & input, unsigned int threadsize, float *
    time)
{
  *time = 0.0f;
  cl_event event;
  //Declare variables:
  T result = 0.0;
  unsigned int numIter = 0;
  unsigned int remWork = input.GetLength();
  //Find the number of iterations required:
  while (remWork > 1)
  {
    if (remWork % (threadsize*threadsize) == 0)
      remWork /= (threadsize*threadsize);
    else
      remWork = remWork/(threadsize*threadsize)+1;
    numIter++;
  }
  MathVector<T> * pInput, * pOut;
  pInput = &input;
  remWork = input.GetLength();
  unsigned int remLength = remWork;
  for (unsigned int i = 0; i < numIter; i += 1)
  {
    //Calculate the remaining work:
    if (remWork % (threadsize*threadsize) == 0)
```

```
        remWork /= (threadsize*threadsize);
      else
        remWork = remWork/(threadsize*threadsize)+1;

      //if at last iteration, create array for storage,
         otherwise keep on GPU:
      if (remWork == 1)
        pOut = new MathVector<T>(__OpenCLManager__, remWork, (T
           *)NULL);
      else
        pOut = new MathVector<T>(__OpenCLManager__, remWork);


      //Call the parallel sum reduction routine:
      HelpSum(__OpenCLManager__, (MathVector<T> *)pInput, (
         MathVector<T> *)pOut, threadsize, remLength, &event);
      remLength = remWork;
      *time += __OpenCLManager__->GetExecutionTime(&event);

      //Reset pointers:
      if (pInput != &input)
      {
        delete pInput;
      }
      pInput = pOut;

      //If at last iteration, swap buffers and delete temp
         buffer:
      if (remWork == 1)
      {
        __OpenCLManager__->SwapGPUBufferData(pOut->GetDataBuffer
           (), pOut->GetData(), remWork, sizeof(T));
        result = *(pOut->GetData());
        delete pOut;
      }
    }
    return result;
}


#endif
```

10.3.4 MemoryControl.h

```
#include "preprocessor.h"
#include <vector>
```

```

#ifndef __MEMORYCONTROL_H__
#define __MEMORYCONTROL_H__

template <class T>
struct __MemoryControl__
{
public:
  T * data;
  cl_mem buffer;
  unsigned int size;
  int RefCount;
  bool BuffersMatch;
};

struct __IndexControl__
{
public:
  unsigned int * data;
  cl_mem buffer;
  unsigned int size;
  int RefCount;
  bool BuffersMatch;
};

#endif /* __MEMORYCONTROL_H__ */
```

10.3.5 MathVector.h

```
#ifndef __MATHVECTOR_H__
#define __MATHVECTOR_H__

#include "MemoryControl.h"
#include "preprocessor.h"

class ClassController
{
  public:
    int Type;
};

template <class T>
class MathVector : public ClassController
{
  public:
    MathVector ();
```

```
        MathVector (OpenCLManager *, unsigned int);
        MathVector (OpenCLManager *, unsigned int,T * real);
        void Init(OpenCLManager *, unsigned int);
        void Init(OpenCLManager *, unsigned int, T * real);
        ¬MathVector ();
        cl_mem & GetDataBuffer();
        unsigned int GetLength();
        void SyncBuffers();
        T * GetData();
        __MemoryControl__<T> * GetMemoryControl();
        void SetMemoryControl(__MemoryControl__<T> *);
    //operators:
        MathVector * operator*(T element);
        MathVector * operator+(MathVector<T> & other);
    private:
        OpenCLManager * __OpenCLManager__;
        unsigned int length;
        __MemoryControl__<T> * memory;
};


#include "OpenCLHigh.h"
#include "OpenCLManager.h"



//
//Constructors:
//
template <class T>
MathVector<T>::MathVector ()
{
}
template <class T>
MathVector<T>::MathVector (OpenCLManager * __OpenCLManager__,
     unsigned int len)
{
    Init(__OpenCLManager__, len);
}
template <class T>
MathVector<T>::MathVector (OpenCLManager * __OpenCLManager__,
     unsigned int len, T * real)
{
    Init(__OpenCLManager__, len, real);
}
template <class T>
void MathVector<T>::Init (OpenCLManager * __OpenCLManager__,
     unsigned int len)
{
```

