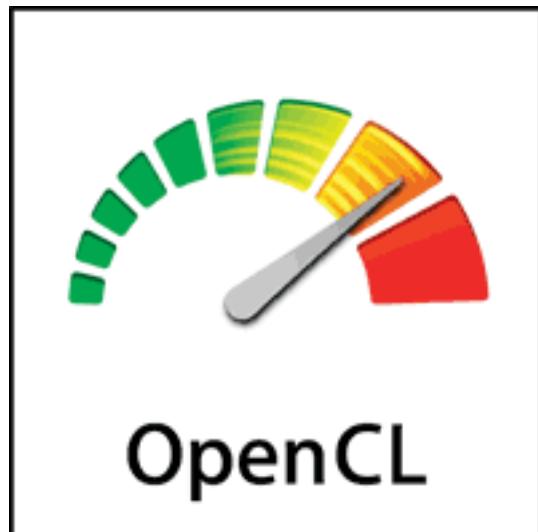


Bachelor-thesis: GPU-Acceleration of Linear Algebra using OpenCL

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 parallelizable.

For ease of use and more rapid prototyping, 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 transferred 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 transferred 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 usefulness 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 simultaneously. 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's¹ 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, either 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 CUDA². 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.

¹Graphical Processing Unit

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

This can most likely be attributed 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 transferred 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 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 possible.

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 noting 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 API³ 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 neccesary 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 MEX⁴ 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 sacrificing efficiency. For

³Application Programming Interface

⁴MATLAB Executable

the actual OpenCL code, the c bindings were chosen as opposed to the c++ bindings. The reasoning 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 introduced, 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 address 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 either 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. Initig 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.

```
1 %get opencl handle
2 gpu = MexInitOpenCL() ;
3
4 %find GPU's :
5 MexPrintGPU(gpu) ;
6
7 %choose gpu to use: (gpu 0) and enable doubles .
8 MexSetGPU(gpu,0, 1)
9
10 %create matrices :
11 A = zeros(3,1) ;
```

```
12 B = sparse([1,2,3],[1,2,3],[1,2,3]);  
13  
14 %move matrices to GPU:  
15 AHandle = MexMatrix(gpu, A);  
16 BHandle = MexSparseMatrix(gpu, B);  
17  
18 %create single precision matrices:  
19 C = single(A);  
20  
21 %create single precision matrices on GPU:  
22 CHandle = MexMatrix(gpu, C);  
23 DHandle = MexSparseMatrix(gpu, B, 1);  
24  
25 %perform calculations:  
26 outDouble = Mexmatrix(gpu, A);  
27 outSingle = MexMatrix(gpu, C);  
28  
29 %sparse matrix times vector for both single and double  
precision:  
30 MexSparseMatrixVector(gpu, BHandle, AHandle, outDouble);  
31 MexSparseMatrixVector(gpu, DHandle, CHandle, outSingle);  
32  
33 %obtain results for use in MATLAB:  
34 A = MexHandleMatrix(gpu, outDouble);  
35 C = MexHandleMatrix(gpu, outSingle);  
36  
37 %clean up:  
38 MexReleaseSparseMatrix(gpu, BHandle);  
39 MexReleaseSparseMatrix(gpu, DHandle);  
40 MexReleaseMatrix(gpu, AHandle);  
41 MexReleaseMatrix(gpu, CHandle);  
42 MexReleaseMatrix(gpu, outDouble);  
43 MexReleaseMatrix(gpu, outSingle);  
44  
45 %release opencl:  
46 MexReleaseOpenCL(gpu);
```

A noteworthy thing regarding the AMD implementation of OpenCL on Ubuntu using MATLAB is that MATLAB resets the DISPLAY environment 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 mathematical 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:

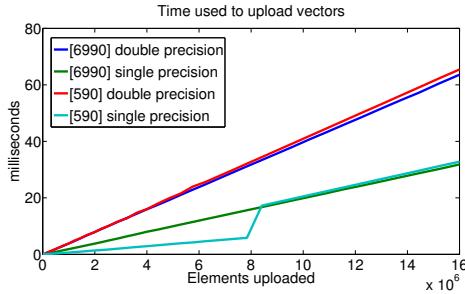


Figure 1: Test of uploading speeds

We observe a linear tendency, which makes sense, as the data is transferred continuously to the GPU memory. As we observe, transferring 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:

$$V_{val} = [1 \ 3 \ 2 \ 4 \ 6 \ 5], \ V_{col} = [1 \ 4 \ 2 \ 3 \ 4 \ 2], \ V_{row} = [1 \ 3 \ 6 \ 6 \ 7]$$

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:

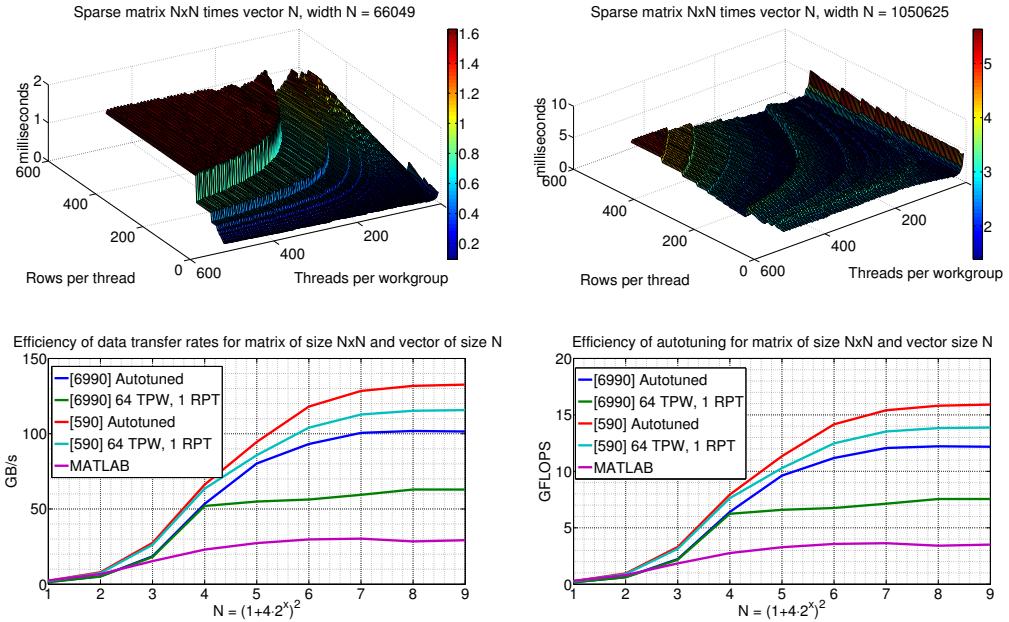


Figure 2: Demonstration of autotuning and performance calculation for a double precision sparse matrix with 6 non-zero values per row.

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 simultaneously. 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.

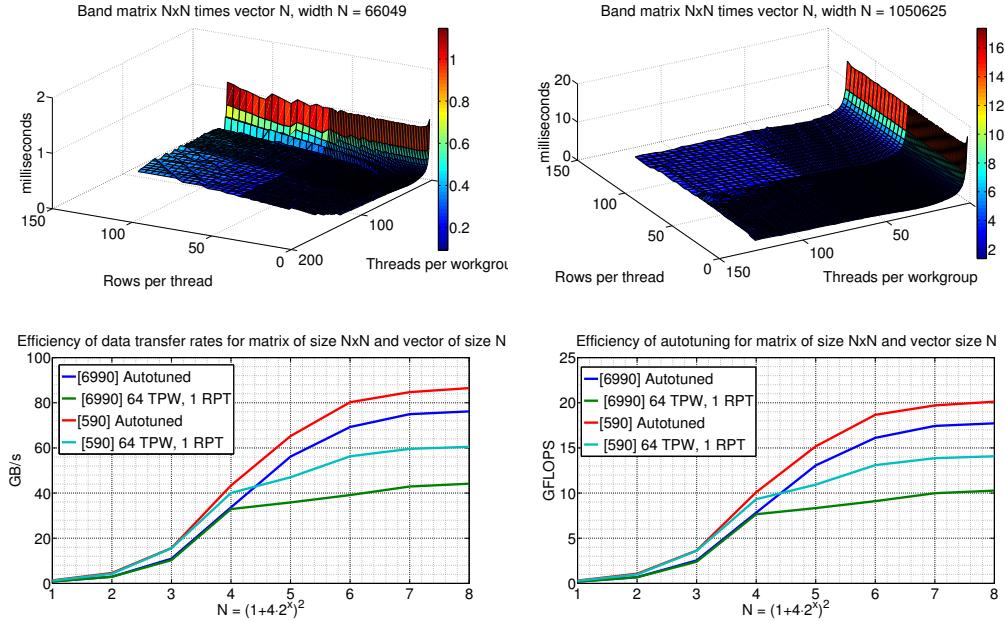


Figure 3: Demonstration of autotuning and performance calculation for a double precision band matrix with bandwidth 3.

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.

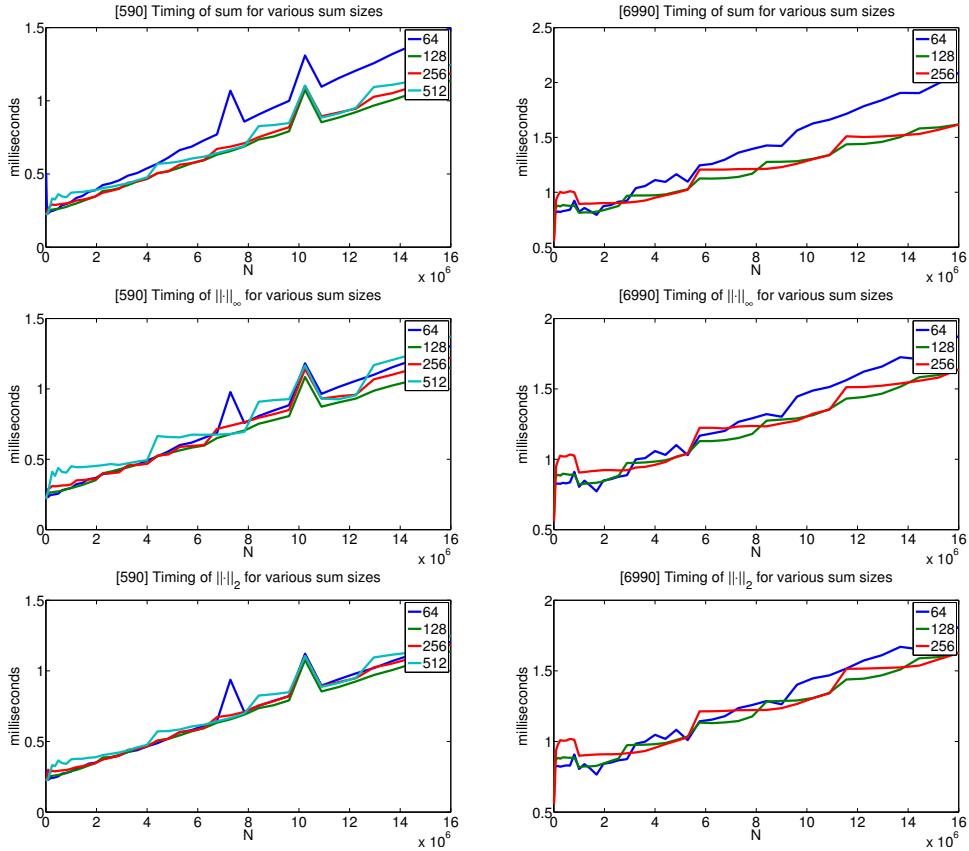


Figure 4: Norm measurements for the 590 and 6990 GPU's with N elements

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](#)

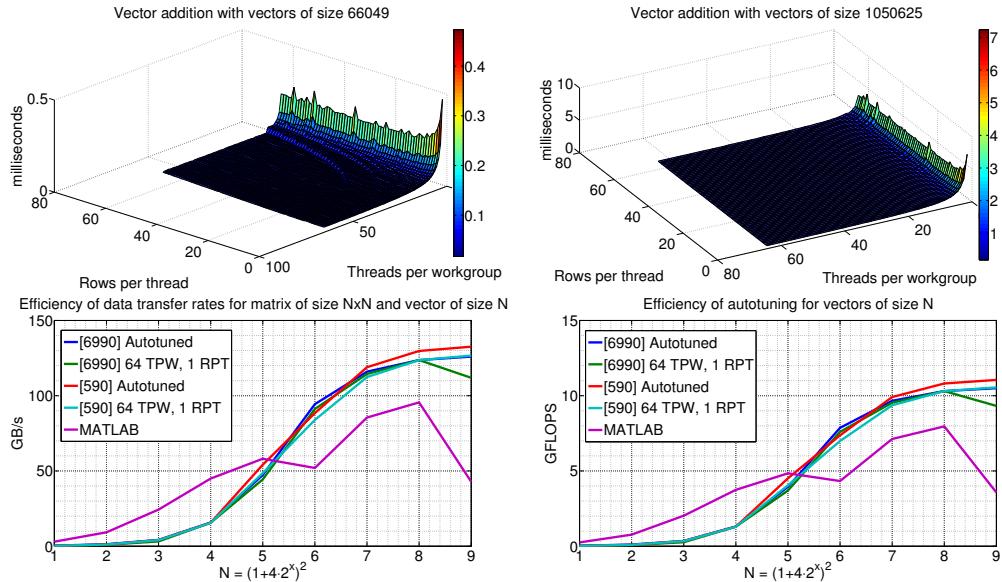


Figure 5: Demonstration of autotuning and performance calculation for a double precision band matrix with bandwidth 3.