```
    length = len;
    memory = __OpenCLManager__->AllocateMemory((T*)NULL, len);
    this->__OpenCLManager__ = __OpenCLManager__;
}
template <class T>
void MathVector<T>::Init (OpenCLManager * __OpenCLManager__,
      unsigned int len, T * real)
{
    length = len;
    memory = __OpenCLManager__->AllocateMemory(real, len);
    this->__OpenCLManager__ = __OpenCLManager__;
}

//
//Destructors:
//
template <class T>
MathVector<T>::¬MathVector ()
{
    __OpenCLManager__->DeleteMemory(memory);
}

//
//Methods:
//
template <class T>
cl_mem & MathVector<T>::GetDataBuffer()
{
    return memory->buffer;
}
template <class T>
unsigned int MathVector<T>::GetLength()
{
    return length;
}
template <class T>
T * MathVector<T>::GetData()
{
    return memory->data;
}

//
//Buffer sync'ing:
//
template <class T>
void MathVector<T>::SyncBuffers()
{
    __OpenCLManager__->SwapGPUBufferData( memory->buffer, memory
        ->data, length, sizeof(T)); /*
```

```
    if (memory->BuffersMatch == false) //obsolete code from c
        time.
    {

      memory->BuffersMatch = true;
    } */
}

template <class T>
__MemoryControl__<T> * MathVector<T>::GetMemoryControl()
{
    return memory;
}

template <class T>
void MathVector<T>::SetMemoryControl(__MemoryControl__<T> *
    newMem)
{
    memory = newMem;
}

#endif /* __MATHVECTOR_H__ */
```

10.3.6 Matrix.h

```
#include "MemoryControl.h"
#include "preprocessor.h"
#include "MathVector.h"

#ifndef __MATRIX_H__
#define __MATRIX_H__


template <class T>
class Matrix : public MathVector<T>
{
    public:
      Matrix(OpenCLManager *, unsigned int, unsigned int);
      Matrix(OpenCLManager *, unsigned int, unsigned int, T *
          real);
      void InitMatrix(OpenCLManager *, unsigned int, unsigned
          int);
      void InitMatrix(OpenCLManager *, unsigned int, unsigned
          int, T * real);
      ~Matrix();
      unsigned int GetWidth();
```

```
    unsigned int GetHeight();
  private:
    unsigned int width, height;
};


//
//Constructors:
//
template <class T>
Matrix<T>::Matrix (OpenCLManager * __OpenCLManager__, unsigned
     int width, unsigned int height)
{
  InitMatrix(__OpenCLManager__, width, height);
}
template <class T>
Matrix<T>::Matrix (OpenCLManager * __OpenCLManager__, unsigned
     int width, unsigned int height, T * real)
{
  InitMatrix(__OpenCLManager__, width, height);
}
template <class T>
void Matrix<T>::InitMatrix (OpenCLManager * __OpenCLManager__,
     unsigned int width, unsigned int height)
{
  this->width = width;
  this->height = height;
  Init(__OpenCLManager__, width*height,(T *)NULL);
}
template <class T>
void Matrix<T>::InitMatrix (OpenCLManager * __OpenCLManager__,
     unsigned int width, unsigned int height, T * real)
{
  this->width = width;
  this->height = height;
  Init(__OpenCLManager__, width*height, real);
}

//
//Getters:
//
template <class T>
unsigned int Matrix<T>::GetWidth()
{
  return width;
}
template <class T>
unsigned int Matrix<T>::GetHeight()
{
```

```
  return height;
}

//
//Destructor:
//
template <class T>
Matrix<T>::¬Matrix ()
{
}

#endif /* __MATRIX_H__ */
```

10.3.7 BandMatrix.h

```
#include "preprocessor.h"
#include "MemoryControl.h"

#ifndef BANDMATRIX_H
#define BANDMATRIX_H
template <class T>
class BandMatrix : public MathVector<T>
{
  public:
    BandMatrix(OpenCLManager * __OpenCLManager__, unsigned int
         height, unsigned int width, unsigned int bandwidth, T
         * data);
    BandMatrix(OpenCLManager * __OpenCLManager__, unsigned int
         height, unsigned int width, unsigned int bandwidth);
    ¬BandMatrix ();
    void InitMatrix(OpenCLManager * __OpenCLManager__,
       unsigned int height, unsigned int width, unsigned int
       bandwidth, T * data);
    unsigned int GetBandwidth();
    unsigned int GetHeight();
    unsigned int GetWidth();
  private:
    unsigned int height;
    unsigned int width;
    unsigned int bandwidth;
    __MemoryControl__<T> * memory;
    void SortVector( T * & data, unsigned int length, unsigned
         int bandwidth);
};

template <class T>
```

```
BandMatrix<T>::BandMatrix(OpenCLManager * __OpenCLManager__,
    unsigned int height, unsigned int width, unsigned int
    bandwidth, T * data)
{
  InitMatrix(__OpenCLManager__, height, width, bandwidth, data
    );
}

template <class T>
void BandMatrix<T>::InitMatrix(OpenCLManager *
    __OpenCLManager__, unsigned int height, unsigned int width
    , unsigned int bandwidth, T * data)
{
  this->height = height;
  this->width = width;
  this->bandwidth = bandwidth;

  //determine length:
  SortVector(data, height, bandwidth);

  Init(__OpenCLManager__, height*(1+2*bandwidth), data);
}

template <class T>
unsigned int BandMatrix<T>::GetWidth()
{
  return width;
}
template <class T>
unsigned int BandMatrix<T>::GetHeight()
{
  return height;
}
template <class T>
unsigned int BandMatrix<T>::GetBandwidth()
{
  return bandwidth;
}
template <class T>
void BandMatrix<T>::SortVector( T * & data, unsigned int
    length, unsigned int bandwidth)
{
  T * newData = (T*)mxMalloc(sizeof(T)*length*(1 + 2*bandwidth
    ));

  unsigned int index = 0;
  int low, high;
  for (int i = 0; i < length; i += 1)
  {
```

```
      for (int j = 0; j < 1+2*bandwidth; j += 1)
      {
        newData[i*(1+2*bandwidth) + j] = data[i + j*length];
      }
    }
    data = newData;
}
#endif
```

10.3.8 SparseMatrix.h

```
#include "preprocessor.h"
#include "MemoryControl.h"
#include "MathVector.h"

#ifndef SPARSEMATRIX_H
#define SPARSEMATRIX_H
template <class T>
class SparseMatrix : public MathVector<T>
{
  public:
    SparseMatrix(OpenCLManager * __OpenCLManager__, unsigned
        int height, unsigned int width, unsigned int length,
        size_t * rows, size_t * columns, T * data);
    ¬SparseMatrix ();
    void InitMatrix(OpenCLManager * __OpenCLManager__,
        unsigned int height, unsigned int width, unsigned int
        length, size_t * rows, size_t * columns, T * data);
    int * GetColumnIndexes();
    int * GetRowIndexes();
    unsigned int GetHeight();
    unsigned int GetWidth();
    unsigned int GetRowVectorLength();
    cl_mem & GetRowIndexBuffer();
    cl_mem & GetColumnIndexBuffer();
    __IndexControl__ * GetRowControl();
    __IndexControl__ * GetColumnControl();
  private:
    unsigned int height;
    unsigned int width;
    __IndexControl__ * column;
    __IndexControl__ * row;
    unsigned int SortVectors(unsigned int length, unsigned * &
         rows, unsigned * & columns, T * & data);
};
template <class T>
```