As can be seen, the performance is not very good. This is unfortunately as expected, as we have to load two values and write one value from global memory. In contrast, we only perform one arithmetic 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 eight 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 extensions of the vector type. The kernel can be found in [10.1.4](#)

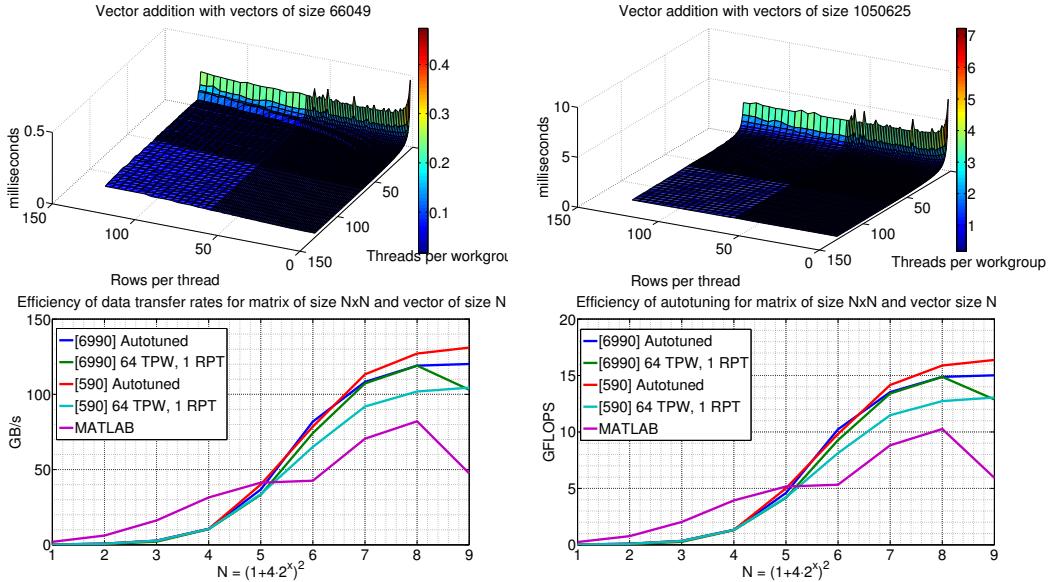


Figure 6: Demonstration of autotuning and performance calculation for a vector multiplied by a constant

Again we observe that our kernel doesn't achieve a very high performance, and again, the blame is given to the fact that it's just one arithmetic 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 extensions of the vector type. The kernel can be found in [10.1.7](#)

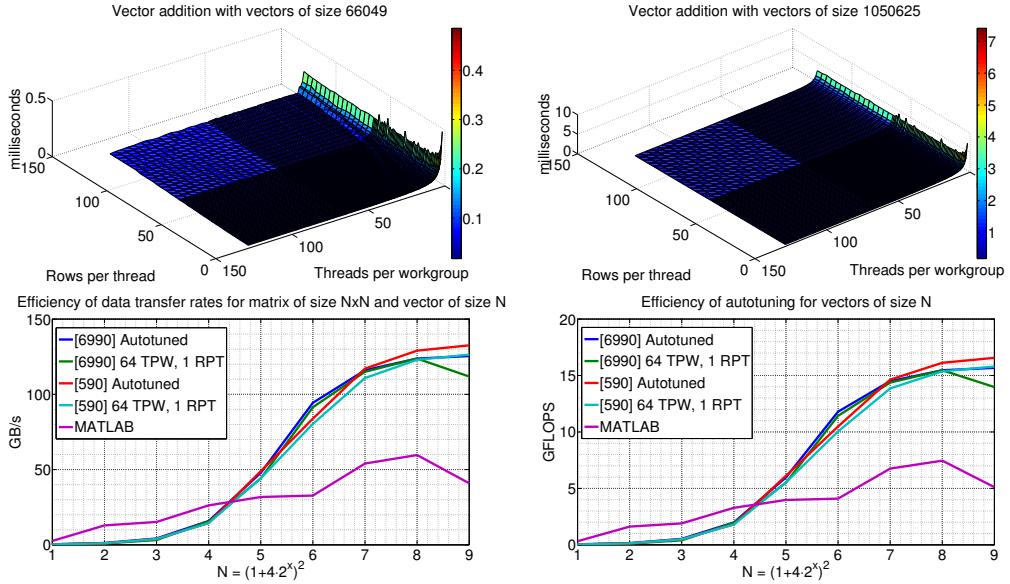


Figure 7: Demonstration of autotuning and performance calculation for a vector multiplied by a constant

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.

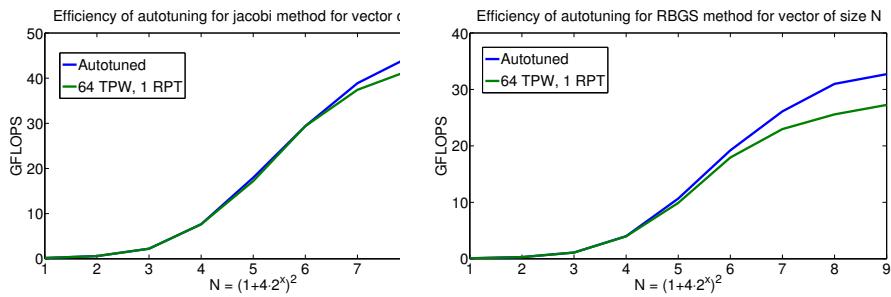


Figure 8: Autotuning results for the jacobi and RBGS method using doubles

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

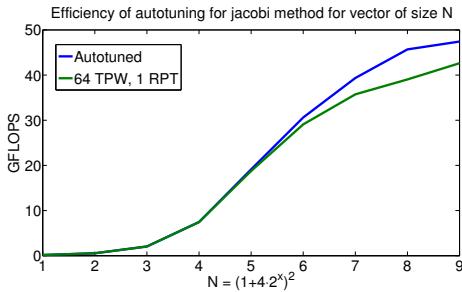


Figure 9: Autotuning of the defect calculator using doubles

Again we observe that the specialized methods perform better. Much of this extra performance can most likely be attributed to the kernels being simpler. 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 specialized 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

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

With dirichlet boundary conditions:

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

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

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

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

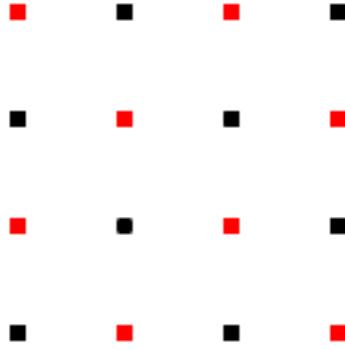


Figure 10: Red-Black ordering

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

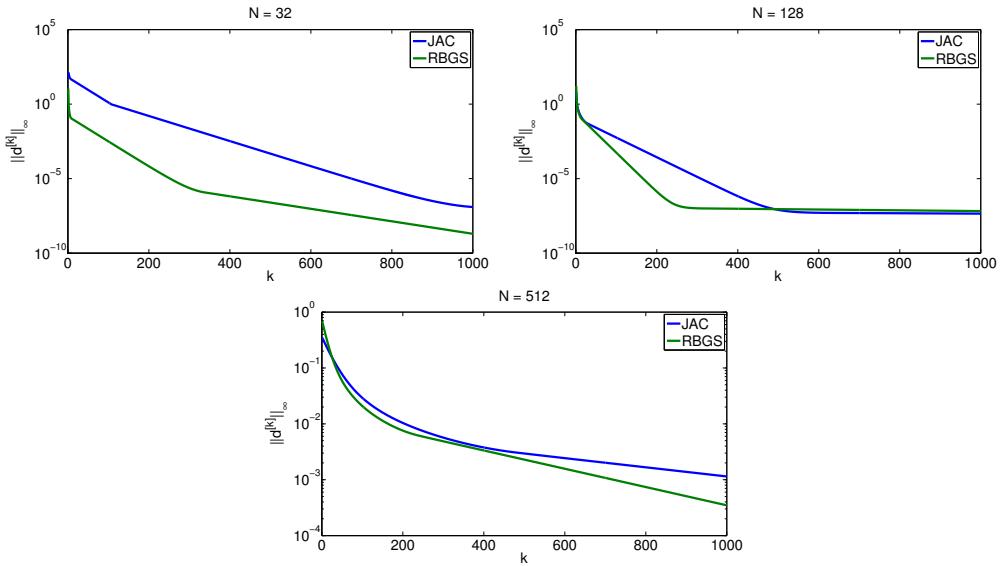


Figure 11: Jacobi and GS-RB smoothers measured against eachother.

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:

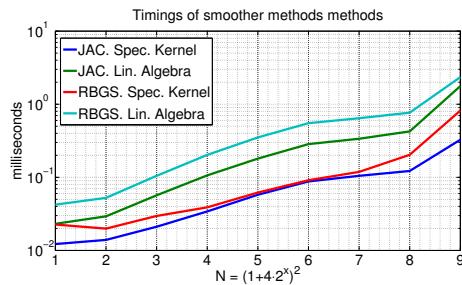


Figure 12: Execution times of the smoother methods on the AMD 6990 GPU. (double precision)

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 transferring 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

$$I_{2h}^h = \frac{1}{4} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}_{2h}^h$$

$$I_{2h}^h = \frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}_h^{2h}$$

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.

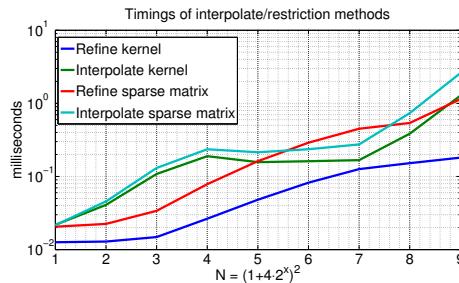


Figure 13: Execution times of the transfer methods on the AMD 6990 GPU.
(double precision)

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
$I_{2h}^{2h}(L.A.)$	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
$I_{2h}^{2h}(L.A.)$	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
$I_{2h}^{2h}(L.A.)$	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
$I_{2h}^{2h}(L.A.)$	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 recognize this form as being one we can handle with our smoothers. The methods for calculating the defect have been implemented and are presented:

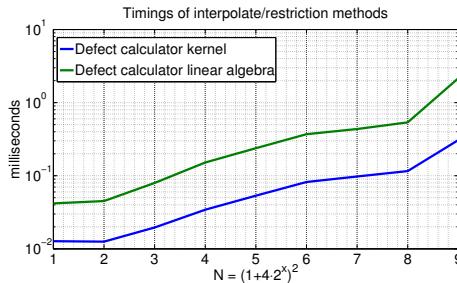


Figure 14: Execution times of defect calculation on the AMD 6990 GPU. (double precision)

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  $u^{[0]}$ .
 $u^{[0]} \leftarrow MultiGrid_V(k, u^{[0]}, b^{[0]}, v_1, v_2)$ 
for  $k = 2, 3, \dots, k_{max}$  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  $i = 1, \dots, v_{cor}$  do
         $u_h \leftarrow SMOOTH(u_h, b_h)$ 
    end for
else
    for  $i = 1, \dots, v_1$  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  $i = 1, \dots, v_2$  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 sacrificing 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.

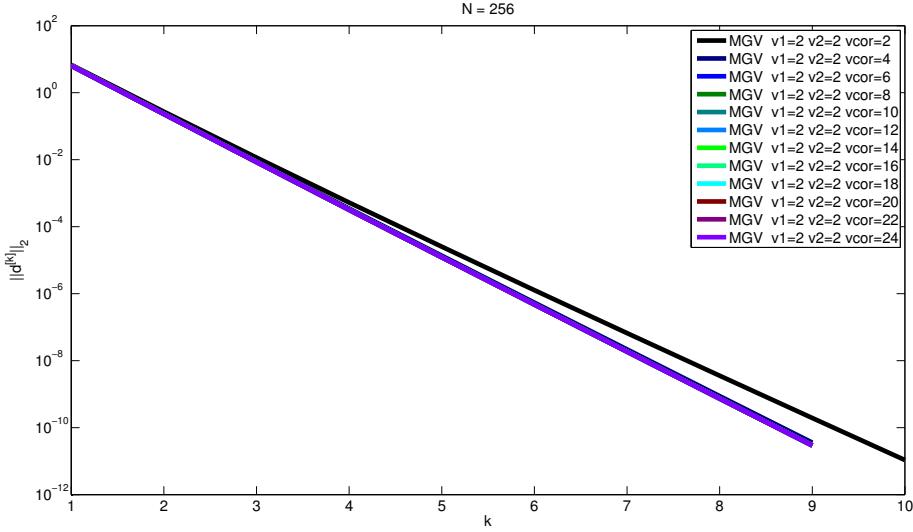


Figure 15: Testing the multigrid method with RBGS on a grid of size [257, 257] with low values of v_1, v_2 and 6 subgrids

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 .

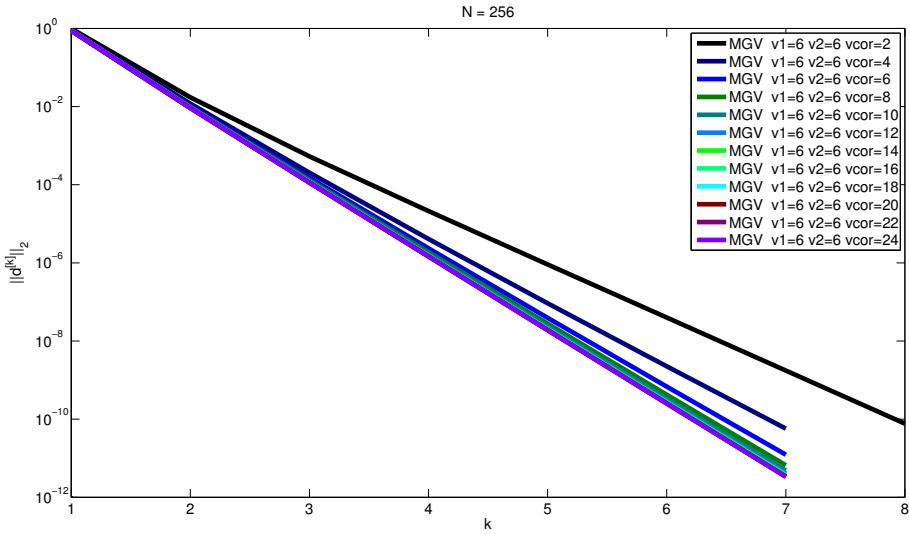


Figure 16: Testing the multigrid method with RBGS on a grid of size [257, 257] with higher values of v_1, v_2 and 6 subgrids

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:

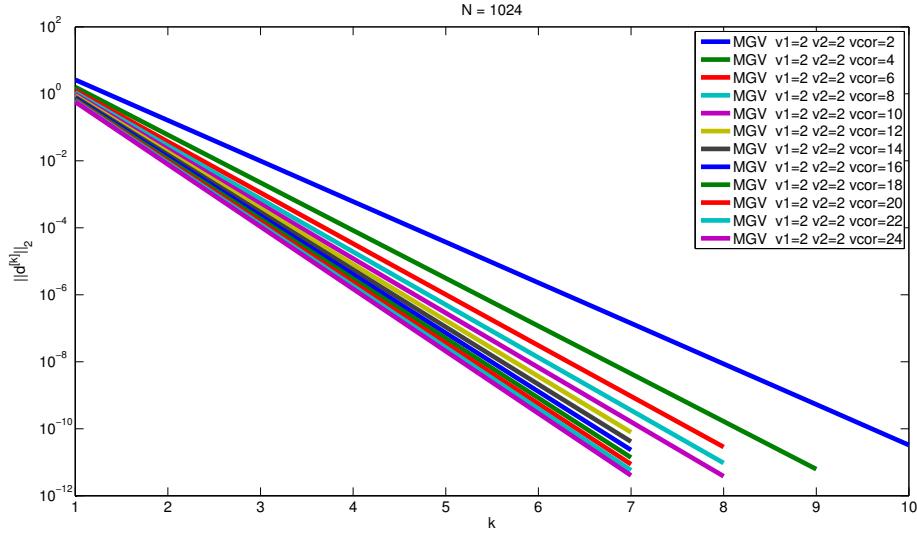


Figure 17: Testing the multigrid method with RBGS on a grid of size [1025, 1025] with low values of v_1, v_2 and 8 subgrids

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 :

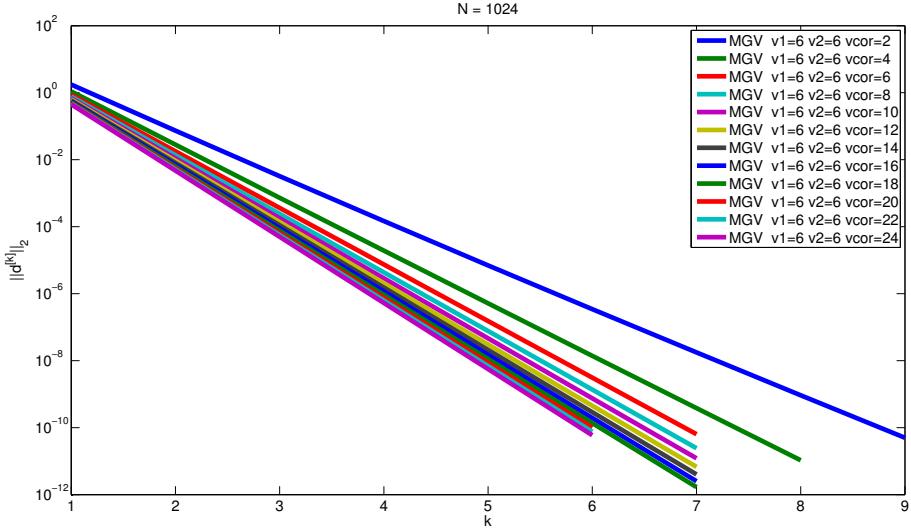


Figure 18: Testing the multigrid method with RBGS on a grid of size [1025, 1025] with higher values of v_1 , v_2 and 8 subgrids

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:

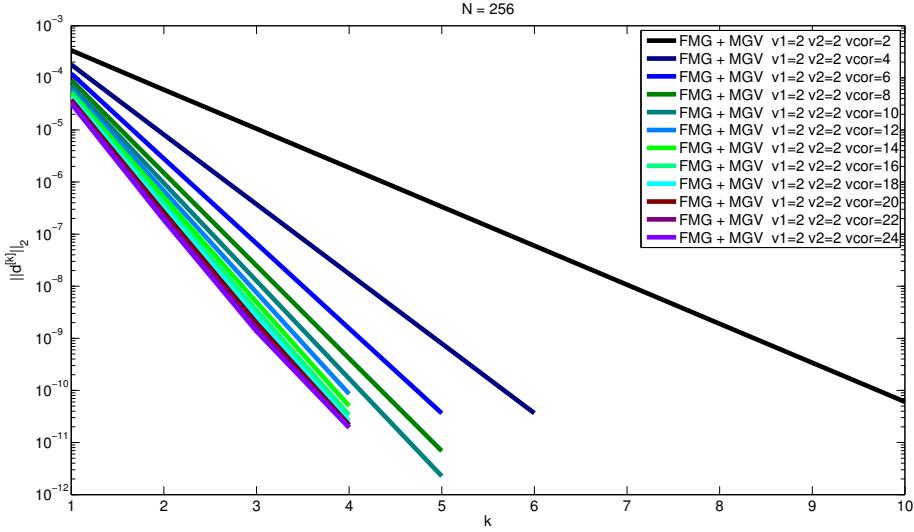


Figure 19: Testing the full multigrid method with RBGS on a grid of size [257, 257] with low values of v_1, v_2 and 6 subgrids

As we can see, based purely on the number of MGV iterations needed, we've achieved quite 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:

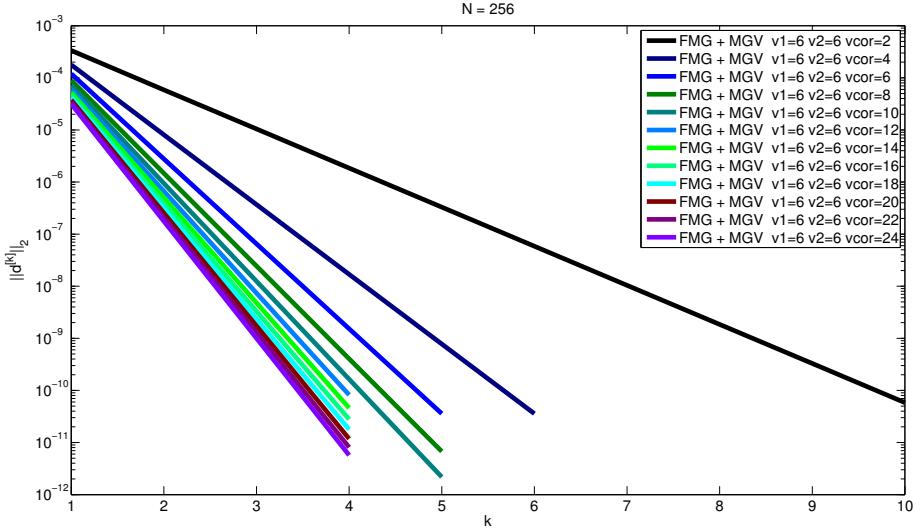


Figure 20: Testing the full multigrid method with RBGS on a grid of size [257, 257] with higher values of v_1 , v_2 and 6 subgrids

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:

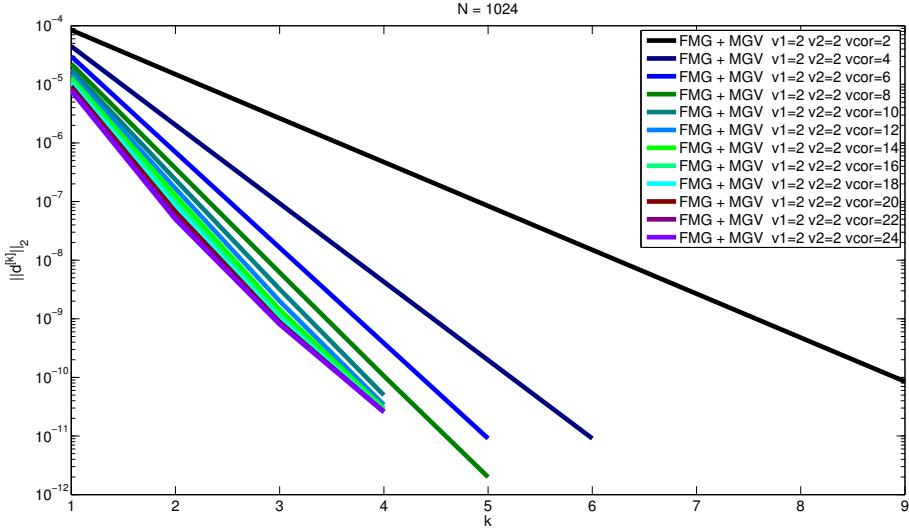


Figure 21: Testing the full multigrid method with RBGS on a grid of size [1025, 1025] with low values of v_1, v_2 and 8 subgrids

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

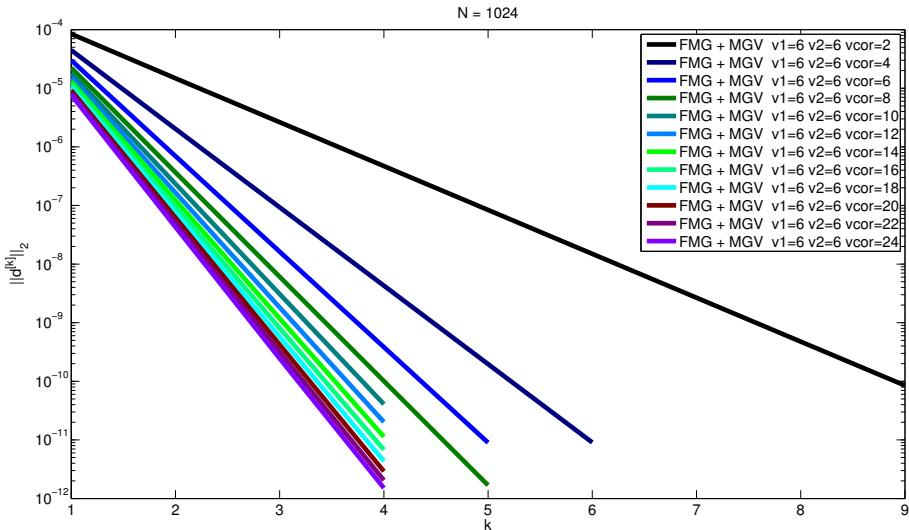


Figure 22: Testing the full multigrid method with RBGS on a grid of size [1025, 1025] with higher values of v_1, v_2 and 8 subgrids

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 reduced 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:

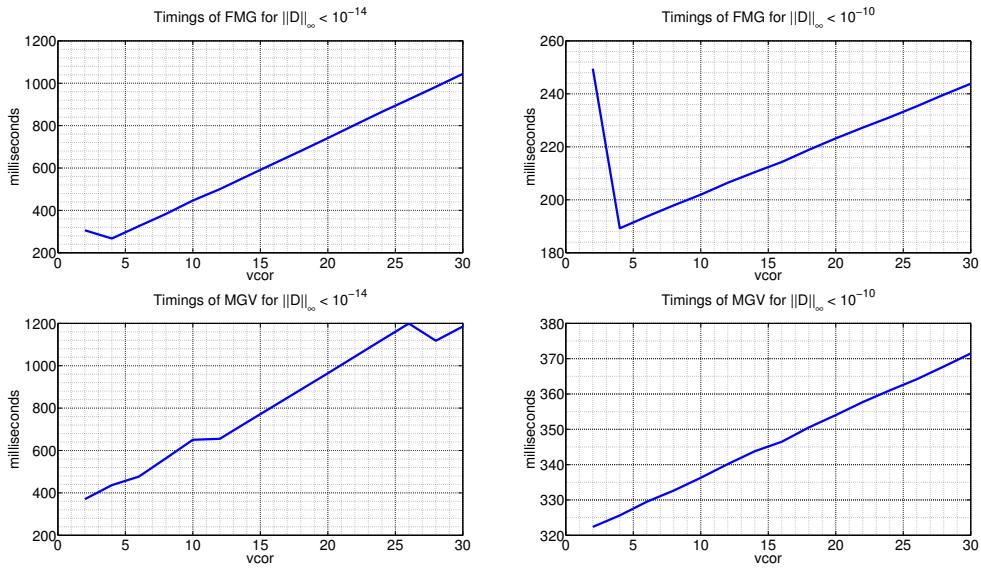


Figure 23: Solving times for FMG+MGV and MGV with $v_1 = v_2 = 2$ (mixed precision, RBGS)