```
SparseMatrix<T>::SparseMatrix(OpenCLManager * __OpenCLManager__, unsigned int height, unsigned int width, unsigned int length, size_t * rows, size_t * columns, T * data)
{
  InitMatrix(__OpenCLManager__, height, width, length, rows, columns, data);
}
template <class T>
void SparseMatrix<T>::InitMatrix(OpenCLManager * __OpenCLManager__, unsigned int height, unsigned int width, unsigned int length, size_t * rows, size_t * columns, T * data)
{
  this->height = height;
  this->width = width;
  unsigned int * columnsNew = (unsigned int *)mxMalloc(sizeof(unsigned int)*length);
  unsigned int * rowsNew = (unsigned int *)mxMalloc(sizeof(unsigned int)*length);
  unsigned int currentCol = 0;
  unsigned int index = 1;
  unsigned int low  = columns[0];
  unsigned int high = columns[1];
  while (true)
  {
    for (unsigned int i = low; i < high; i++)
    {
      columnsNew[i] = currentCol;
      rowsNew[i] = rows[i];
    }
    if (high >= length)
      break;
    currentCol++;
    low = high;
    index++;
    high = columns[index];
  }
  unsigned int rowLength = SortVectors(length,rowsNew,columnsNew,data);
  Init(__OpenCLManager__, length, data);
  column = __OpenCLManager__->AllocateIndex(columnsNew, length);
  row = __OpenCLManager__->AllocateIndex(rowsNew, rowLength+1);
}
template <class T>
unsigned int SparseMatrix<T>::GetWidth()
{
```

```
  return width;
}
template <class T>
unsigned int SparseMatrix<T>::GetHeight()
{
  return height;
}
template <class T>
int * SparseMatrix<T>::GetColumnIndexes()
{
  return column->data;
}
template <class T>
int * SparseMatrix<T>::GetRowIndexes()
{
  return row->data;
}
template <class T>
unsigned int SparseMatrix<T>::SortVectors(unsigned int length,
    unsigned * & rows, unsigned * & columns, T * & data)
{
  int i = 0;
  int temp;
  T tempData;
  //Sort arrays:
  while (i < length-1)
  {
    if ((rows[i] > rows[i+1]) || ((rows[i] == rows[i+1]) && (
        columns[i] > columns[i+1])))
    {
      temp = rows[i+1];
      rows[i+1] = rows[i];
      rows[i] = temp;
      temp = columns[i+1];
      columns[i+1] = columns[i];
      columns[i] = temp;
      tempData = data[i+1];
      data[i+1] = data[i];
      data[i] = tempData;
      if (i > 0)
      {
        i--;
      }
    }
    else
    {
      i++;
    }
  }
```

```
  unsigned int rowLength=height;

  //Declare new array:
  unsigned int * newRows = (unsigned int *)mxMalloc(sizeof(
      unsigned int)*(rowLength+1));

unsigned int currentcount = 0;
newRows[0] = 0;
for (i = 0; i < rowLength; i++)
{
  while(currentcount < length)
  {
    if (rows[currentcount]==i)
    {
      currentcount++;
    }
    else
    {
      newRows[i+1] = currentcount;
      break;
    }
  }
  if (currentcount == length)
  {
    newRows[i+1] = length;
  }
}

  rows = newRows;
  return rowLength;
}
template <class T>
cl_mem & SparseMatrix<T>::GetColumnIndexBuffer()
{
  return column->buffer;
}
template <class T>
cl_mem & SparseMatrix<T>::GetRowIndexBuffer()
{
  return row->buffer;
}

template <class T>
__IndexControl__ * SparseMatrix<T>::GetRowControl()
{
  return row;
}

template <class T>
```

```
__IndexControl__ * SparseMatrix<T>::GetColumnControl()
{
  return column;
}
template <class T>
unsigned int SparseMatrix<T>::GetRowVectorLength()
{
  return row->size;
}

#endif
```

10.3.9 preprocessor.h

```
#ifndef PREPROCESSOR_H
#define PREPROCESSOR_H

#define __DEBUG__
#define __PROGRAM_BUILD_INFO__ "build_info.txt"
//#pragma OPENCL EXTENSION cl_khr_fp64 : enable
//#define USE_DOUBLE_PRECISION

//Define standard libraries used:
#include <string>
#include <iostream>
#include <vector>
#ifdef MAC
#include <OpenCL/cl.h>
#else
#include <CL/cl.h>
#endif


//Define commands for handling errors:
#ifdef __DEBUG__

#endif


#endif
```