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 :

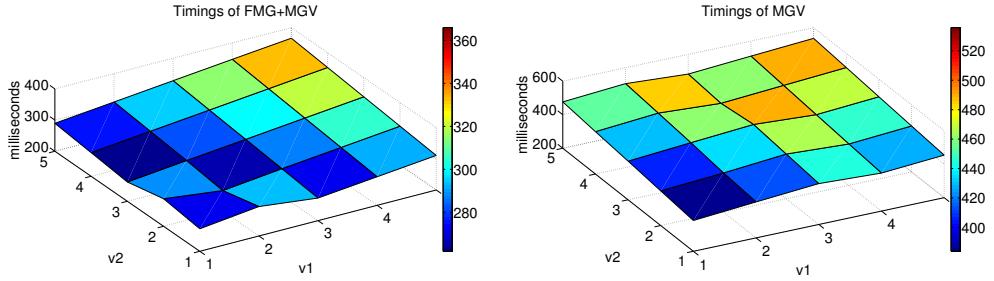


Figure 24: Solving times for FMG+MGV and MGV with $v_{cor} = 4$ (mixed precision, RBGS)

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 perform 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:

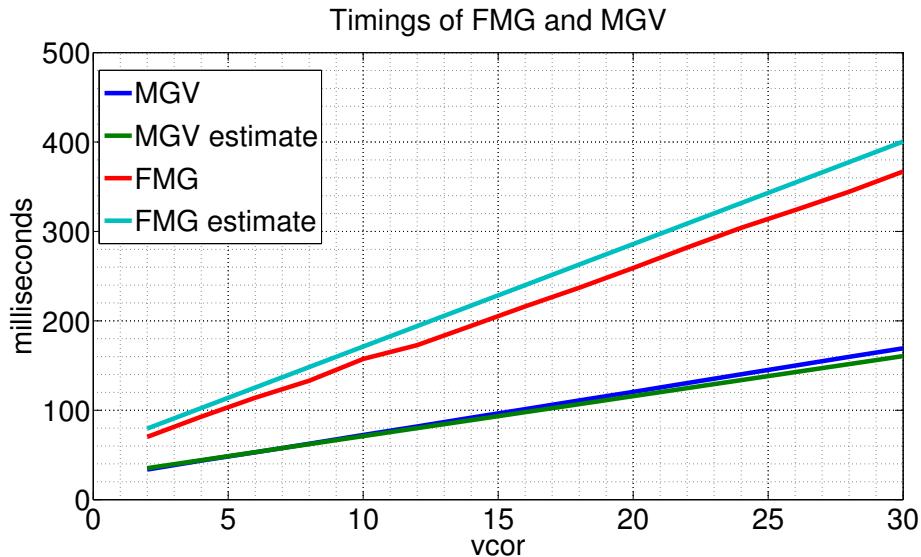


Figure 25: Comparing timings of FMG and MGV on the AMD 6990 GPU with estimates

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 a lot, 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,

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

Where

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

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

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

Which means that

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

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:

$$\begin{aligned}\theta^{(0,0)} &:= (\theta_1, \theta_2) & \theta^{(1,1)} &:= (\bar{\theta}_1, \bar{\theta}_2) \\ \theta^{(1,0)} &:= (\bar{\theta}_1, \theta_2) & \theta^{(0,1)} &:= (\theta_1, \bar{\theta}_2)\end{aligned}$$

Where

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

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 transferred 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 recognize that the point of the error correcter is to eliminate 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 eliminates 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 GPULAB⁵. 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

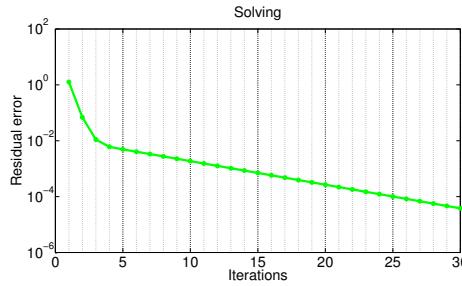


Figure 26: Solving the wave equation using RBGS with $v_1 = 2, v_2 = 2, v_{cor} = 20$

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

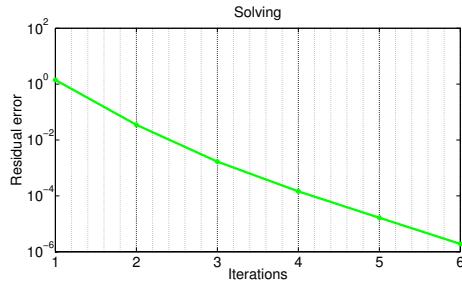


Figure 27: Solving the wave equation using RBGS with $v_1 = 2, v_2 = 2, v_{cor} = 20$ and 2 subgrids

⁵A 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:

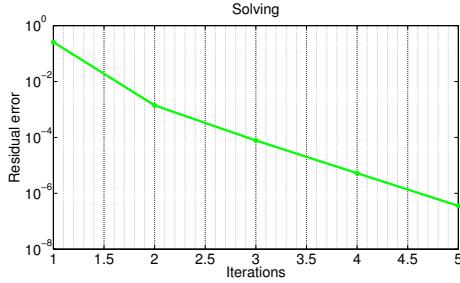


Figure 28: Solving the wave equation using RBGS with $v_1 = 6, v_2 = 6, v_{cor} = 20$ and 2 subgrids

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

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

The physical situation is a water-tank with depth h and free surface elevation ζ . The poisson RBGS solver has been implemented in GPULAB⁶. 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

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

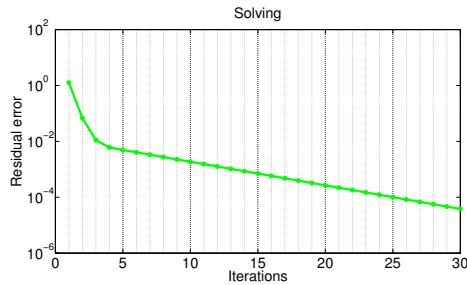


Figure 29: Solving the wave equation using RBGS with $v_1 = 2, v_2 = 2, v_{cor} = 20$

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

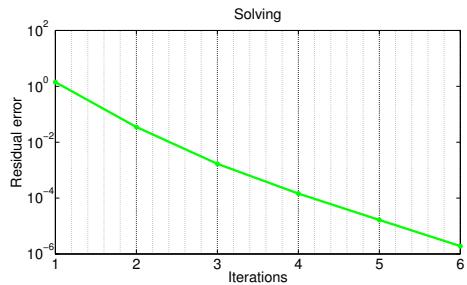


Figure 30: Solving the wave equation using RBGS with $v_1 = 2, v_2 = 2, v_{cor} = 20$ and 2 subgrids

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:

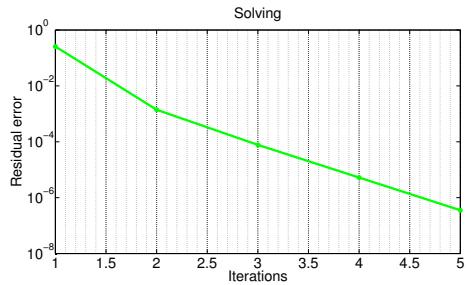


Figure 31: Solving the wave equation using RBGS with $v_1 = 6, v_2 = 6, v_{cor} = 20$ and 2 subgrids

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 neccesary 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 prototyping 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. Referring again to gustafsons law, there's a lot to gain, if we could connect the GPU's.

September 13, 2012

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 "Stanmore 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

```

1 #include <KernelHeaders.h>
2 __kernel void SparseMatrixVectorFF(__global float * matData,
3                                     __global unsigned int * matCol, __global unsigned int *
4                                     matRow, __global float * vecData, __global float *
5                                     returnData, unsigned int rowVectorLength)
6 {
7     unsigned int j;
8     unsigned int i = get_global_id(0);
9     float temp;
10    unsigned int start, end, col;
11    while (i < rowVectorLength-1)
12    {
13        start = matRow[i];
14        end = matRow[i+1];
15        temp = matData[start] * vecData[matCol[start]];
16        for (j = start+1; j < end; j += 1)
17        {
18            col = matCol[j];
19            temp += matData[j] * vecData[col];
20        }
21    }
22
23 #ifdef __DOUBLE_ALLOWED__
24
25 __kernel void SparseMatrixVectorDF(__global double * matData,
26                                     __global unsigned int * matCol, __global unsigned int *
27                                     matRow, __global float * vecData, __global double *
28                                     returnData, unsigned int rowVectorLength)
29 {
30     int a = 0;
31
32 __kernel void SparseMatrixVectorDD(__global double * matData,
33                                     __global unsigned int * matCol, __global unsigned int *
34                                     matRow, __global float * vecData, __global double *
35                                     returnData, unsigned int rowVectorLength)
36 }
```

```

matRow, __global double * vecData, __global double *
returnData, unsigned int rowVectorLength)
31 {
32     unsigned int j;
33     unsigned int i = get_global_id(0);
34     double temp;
35     unsigned int start, end, col;
36     while (i < rowVectorLength-1)
37     {
38         start = matRow[i];
39         end = matRow[i+1];
40         temp = matData[start] * vecData[matCol[start]];
41         for (j = start+1; j < end; j += 1)
42         {
43             col = matCol[j];
44             temp += matData[j] * vecData[col];
45         }
46         returnData[i] = temp;
47         i += get_global_size(0);
48     }
49 }
50
51 __kernel void SparseMatrixVectorFD(__global float * matData,
52                                     __global unsigned int * matCol, __global unsigned int *
53                                     matRow, __global double * vecData, __global float *
54                                     returnData, unsigned int rowVectorLength)
55 {
56     int a = 0;
57 }
58
59 #endif

```

10.1.2 Band matrix times vector kernel

```

1 #include <KernelHeaders.h>
2 __kernel void BandMatrixVectorFF(__global float * matData,
3                                 __global float * vecData, __global float * returnData,
4                                 unsigned int width, unsigned int bandwidth, unsigned int
5                                 length)
6 {
7     int j;
8     int i = get_global_id(0);
9     int low = i - bandwidth;
10    int high = i + bandwidth;
11    float temp = 0.0;
12    for (j = low; j <= high; j += 1)

```

```

10     {
11         if ((j >= 0) && (j < width))
12         {
13             temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
14         }
15     }
16     returnData[i] = temp;
17 }
18
19 #ifdef __DOUBLE_ALLOWED__
20
21 __kernel void BandMatrixVectorFD(__global float * matData,
22                                 __global double * vecData, __global double * returnData,
23                                 unsigned int width, unsigned int bandwidth, unsigned int
24                                 length)
25 {
26     int j;
27     int i = get_global_id(0);
28     int low = i - bandwidth;
29     int high = i + bandwidth;
30     double temp = 0.0;
31     for (j = low; j <= high; j += 1)
32     {
33         if ((j >= 0) && (j < width))
34         {
35             temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
36         }
37     }
38     returnData[i] = temp;
39 }
40
41 __kernel void BandMatrixVectorDD(__global double * matData,
42                                 __global double * vecData, __global double * returnData,
43                                 unsigned int width, unsigned int bandwidth, unsigned int
44                                 length)
45 {
46     int j;
47     int i = get_global_id(0);
48     int low = i - bandwidth;
49     int high = i + bandwidth;
50     double temp = 0.0;
51     for (j = low; j <= high; j += 1)
52     {
53         if ((j >= 0) && (j < width))
54         {
55             temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
56         }
57     }
58     returnData[i] = temp;

```

```

53 }
54
55 __kernel void BandMatrixVectorDF(__global double * matData,
56     __global float * vecData, __global float * returnData,
57     unsigned int width, unsigned int bandwidth, unsigned int
58     length)
59 {
60     int j;
61     int i = get_global_id(0);
62     int low = i - bandwidth;
63     int high = i + bandwidth;
64     double temp = 0.0;
65     for (j = low; j <= high; j += 1)
66     {
67         if ((j >= 0) && (j < width))
68         {
69             temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
70         }
71     }
72 #endif

```

10.1.3 Upper diagonal matrix times vector kernel

```

1 #include <KernelHeaders.h>
2 __kernel void BandMatrixVectorFF(__global float * matData,
3     __global float * vecData, __global float * returnData,
4     unsigned int width, unsigned int bandwidth, unsigned int
5     length)
6 {
7     int j;
8     int i = get_global_id(0);
9     int low = i - bandwidth;
10    int high = i + bandwidth;
11    float temp = 0.0;
12    for (j = low; j <= high; j += 1)
13    {
14        if ((j >= 0) && (j < width))
15        {
16            temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
17        }
18    }
19    returnData[i] = temp;
20 }

```

```

18
19 #ifdef __DOUBLE_ALLOWED__
20
21 __kernel void BandMatrixVectorFD(__global float * matData,
22                                 __global double * vecData, __global double * returnData,
23                                 unsigned int width, unsigned int bandwidth, unsigned int
24                                 length)
25 {
26     int j;
27     int i = get_global_id(0);
28     int low = i - bandwidth;
29     int high = i + bandwidth;
30     double temp = 0.0;
31     for (j = low; j <= high; j += 1)
32     {
33         if ((j >= 0) && (j < width))
34         {
35             temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
36         }
37     }
38     returnData[i] = temp;
39 }
40
41 __kernel void BandMatrixVectorDD(__global double * matData,
42                                 __global double * vecData, __global double * returnData,
43                                 unsigned int width, unsigned int bandwidth, unsigned int
44                                 length)
45 {
46     int j;
47     int i = get_global_id(0);
48     int low = i - bandwidth;
49     int high = i + bandwidth;
50     double temp = 0.0;
51     for (j = low; j <= high; j += 1)
52     {
53         if ((j >= 0) && (j < width))
54         {
55             temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
56         }
57     }
58     returnData[i] = temp;
59 }
60
61 __kernel void BandMatrixVectorDF(__global double * matData,
62                                 __global float * vecData, __global float * returnData,
63                                 unsigned int width, unsigned int bandwidth, unsigned int
64                                 length)
65 {
66     int j;

```

```

58     int i = get_global_id(0);
59     int low = i - bandwidth;
60     int high = i + bandwidth;
61     double temp = 0.0;
62     for (j = low; j <= high; j += 1)
63     {
64         if ((j >= 0) && (j < width))
65         {
66             temp += vecData[j] * matData[i*(1+2*bandwidth)-low+j];
67         }
68     }
69     returnData[i] = temp;
70 }
71
72 #endif

```

10.1.4 Vector times constant kernel

```

1 #include <KernelHeaders.h>
2 __kernel void VectorTimesConstantFF(__global float* vector,
3                                     __global float* output, float element)
4 {
5     int i = get_global_id(0);
6     output[i] = vector[i] * element;
7 }
8
9 #ifdef __DOUBLE_ALLOWED__
10 __kernel void VectorTimesConstantFD(__global float* vector,
11                                     __global float* output, double element)
12 {
13     int i = get_global_id(0);
14     output[i] = vector[i] * element;
15 }
16 __kernel void VectorTimesConstantDD(__global double* vector,
17                                     __global double* output, double element)
18 {
19     int i = get_global_id(0);
20     output[i] = vector[i] * element;
21 }
21 #endif

```

10.1.5 Vector plus vector kernel

```

1 #include <KernelHeaders.h>
2 __kernel void VectorPlusVectorFF(__global float* vector1,
3                                 __global float* vector2, __global float* output)
4 {
5     int i = get_global_id(0);
6     output[i] = vector1[i] + vector2[i];
7 }
8
9 #ifdef __DOUBLE_ALLOWED__
10 __kernel void VectorPlusVectorFD(__global float* vector1,
11                                 __global double* vector2, __global float* output)
12 {
13     int i = get_global_id(0);
14     output[i] = vector1[i] + vector2[i];
15 }
16 __kernel void VectorPlusVectorDD(__global double* vector1,
17                                 __global double* vector2, __global double* output)
18 {
19     int i = get_global_id(0);
20     output[i] = vector1[i] + vector2[i];
21 }
22 __kernel void VectorPlusVectorDF(__global double* vector1,
23                                 __global float* vector2, __global double* output)
24 {
25     int i = get_global_id(0);
26     output[i] = vector1[i] + vector2[i];
27 #endif

```

10.1.6 Vector minus vector kernel

```

1 #include <KernelHeaders.h>
2 __kernel void VectorMinusVectorFF(__global float* vector1,
3                                 __global float* vector2, __global float* output)
4 {
5     int i = get_global_id(0);
6     output[i] = vector1[i] - vector2[i];
7 }
8
9 #ifdef __DOUBLE_ALLOWED__

```

```

10 __kernel void VectorMinusVectorFD(__global float* vector1,
11                                     __global double* vector2, __global float* output)
12 {
13     int i = get_global_id(0);
14     output[i] = vector1[i] - vector2[i];
15 }
16 __kernel void VectorMinusVectorDD(__global double* vector1,
17                                     __global double* vector2, __global double* output)
18 {
19     int i = get_global_id(0);
20     output[i] = vector1[i] - vector2[i];
21 }
22 __kernel void VectorMinusVectorDF(__global double* vector1,
23                                     __global float* vector2, __global double* output)
24 {
25     int i = get_global_id(0);
26     output[i] = vector1[i] - vector2[i];
27 #endif

```

10.1.7 Vector minus vector times constant kernel

```

1 #include <KernelHeaders.h>
2 __kernel void VectorMinusVectorConstantFF(__global float*
3                                         vector1, __global float* vector2, __global float* output,
4                                         float con, unsigned int length)
5 {
6     int i = get_global_id(0);
7     while (i < length)
8     {
9         output[i] = vector1[i] - con*vector2[i];
10    i += get_global_size(0);
11 }
12
13 #ifdef __DOUBLE_ALLOWED__
14 __kernel void VectorMinusVectorConstantFD(__global float*
15                                         vector1, __global double* vector2, __global float* output,
16                                         double con, unsigned int length)
17 {
18     int i = get_global_id(0);
19     while (i < length)
20     {

```

```

19     output[ i ] = vector1[ i ] - con*vector2[ i ];
20     i += get_global_size(0);
21 }
22 }
23
24 __kernel void VectorMinusVectorConstantDD( __global double*
25     vector1, __global double* vector2, __global double* output
26     , double con, unsigned int length)
27 {
28     int i = get_global_id(0);
29     while ( i < length )
30     {
31         output[ i ] = vector1[ i ] - con*vector2[ i ];
32         i += get_global_size(0);
33     }
34 }
35
36 __kernel void VectorMinusVectorConstantDF( __global double*
37     vector1, __global float* vector2, __global double* output,
38     double con, unsigned int length)
39 {
40     int i = get_global_id(0);
41     while ( i < length )
42     {
43         output[ i ] = vector1[ i ] - con*vector2[ i ];
44         i += get_global_size(0);
45     }
46 }
47 #endif

```

10.1.8 Vector sum kernel

```

1 #include <KernelHeaders.h>
2 __kernel void VectorReductionF( __global float* input, __global
3     float* output, unsigned int threadSize, unsigned int
4     problemSize)
5 {
6     //Declare shared memory space:
7     __local float sdata[512];
8     unsigned int tid = get_local_id(0);
9     unsigned int i = get_group_id(0)*(threadSize*2) + tid ;
10    unsigned int gridSize = threadSize*2*get_num_groups(0);
11    float tempCounter = 0.0;
12    while ( i < problemSize)
13    {
14        tempCounter += input[ i ];
15    }
16    if (tid == 0)
17    {
18        output[ i ] = tempCounter;
19    }
20 }

```

```

13     if ( i + threadSize < problemSize )
14     {
15         tempCounter += input[ i + threadSize ];
16     }
17     i += gridSize;
18 }
19 sdata[ tid ] = tempCounter;
20 barrier(CLK_LOCAL_MEM_FENCE);
21 if ( threadSize ≥ 512 )
22 {
23     if ( tid < 256 )
24     {
25         sdata[ tid ] += sdata[ tid + 256 ];
26     }
27     barrier(CLK_LOCAL_MEM_FENCE);
28 }
29 if ( threadSize ≥ 256 )
30 {
31     if ( tid < 128 )
32     {
33         sdata[ tid ] += sdata[ tid + 128 ];
34     }
35     barrier(CLK_LOCAL_MEM_FENCE);
36 }
37 if ( threadSize ≥ 128 )
38 {
39     if ( tid < 64 )
40     {
41         sdata[ tid ] += sdata[ tid + 64 ];
42     }
43     barrier(CLK_LOCAL_MEM_FENCE);
44 }
45 if ( tid < 32 )
46 {
47     if ( threadSize ≥ 64 )
48     {
49         sdata[ tid ] += sdata[ tid + 32 ];
50     }
51     if ( threadSize ≥ 32 )
52     {
53         sdata[ tid ] += sdata[ tid + 16 ];
54     }
55     if ( threadSize ≥ 16 )
56     {
57         sdata[ tid ] += sdata[ tid + 8 ];
58     }
59     if ( threadSize ≥ 8 )
60     {
61         sdata[ tid ] += sdata[ tid + 4 ];

```

```

62      }
63      if (threadSize >= 4)
64      {
65          sdata[tid] += sdata[tid + 2];
66      }
67      if (threadSize >= 2)
68      {
69          sdata[tid] += sdata[tid + 1];
70      }
71  }
72  if (tid == 0)
73  {
74      output[get_group_id(0)] = sdata[0];
75  }
76 }
77
78 #ifdef __DOUBLE_ALLOWED__
79 __kernel void VectorReductionD(__global double* input, __global
80                                double* output, unsigned int threadSize, unsigned int
81                                problemSize)
82 {
83     //Declare shared memory space:
84     __local double sdata[512];
85     unsigned int tid = get_local_id(0);
86     unsigned int i = get_group_id(0)*(threadSize*2) + tid;
87     unsigned int gridSize = threadSize*2*get_num_groups(0);
88     double tempCounter = 0.0;
89     while (i < problemSize)
90     {
91         tempCounter += input[i];
92         if (i + threadSize < problemSize)
93         {
94             tempCounter += input[i + threadSize];
95         }
96         i += gridSize;
97     }
98     sdata[tid] = tempCounter;
99     barrier(CLK_LOCAL_MEM_FENCE);
100    if (threadSize >= 512)
101    {
102        if (tid < 256)
103        {
104            sdata[tid] += sdata[tid + 256];
105        }
106        barrier(CLK_LOCAL_MEM_FENCE);
107    }
108    if (threadSize >= 256)
109    {
110        if (tid < 128)
111        {
112            sdata[tid] += sdata[tid + 128];
113        }
114        barrier(CLK_LOCAL_MEM_FENCE);
115    }
116    if (threadSize >= 128)
117    {
118        if (tid < 64)
119        {
120            sdata[tid] += sdata[tid + 64];
121        }
122        barrier(CLK_LOCAL_MEM_FENCE);
123    }
124    if (threadSize >= 64)
125    {
126        if (tid < 32)
127        {
128            sdata[tid] += sdata[tid + 32];
129        }
130        barrier(CLK_LOCAL_MEM_FENCE);
131    }
132    if (threadSize >= 32)
133    {
134        if (tid < 16)
135        {
136            sdata[tid] += sdata[tid + 16];
137        }
138        barrier(CLK_LOCAL_MEM_FENCE);
139    }
140    if (threadSize >= 16)
141    {
142        if (tid < 8)
143        {
144            sdata[tid] += sdata[tid + 8];
145        }
146        barrier(CLK_LOCAL_MEM_FENCE);
147    }
148    if (threadSize >= 8)
149    {
150        if (tid < 4)
151        {
152            sdata[tid] += sdata[tid + 4];
153        }
154        barrier(CLK_LOCAL_MEM_FENCE);
155    }
156    if (threadSize >= 4)
157    {
158        if (tid < 2)
159        {
160            sdata[tid] += sdata[tid + 2];
161        }
162        barrier(CLK_LOCAL_MEM_FENCE);
163    }
164    if (threadSize >= 2)
165    {
166        if (tid == 0)
167        {
168            output[get_group_id(0)] = sdata[0];
169        }
170    }
171 }
172
173 #endif

```

```

109      {
110          sdata[tid] += sdata[tid + 128];
111      }
112      barrier(CLK_LOCAL_MEM_FENCE);
113  }
114  if (threadSize >= 128)
115  {
116      if (tid < 64)
117      {
118          sdata[tid] += sdata[tid + 64];
119      }
120      barrier(CLK_LOCAL_MEM_FENCE);
121  }
122  if (tid < 32)
123  {
124      if (threadSize >= 64)
125      {
126          sdata[tid] += sdata[tid + 32];
127      }
128      if (threadSize >= 32)
129      {
130          sdata[tid] += sdata[tid + 16];
131      }
132      if (threadSize >= 16)
133      {
134          sdata[tid] += sdata[tid + 8];
135      }
136      if (threadSize >= 8)
137      {
138          sdata[tid] += sdata[tid + 4];
139      }
140      if (threadSize >= 4)
141      {
142          sdata[tid] += sdata[tid + 2];
143      }
144      if (threadSize >= 2)
145      {
146          sdata[tid] += sdata[tid + 1];
147      }
148  }
149  if (tid == 0)
150  {
151      output[get_group_id(0)] = sdata[0];
152  }
153 }
154 #endif

```

10.1.9 Vector 2-norm kernel

```

1 #include <KernelHeaders.h>
2 __kernel void Norm2F(__global float* input, __global float*
3                      output, unsigned int threadSize, unsigned int problemSize)
4 {
5     //Declare shared memory space:
6     __local float sdata[512];
7     unsigned int tid = get_local_id(0);
8     unsigned int i = get_group_id(0)*(threadSize*2) + tid;
9     unsigned int gridSize = threadSize*2*get_num_groups(0);
10    float tempCounter = 0.0;
11    while (i < problemSize)
12    {
13        tempCounter += input[i]*input[i];
14        if (i + threadSize < problemSize)
15        {
16            tempCounter += input[i + threadSize]*input[i +
17                           threadSize];
18        }
19        i += gridSize;
20    }
21    sdata[tid] = tempCounter;
22    barrier(CLK_LOCAL_MEM_FENCE);
23    if (threadSize ≥ 512)
24    {
25        if (tid < 256)
26        {
27            sdata[tid] += sdata[tid + 256];
28        }
29        barrier(CLK_LOCAL_MEM_FENCE);
30    }
31    if (threadSize ≥ 256)
32    {
33        if (tid < 128)
34        {
35            sdata[tid] += sdata[tid + 128];
36        }
37        barrier(CLK_LOCAL_MEM_FENCE);
38    }
39    if (threadSize ≥ 128)
40    {
41        if (tid < 64)
42        {
43            sdata[tid] += sdata[tid + 64];
44        }
45    }
46    barrier(CLK_LOCAL_MEM_FENCE);

```

```

44    }
45    if (tid < 32)
46    {
47        if (threadSize ≥ 64)
48        {
49            sdata[tid] += sdata[tid + 32];
50        }
51        if (threadSize ≥ 32)
52        {
53            sdata[tid] += sdata[tid + 16];
54        }
55        if (threadSize ≥ 16)
56        {
57            sdata[tid] += sdata[tid + 8];
58        }
59        if (threadSize ≥ 8)
60        {
61            sdata[tid] += sdata[tid + 4];
62        }
63        if (threadSize ≥ 4)
64        {
65            sdata[tid] += sdata[tid + 2];
66        }
67        if (threadSize ≥ 2)
68        {
69            sdata[tid] += sdata[tid + 1];
70        }
71    }
72    if (tid == 0)
73    {
74        output[get_group_id(0)] = sdata[0];
75    }
76 }
77
78
79 #ifdef __DOUBLE_ALLOWED__
80 __kernel void Norm2D(__global double* input, __global double*
81                      output, unsigned int threadSize, unsigned int problemSize)
82 {
83     //Declare shared memory space:
84     __local double sdata[512];
85     unsigned int tid = get_local_id(0);
86     unsigned int i = get_group_id(0)*(threadSize*2) + tid;
87     unsigned int gridSize = threadSize*2*get_num_groups(0);
88     double tempCounter = 0.0;
89     while (i < problemSize)
90     {
91         tempCounter += input[i]*input[i];
92         if (i + threadSize < problemSize)

```

```

92      {
93          tempCounter += input[ i + threadSize]*input[ i +
94              threadSize];
95          i += gridSize;
96      }
97      sdata[tid] = tempCounter;
98      barrier(CLK_LOCAL_MEM_FENCE);
99      if (threadSize ≥ 512)
100     {
101         if (tid < 256)
102         {
103             sdata[tid] += sdata[tid + 256];
104         }
105         barrier(CLK_LOCAL_MEM_FENCE);
106     }
107     if (threadSize ≥ 256)
108     {
109         if (tid < 128)
110         {
111             sdata[tid] += sdata[tid + 128];
112         }
113         barrier(CLK_LOCAL_MEM_FENCE);
114     }
115     if (threadSize ≥ 128)
116     {
117         if (tid < 64)
118         {
119             sdata[tid] += sdata[tid + 64];
120         }
121         barrier(CLK_LOCAL_MEM_FENCE);
122     }
123     if (tid < 32)
124     {
125         if (threadSize ≥ 64)
126         {
127             sdata[tid] += sdata[tid + 32];
128         }
129         if (threadSize ≥ 32)
130         {
131             sdata[tid] += sdata[tid + 16];
132         }
133         if (threadSize ≥ 16)
134         {
135             sdata[tid] += sdata[tid + 8];
136         }
137         if (threadSize ≥ 8)
138         {
139             sdata[tid] += sdata[tid + 4];

```

```

140      }
141      if (threadSize >= 4)
142      {
143          sdata[tid] += sdata[tid + 2];
144      }
145      if (threadSize >= 2)
146      {
147          sdata[tid] += sdata[tid + 1];
148      }
149  }
150  if (tid == 0)
151  {
152      output[get_group_id(0)] = sdata[0];
153  }
154 }
155 #endif

```

10.1.10 Vector ∞ -norm kernel

```

1 #include <KernelHeaders.h>
2 __kernel void NormInff(__global float* input, __global float*
3                         output, unsigned int threadSize, unsigned int problemSize)
4 {
5     //Declare shared memory space:
6     __local float sdata[512];
7     unsigned int tid = get_local_id(0);
8     unsigned int i = get_group_id(0)*(threadSize*2) + tid;
9     unsigned int gridSize = threadSize*2*get_num_groups(0);
10    float tempCounter = 0.0;
11    float tempNew;
12    while (i < problemSize)
13    {
14        tempNew = input[i]; if (tempNew < 0) tempNew = -tempNew;
15        if (tempNew > tempCounter) tempCounter = tempNew;
16        if (i + threadSize < problemSize)
17        {
18            tempNew = input[i + threadSize]*input[i + threadSize];
19            if (tempNew < 0) tempNew = -tempNew;
20            if (tempNew > tempCounter) tempCounter = tempNew;
21        }
22        i += gridSize;
23    }
24    sdata[tid] = tempCounter;
25    barrier(CLK_LOCAL_MEM_FENCE);
26    if (threadSize >= 512)
27    {

```

```
26     if ( tid < 256 && sdata[ tid + 256 ] > sdata[ tid ] )
27     {
28         sdata[ tid ] = sdata[ tid + 256 ];
29     }
30     barrier(CLK_LOCAL_MEM_FENCE);
31 }
32 if ( threadSize ≥ 256 )
33 {
34     if ( tid < 128 && sdata[ tid + 128 ] > sdata[ tid ] )
35     {
36         sdata[ tid ] = sdata[ tid + 128 ];
37     }
38     barrier(CLK_LOCAL_MEM_FENCE);
39 }
40 if ( threadSize ≥ 128 )
41 {
42     if ( tid < 64 && sdata[ tid + 64 ] > sdata[ tid ] )
43     {
44         sdata[ tid ] = sdata[ tid + 64 ];
45     }
46     barrier(CLK_LOCAL_MEM_FENCE);
47 }
48 if ( tid < 32 )
49 {
50     if ( threadSize ≥ 64 && sdata[ tid + 32 ] > sdata[ tid ] )
51     {
52         sdata[ tid ] = sdata[ tid + 32 ];
53     }
54     if ( threadSize ≥ 32 && sdata[ tid + 16 ] > sdata[ tid ] )
55     {
56         sdata[ tid ] = sdata[ tid + 16 ];
57     }
58     if ( threadSize ≥ 16 && sdata[ tid + 8 ] > sdata[ tid ] )
59     {
60         sdata[ tid ] = sdata[ tid + 8 ];
61     }
62     if ( threadSize ≥ 8 && sdata[ tid + 4 ] > sdata[ tid ] )
63     {
64         sdata[ tid ] = sdata[ tid + 4 ];
65     }
66     if ( threadSize ≥ 4 && sdata[ tid + 2 ] > sdata[ tid ] )
67     {
68         sdata[ tid ] = sdata[ tid + 2 ];
69     }
70     if ( threadSize ≥ 2 && sdata[ tid + 1 ] > sdata[ tid ] )
71     {
72         sdata[ tid ] = sdata[ tid + 1 ];
73     }
74 }
```

```

75     if (tid == 0)
76     {
77         output[get_group_id(0)] = sdata[0];
78     }
79 }
80
81 #ifdef __DOUBLE_ALLOWED__
82 __kernel void NormInfD(__global double* input, __global double*
83                         output, unsigned int threadSize, unsigned int problemSize
84                         )
83 {
84     //Declare shared memory space:
85     __local double sdata[512];
86     unsigned int tid = get_local_id(0);
87     unsigned int i = get_group_id(0)*(threadSize*2) + tid;
88     unsigned int gridSize = threadSize*2*get_num_groups(0);
89     double tempCounter = 0.0;
90     double tempNew;
91     while (i < problemSize)
92     {
93         tempNew = input[i]; if (tempNew < 0) tempNew = -tempNew;
94         if (tempNew > tempCounter) tempCounter = tempNew;
95         if (i + threadSize < problemSize)
96         {
97             tempNew = input[i + threadSize]*input[i + threadSize];
98             if (tempNew < 0) tempNew = -tempNew;
99             if (tempNew > tempCounter) tempCounter = tempNew;
100         }
101         i += gridSize;
102     }
103     sdata[tid] = tempCounter;
104     barrier(CLK_LOCAL_MEM_FENCE);
105     if (threadSize ≥ 512)
105     {
106         if (tid < 256 && sdata[tid + 256] > sdata[tid])
107         {
108             sdata[tid] = sdata[tid + 256];
109         }
110         barrier(CLK_LOCAL_MEM_FENCE);
111     }
112     if (threadSize ≥ 256)
113     {
114         if (tid < 128 && sdata[tid + 128] > sdata[tid])
115         {
116             sdata[tid] = sdata[tid + 128];
117         }
118         barrier(CLK_LOCAL_MEM_FENCE);
119     }
120     if (threadSize ≥ 128)

```

```

121  {
122      if (tid < 64 && sdata[tid + 64] > sdata[tid])
123      {
124          sdata[tid] = sdata[tid + 64];
125      }
126      barrier(CLK_LOCAL_MEM_FENCE);
127  }
128  if (tid < 32)
129  {
130      if (threadSize ≥ 64 && sdata[tid + 32] > sdata[tid])
131      {
132          sdata[tid] = sdata[tid + 32];
133      }
134      if (threadSize ≥ 32 && sdata[tid + 16] > sdata[tid])
135      {
136          sdata[tid] = sdata[tid + 16];
137      }
138      if (threadSize ≥ 16 && sdata[tid + 8] > sdata[tid])
139      {
140          sdata[tid] = sdata[tid + 8];
141      }
142      if (threadSize ≥ 8 && sdata[tid + 4] > sdata[tid])
143      {
144          sdata[tid] = sdata[tid + 4];
145      }
146      if (threadSize ≥ 4 && sdata[tid + 2] > sdata[tid])
147      {
148          sdata[tid] = sdata[tid + 2];
149      }
150      if (threadSize ≥ 2 && sdata[tid + 1] > sdata[tid])
151      {
152          sdata[tid] = sdata[tid + 1];
153      }
154  }
155  if (tid == 0)
156  {
157      output[get_group_id(0)] = sdata[0];
158  }
159 }
160
161 #endif

```

10.1.11 Jacobi method kernel

```

1 #include <KernelHeaders.h>

```

```

2 __kernel void JacobiMethodF(__global float * output, __global
    float * matData, __global float * rhsData, unsigned int
    width, unsigned int height, float h)
3 {
4     int j = get_global_id(1)+1;
5     int i = get_global_id(0)+1;
6     if (i < width-1)
7     {
8         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;
9         output[i + j*width] = temp;
10    }
11 }
12
13 #ifdef __DOUBLE_ALLOWED__
14
15 __kernel void JacobiMethodD(__global double * output, __global
    double * matData, __global double * rhsData, unsigned int
    width, unsigned int height, double h)
16 {
17     int j = get_global_id(1)+1;
18     int i = get_global_id(0)+1;
19     if (i < width-1)
20     {
21         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;
22         output[i + j*width] = temp;
23     }
24 }
25
26#endif

```

10.1.12 RBGS - Red kernel

```

1 #include <KernelHeaders.h>
2 __kernel void JacobiMethodEvenF(__global float * matData,
    __global float * rhsData, unsigned int width, unsigned int
    height, float h)
3 {
4     int j = get_global_id(1)+1;
5     int i = get_global_id(0)*2+1+j%2;
6     if (i < width-1)
7     {

```

```

8     float temp = (matData[ i + (j-1)*width ] + matData[ i + (j+1)
9         *width ] + matData[ i-1 + j*width ] + matData[ i+1 + j*
10        width ] - rhsData[ i + j*width]*h*h) / 4.0;
11    matData[ i + j*width ] = temp;
12 }
13 #ifdef __DOUBLE_ALLOWED__
14
15 __kernel void JacobiMethodEvenD( __global double * matData,
16                                 __global double * rhsData, unsigned int width, unsigned
17                                 int height, double h)
18 {
19     int j = get_global_id(1)+1;
20     int i = get_global_id(0)*2+1+j%2;
21     if ( i < width-1)
22     {
23         double temp = (matData[ i + (j-1)*width ] + matData[ i + (j
24             +1)*width ] + matData[ i-1 + j*width ] + matData[ i+1 + j*
25             width ] - rhsData[ i + j*width]*h*h) / 4.0;
26         matData[ i + j*width ] = temp;
27     }
28 }
29
30 #endif

```

10.1.13 RBGS - Black kernel

```

1 #include <KernelHeaders.h>
2 __kernel void JacobiMethodOddF( __global float * matData,
3                                 __global float * rhsData, unsigned int width, unsigned int
4                                 height, float h)
5 {
6     int j = get_global_id(1)+1;
7     int i = get_global_id(0)*2+1+(j+1)%2;
8     if ( i < width-1)
9     {
10         float temp = (matData[ i + width*(j-1) ] + matData[ i + width
11             *(j+1) ] + matData[ i-1 + j*width ] + matData[ i+1 + width
12             *j ] - rhsData[ i + width*j]*h*h) / 4.0;
13         matData[ i + j*width ] = temp;
14     }
15 }
16
17 #ifdef __DOUBLE_ALLOWED__

```

```

15
16 __kernel void JacobiMethodOddD(__global double * matData,
17     __global double * rhsData, unsigned int width, unsigned
18     int height, double h)
19 {
20     int j = get_global_id(1)+1;
21     int i = get_global_id(0)*2+1+(j+1)%2;
22     if (i < width-1)
23     {
24         double temp = (matData[i + width*(j-1)] + matData[i +
25             width*(j+1)] + matData[i-1 + j*width] + matData[i+1 +
26             width*j] - rhsData[i + width*j]*h*h)/4.0;
27         matData[i + j*width] = temp;
28     }
29 }
30
31 #endif

```

10.1.14 Poisson defect kernel

```

1 #include <KernelHeaders.h>
2 __kernel void JacobiCalcDefectF(__global float * matData,
3     __global float * rhsData, __global float * defectData,
4     unsigned int width, unsigned int height, float h)
5 {
6     int i = get_global_id(0);
7     int j = get_global_id(1);
8     float temp = 0.0;
9     if ((i == 0) || (j == 0) || (i == get_global_size(0)-1) ||
10        (j == get_global_size(1)-1))
11     {
12         //nothing
13     }
14     else
15     {
16         temp = -rhsData[i + j*width]*h*h - ( 4*matData[i + j*width]
17             - matData[i-1 + j*width] - matData[i+1 + j*width] -
18             matData[i + (j+1)*width] - matData[i + (j-1)*width] );
19     }
20     defectData[i + j*width] = temp;
21 }
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
39 #ifdef __DOUBLE_ALLOWED__
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
59 #endif

```

```

21 __kernel void JacobiCalcDefectD(__global double * matData,
22                                 __global double * rhsData, __global double * defectData,
23                                 unsigned int width, unsigned int height, double h)
24 {
25     int i = get_global_id(0);
26     int j = get_global_id(1);
27     double temp = 0.0;
28     if ((i == 0) || (j == 0) || (i == get_global_size(0)-1) ||
29         (j == get_global_size(1)-1))
30     {
31         //nothing
32     }
33     else
34     {
35         temp = -rhsData[i + j*width]*h*h - ( 4*matData[i + j*width]
36                                             - matData[i-1 + j*width] - matData[i+1 + j*width] -
37                                             matData[i + (j+1)*width] - matData[i + (j-1)*width] );
38     }
39     defectData[i + j*width] = temp;
40 }
41
42 #endif

```

10.1.15 Interpolate kernel

```

1 #include <KernelHeaders.h>
2 __kernel void RefineCTFFF(__global float* fine, __global float
3                           * coarse, unsigned int corWidth)
4 {
5     int i = get_global_id(0);
6     int j = get_global_id(1);
7     float tempData = 0.0;
8
9     //load part of corGrid to shared mem:
10
11    if ((i == 0) || (j == 0) || (i >= get_global_size(0)-1) || (j
12        >= get_global_size(1)-1))
13    {
14        //Nothing, tempData = 0.0;
15    }
16    else if ((i%2 == 0) && (j%2 == 0) )
17    {
18        tempData += 4*coarse[i/2 + j/2*corWidth];
19    }
20    else if (i%2 == 1 && j%2 == 0)
21    {
22        tempData += 4*coarse[i/2 + (j-1)/2*corWidth];
23    }
24    else if (i%2 == 1 && j%2 == 1)
25    {
26        tempData += 4*coarse[(i-1)/2 + j/2*corWidth];
27    }
28
29    fine[i + j*corWidth] = tempData;
30 }

```

```

20  {
21      tempData += 2*coarse[i/2 + j/2*corWidth];
22      tempData += 2*coarse[i/2+1 + j/2*corWidth];
23  }
24  else if (i%2 == 0 && j%2 == 1)
25  {
26      tempData += 2*coarse[i/2 + j/2*corWidth];
27      tempData += 2*coarse[i/2 + j/2*corWidth+corWidth];
28  }
29  else if (i%2 == 1 && j%2 == 1)
30  {
31      tempData += coarse[i/2 + j/2*corWidth];
32      tempData += coarse[i/2 + j/2*corWidth+corWidth];
33      tempData += coarse[i/2+1 + j/2*corWidth];
34      tempData += coarse[i/2+1 + j/2*corWidth+corWidth];
35  }
36
37 fine[i+j*(2*corWidth-1)] = tempData/4.0;
38 }
39
40
41 #ifdef __DOUBLE_ALLOWED__
42 __kernel void RefineCTFFD(__global double* fine, __global
43                           float* coarse, unsigned int corWidth)
44 {
45     int i = get_global_id(0);
46     int j = get_global_id(1);
47     double tempData = 0.0;
48
49     if ((i == 0) || (j == 0) || (i == get_global_size(0)-1) || (
50         j == get_global_size(1)-1))
51     {
52         //Nothing, tempData = 0.0;
53     }
54     else if ((i%2 == 0) && (j%2 == 0) )
55     {
56         tempData += 4*coarse[i/2 + j/2*corWidth];
57     }
58     else if (i%2 == 1 && j%2 == 0)
59     {
60         tempData += 2*coarse[i/2 + j/2*corWidth];
61         tempData += 2*coarse[i/2+1 + j/2*corWidth];
62     }
63     else if (i%2 == 0 && j%2 == 1)
64     {
65         tempData += 2*coarse[i/2 + j/2*corWidth];
66         tempData += 2*coarse[i/2 + j/2*corWidth+corWidth];
67     }
68     else if (i%2 == 1 && j%2 == 1)
69     {
70         tempData += coarse[i/2 + j/2*corWidth];
71         tempData += coarse[i/2 + j/2*corWidth+corWidth];
72     }
73
74     fine[i+j*(2*corWidth-1)] = tempData/4.0;
75 }
76

```

```

67    {
68        tempData += coarse[ i/2 + j/2*corWidth ];
69        tempData += coarse[ i/2 + j/2*corWidth+corWidth ];
70        tempData += coarse[ i/2+1 + j/2*corWidth ];
71        tempData += coarse[ i/2+1 + j/2*corWidth+corWidth ];
72    }
73
74    fine[ i+j*(2*corWidth-1) ] = tempData / 4.0;
75 }
76
77 __kernel void RefineCTFDD( __global double* fine , __global
78                           double* coarse , unsigned int corWidth)
78 {
79     int i = get_global_id(0);
80     int j = get_global_id(1);
81     double tempData = 0.0;
82
83     if ((i == 0) || (j == 0) || (i == get_global_size(0)-1) || (
84         j == get_global_size(1)-1))
84     {
85         //Nothing , tempData = 0.0;
86     }
87     else if ( (i%2 == 0) && (j%2 == 0) )
88     {
89         tempData += 4*coarse[ i/2 + j/2*corWidth ];
90     }
91     else if (i%2 == 1 && j%2 == 0)
92     {
93         tempData += 2*coarse[ i/2 + j/2*corWidth ];
94         tempData += 2*coarse[ i/2+1 + j/2*corWidth ];
95     }
96     else if (i%2 == 0 && j%2 == 1)
97     {
98         tempData += 2*coarse[ i/2 + j/2*corWidth ];
99         tempData += 2*coarse[ i/2 + j/2*corWidth+corWidth ];
100    }
101    else if (i%2 == 1 && j%2 == 1)
102    {
103        tempData += coarse[ i/2 + j/2*corWidth ];
104        tempData += coarse[ i/2 + j/2*corWidth+corWidth ];
105        tempData += coarse[ i/2+1 + j/2*corWidth ];
106        tempData += coarse[ i/2+1 + j/2*corWidth+corWidth ];
107    }
108
109    fine[ i+j*(2*corWidth-1) ] = tempData / 4.0;
110 }
111 #endif

```

10.1.16 Restriction kernel

```

1 #include <KernelHeaders.h>
2 __kernel void RefineFTCFF(__global float* fine, __global float
3 * coarse, unsigned int fineWidth)
4 {
5     int i = get_global_id(0);
6     int j = get_global_id(1);
7     float tempData = 0.0;
8     if ( (i == 0) || (j == 0) || (i == get_global_size(0)-1) ||
9         (j == get_global_size(1)-1) )
10    {
11        //nothing
12    }
13    else
14    {
15        tempData += fine[2*i-1 + 2*j*fineWidth - fineWidth];
16        tempData += 2 * fine[2*i + 2*j*fineWidth - fineWidth];
17        tempData += fine[2*i+1 + 2*j*fineWidth - fineWidth];
18        tempData += 2 * fine[2*i-1 + 2*j*fineWidth];
19        tempData += 4 * fine[2*i + 2*j*fineWidth];
20        tempData += 2 * fine[2*i+1 + 2*j*fineWidth];
21        tempData += fine[2*i-1 + 2*j*fineWidth + fineWidth];
22        tempData += 2 * fine[2*i + 2*j*fineWidth + fineWidth];
23        tempData += fine[2*i+1 + 2*j*fineWidth + fineWidth];
24    }
25    coarse[i+j*(fineWidth/2+1)] = tempData/16.0;
26
27
28
29 #ifdef __DOUBLE_ALLOWED__
30 __kernel void RefineFTCDF(__global double* fine, __global
31 float* coarse, unsigned int fineWidth)
32 {
33     int i = get_global_id(0);
34     int j = get_global_id(1);
35     float tempData = 0.0;
36     if ( (i == 0) || (j == 0) || (i == get_global_size(0)-1) ||
37         (j == get_global_size(1)-1) )
38     {
39        //nothing
40    }
41    tempData += fine[2*i-1 + 2*j*fineWidth - fineWidth];

```

```

42     tempData += 2 * fine[2*i    + 2*j*fineWidth - fineWidth];
43     tempData +=      fine[2*i+1 + 2*j*fineWidth - fineWidth];
44     tempData += 2 * fine[2*i-1 + 2*j*fineWidth];
45     tempData += 4 * fine[2*i    + 2*j*fineWidth];
46     tempData += 2 * fine[2*i+1 + 2*j*fineWidth];
47     tempData +=      fine[2*i-1 + 2*j*fineWidth + fineWidth];
48     tempData += 2 * fine[2*i    + 2*j*fineWidth + fineWidth];
49     tempData +=      fine[2*i+1 + 2*j*fineWidth + fineWidth];
50 }
51
52 coarse[i+j*(fineWidth/2+1)] = tempData/16.0;
53 }
54
55 __kernel void RefineFTCDD(__global double* fine, __global
56                           double* coarse, unsigned int fineWidth)
57 {
58     int i = get_global_id(0);
59     int j = get_global_id(1);
60     double tempData = 0.0;
61     if ((i == 0) || (j == 0) || (i == get_global_size(0)-1) ||
62         (j == get_global_size(1)-1))
63     {
64         //nothing
65     }
66     else
67     {
68         tempData +=      fine[2*i-1 + 2*j*fineWidth - fineWidth];
69         tempData += 2 * fine[2*i    + 2*j*fineWidth - fineWidth];
70         tempData +=      fine[2*i+1 + 2*j*fineWidth - fineWidth];
71         tempData += 2 * fine[2*i-1 + 2*j*fineWidth];
72         tempData += 4 * fine[2*i    + 2*j*fineWidth];
73         tempData += 2 * fine[2*i+1 + 2*j*fineWidth];
74         tempData +=      fine[2*i-1 + 2*j*fineWidth + fineWidth];
75     }
76
77     coarse[i+j*(fineWidth/2+1)] = tempData/16.0;
78 }
79 #endif

```

10.1.17 KernelHeaders

```

1 #ifdef __DOUBLE_ALLOWED__
2 #pragma OPENCL EXTENSION cl_khr_fp64 : enable
3 #endif

```

10.2 Mex Bindings

All MEX bindings are provided here:

10.2.1 MexInitOpenCL.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3
4 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
                  mxArray *prhs[])
5 {
6     if (nrhs != 0)
7     {
8         mexPrintf("Failed to init OpenCL\n");
9         return;
10    }
11    void * tempMng = mxMalloc(sizeof(OpenCLManager));
12    OpenCLManager * __OpenCLManager__ = new(tempMng)
13        OpenCLManager();
14    mexMakeMemoryPersistent(tempMng);
15    const mwSize rows = 1;
16    plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,mxREAL
17                                  );
18    size_t * data = (size_t *) mxGetData(plhs[0]);
19    *data = (size_t)tempMng;
20 }
```

10.2.2 MexReleaseOpenCL.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3
4 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
                  mxArray *prhs[])
5 {
6     if (nrhs != 1)
7     {
8         mexPrintf("Handle not provided");
9         return;
10    }
11    //get opencl handle:
12    size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
13    void * temp = (void *)*tempPtr;
```

```
14     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
15     __OpenCLManager__->ReleaseOpenCL();
16     mxFree(__OpenCLManager__);
17 }
```

10.2.3 MexSetGPU.cpp

```
1 #include "mex.h"
2 #include "OpenCLManager.h"
3
4 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
5 mxArray *prhs[])
6 {
7     if (nrhs != 2 && nrhs != 3)
8     {
9         mexPrintf("Handle not provided");
10    return;
11 }
12 //Get OpenCL handle:
13 uintptr_t * tempPtr = (uintptr_t *)mxGetData(prhs[0]);
14 void * temp = (void *)*tempPtr;
15 OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
16 //Get value:
17 double * tempPtrVal = (double *)mxGetData(prhs[1]);
18 double type = *tempPtrVal + 0.1;
19 if (nrhs == 3)
20 {
21     double * tempPtrRes = (double *)mxGetData(prhs[2]);
22     double res = *tempPtrRes + 0.1;
23     if (((unsigned int)res) == 1)
24     {
25         __OpenCLManager__->AllowDouble();
26     }
27     __OpenCLManager__->SetActiveGPU((unsigned int)type);
28 }
```

10.2.4 MexPrintGPU.cpp

```
1 #include "mex.h"
2 #include "OpenCLManager.h"
3
```

```

4 void mexFunction(int nlhs , mxArray *plhs[ ] ,int nrhs , const
      mxArray *prhs[ ])
5 {
6   size_t * tempPtr = (size_t *)mxGetData(prhs[0]) ;
7   void * temp = (void *)*tempPtr ;
8   OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp ;
9   __OpenCLManager__->PrintGPUs() ;
10 }
```

10.2.5 MexHandleMatrix.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "Matrix.h"
5
6 /**
7 //Function for getting data from the GPU vectors to the cpu
     vectors .
8 /**
9
10 void mexFunction(int nlhs , mxArray *plhs[ ] ,int nrhs , const
      mxArray *prhs[ ])
11 {
12   if (nrhs != 2)
13   {
14     mexPrintf("array handle not provided. Input: Handle ,
           Approximation , rhs\n");
15     return ;
16   }
17
18 //get opencl handle:
19 size_t * tempPtr = (size_t *)mxGetData(prhs[0]) ;
20 void * temp = (void *)*tempPtr ;
21 OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp ;
22
23 //get vector handle:
24 tempPtr = (size_t *)mxGetData(prhs[1]) ;
25 size_t type = tempPtr[1];
26
27 if (type == 0) //float
28 {
29   Matrix<float> * uh = (Matrix<float> *)((void *)tempPtr[0])
           ;
30 }
```

```

31     plhs[0] = mxCreateDoubleMatrix(uh->GetWidth() ,uh->
32         GetHeight() ,mxREAL) ;
33     uh->SyncBuffers () ;
34     double * data = (double *) mxGetData( plhs[0]) ;
35     float * input = uh->GetData() ;
36     for (unsigned int i = 0; i < uh->GetLength() ; i += 1)
37     {
38         data[ i ] = (double)input[ i ] ;
39     }
40 }
41 else if (type == 1) //double
42 {
43     Matrix<double> * uh = (Matrix<double> *)(( void *)tempPtr
44         [0]) ;
45     uh->SyncBuffers () ;
46     plhs[0] = mxCreateDoubleMatrix(uh->GetWidth() ,uh->
47         GetHeight() ,mxREAL) ;
48     double * data = (double *) mxGetData( plhs[0]) ;
49     double * input = uh->GetData() ;
50     for (unsigned int i = 0; i < uh->GetLength() ; i += 1)
51     {
52         data[ i ] = input[ i ] ;
53     }
54 }
```

10.2.6 MexMatrix.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "Matrix.h"
5
6 /**
7 //Get a matrix from matlab and put it on the gpu.
8 /**
9
10 void mexFunction(int nlhs , mxArray *plhs[ ] ,int nrhs , const
11                 mxArray *prhs[ ])
12 {
13     if (nrhs != 2)
14     {
15         mexPrintf("Handle or vector not provided");
16     }

```

```

17 size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
18 void * temp = (void *)*tempPtr;
19 OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
20
21 if (mxIsDouble(prhs[1]))
22 {
23     int M = mxGetM(prhs[1]);
24     int N = mxGetN(prhs[1]);
25     double * vecPtr = (double *)mxGetPr(prhs[1]);
26
27 //populate gpu:
28 void * tempMat = mxMalloc(sizeof(Matrix<double>));
29 Matrix<double> * matrix = (Matrix<double> *)tempMat;
30 matrix->InitMatrix(__OpenCLManager__, M, N, (double *)vecPtr);
31 matrix->Type = 1;
32 mexMakeMemoryPersistent(tempMat);
33
34
35 //return handle to gpu vector:
36 const mwSize rows = 2;
37 plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
38                               mxREAL);
39 size_t * data = (size_t *) mxGetData(plhs[0]);
40 data[0] = (size_t)tempMat;
41 data[1] = 1; //double
42 }
43 else if (mxIsSingle(prhs[1]))
44 {
45     int N = mxGetN(prhs[1]);
46     int M = mxGetM(prhs[1]);
47
48     float * vecPtr = (float *)mxGetPr(prhs[1]);
49
50 //populate gpu:
51 void * tempMat = mxMalloc(sizeof(Matrix<float>));
52 Matrix<float> * matrix = (Matrix<float> *)tempMat;
53 matrix->InitMatrix(__OpenCLManager__, M, N, (float *)vecPtr);
54 matrix->Type = 0;
55 mexMakeMemoryPersistent(tempMat);
56
57 //return handle to gpu vector:
58 const mwSize rows = 2;
59 plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
60                               mxREAL);
61 size_t * data = (size_t *) mxGetData(plhs[0]);
62 data[0] = (size_t)tempMat;
63 data[1] = 0; //float

```

```
62  }
63 }
```

10.2.7 MexReleaseMatrix.cpp

```
1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "Matrix.h"
5
6 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
    mxArray *prhs[])
7 {
8     if (nrhs != 2)
9     {
10         mexPrintf("matrix not provided");
11         return;
12     }
13 //Get OpenCL handle:
14 uintptr_t * handlePtr = (uintptr_t *)mxGetData(prhs[0]);
15 void * temp = (void *)*handlePtr;
16 OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
17 //get vector handle:
18 size_t * tempPtr = (size_t *)mxGetData(prhs[1]);
19 size_t type = tempPtr[1];
20 if (type == 0) //float
21 {
22     Matrix<float> * mat = (Matrix<float> *)((void *)tempPtr
23         [0]);
24     __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
25     mxFree(mat);
26 } else if (type == 1) //double
27 {
28     Matrix<double> * mat = (Matrix<double> *)((void *)tempPtr
29         [0]);
30     __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
31     mxFree(mat);
32 }
```

10.2.8 MexBandMatrix.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "BandMatrix.h"
5
6 /**
7 //Get a matrix from matlab and put it on the gpu.
8 /**
9
10 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
11                 mxArray *prhs[])
12 {
13     if (nrhs != 2)
14     {
15         mexPrintf("Handle or vector not provided");
16         return;
17     }
18     size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
19     void * temp = (void *)tempPtr;
20     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
21
22     if (mxIsDouble(prhs[1]))
23     {
24         int M = mxGetM(prhs[1]);
25         int N = mxGetN(prhs[1]);
26         double * vecPtr = (double *)mxGetPr(prhs[1]);
27         mexPrintf("N %u, M %u\n", N, M);
28
29         //populate gpu:
30         void * tempMat = mxMalloc(sizeof(BandMatrix<double>));
31         BandMatrix<double> * matrix = (BandMatrix<double> *)
32             tempMat;
33         matrix->InitMatrix(__OpenCLManager__, M, M, N/2, (double *)
34             vecPtr);
35         mexMakeMemoryPersistent(tempMat);
36
37         //return handle to gpu vector:
38         const mwSize rows = 2;
39         plhs[0] = mxCreateNumericArray(1, &rows, mxUINT64_CLASS,
40             mxREAL);
41         size_t * data = (size_t *) mxGetData(plhs[0]);
42         data[0] = (size_t)tempMat;
43         data[1] = 1; //double
44     }
45     else if (mxIsSingle(prhs[1]))
46     {
47         int N = mxGetN(prhs[1]);
48         int M = mxGetM(prhs[1]);

```

```

46
47     float * vecPtr = (float *)mxGetPr(prhs[1]);
48
49 //populate gpu:
50 void * tempMat = mxMalloc(sizeof(BandMatrix<float>));
51 BandMatrix<float> * matrix = (BandMatrix<float> *)tempMat;
52 matrix->InitMatrix(__OpenCLManager__,M, N, N/2, (float *)
53     vecPtr);
54 mexMakeMemoryPersistent(tempMat);
55 //return handle to gpu vector:
56 const mwSize rows = 2;
57 plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
58     mxREAL);
59 size_t * data = (size_t *) mxGetData(plhs[0]);
60 data[0] = (size_t)tempMat;
61 data[1] = 0; //float
62 }
```

10.2.9 MexReleaseBandMatrix.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "BandMatrix.h"
5
6 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
    mxArray *prhs[])
7 {
8     if (nrhs != 2)
9     {
10         mexPrintf("matrix not provided");
11         return;
12     }
13 //Get OpenCL handle:
14 uintptr_t * handlePtr = (uintptr_t *)mxGetData(prhs[0]);
15 void * temp = (void *)*handlePtr;
16 OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
17 //get vector handle:
18 size_t * tempPtr = (size_t *)mxGetData(prhs[1]);
19 size_t type = tempPtr[1];
20 if (type == 0) //float
21 {
22     BandMatrix<float> * mat = (BandMatrix<float> *)((void *)
        tempPtr[0]);
```

```

23     __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
24     mxFree(mat);
25 }
26 else if (type == 1) //double
27 {
28     BandMatrix<double> * mat = (BandMatrix<double> *)((void *)tempPtr[0]);
29     __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
30     mxFree(mat);
31 }
32 }
```

10.2.10 MexSparseMatrix.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "SparseMatrix.h"
4
5 /**
6 //Get a matrix from matlab and put it on the gpu.
7 /**
8
9 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
10                 mxArray *prhs[])
11 {
12     if (nrhs < 2)
13     {
14         mexPrintf("Handle or vector not provided");
15         return;
16     }
17     size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
18     size_t * rows = mxGetIr(prhs[1]);
19     unsigned int nnz = (unsigned int)mxGetNzmax(prhs[1]);
20     size_t * columns = mxGetJc(prhs[1]);
21     void * temp = (void *)tempPtr;
22     int M = mxGetM(prhs[1]);
23     int N = mxGetN(prhs[1]);
24     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
25
26     bool isDouble = true;
27     if (nrhs > 2)
28     {
29         double * isDoublePtr = (double *)mxGetData(prhs[2]);
30         isDouble = (*isDoublePtr + 0.1) < 1;
31     }
32     if (isDouble)
```

```

32  {
33      double * vecPtr = (double *)mxGetPr(prhs[1]);
34
35 //populate gpu:
36 void * tempMat = mxMalloc(sizeof(SparseMatrix<double>));
37 SparseMatrix<double> * matrix = (SparseMatrix<double> *)
38     tempMat;
39 matrix->InitMatrix(__OpenCLManager__,M, N, nnz, rows,
40     columns, (double *)vecPtr);
41 mexMakeMemoryPersistent(tempMat);
42
43 //return handle to gpu vector:
44 const mwSize rows = 2;
45 plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
46     mxREAL);
47 size_t * data = (size_t *) mxGetData(plhs[0]);
48 data[0] = (size_t)tempMat;
49 data[1] = 1; //double
50 }
51 else
52 {
53     double * vecPtr = (double *)mxGetPr(prhs[1]);
54     float * vecFloatPtr = (float *)malloc(sizeof(float)*nnz);
55     for (unsigned int i = 0; i < nnz; i += 1)
56     {
57         vecFloatPtr[i] = (float)vecPtr[i];
58     }
59 //populate gpu:
60 void * tempMat = mxMalloc(sizeof(SparseMatrix<float>));
61 SparseMatrix<float> * matrix = (SparseMatrix<float> *)
62     tempMat;
63 matrix->InitMatrix(__OpenCLManager__,M, N, nnz, rows,
64     columns, (float *)vecFloatPtr);
65 mexMakeMemoryPersistent(tempMat);
66
67 //return handle to gpu vector:
68 const mwSize rows = 2;
69 plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
70     mxREAL);
71 size_t * data = (size_t *) mxGetData(plhs[0]);
72 data[0] = (size_t)tempMat;
73 data[1] = 0; //float
74 }
75 }

```

10.2.11 MexReleaseSparseMatrix.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "SparseMatrix.h"
5
6 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
    mxArray *prhs[])
7 {
8     if (nrhs != 2)
9     {
10         mexPrintf("matrix not provided");
11         return;
12     }
13     //Get OpenCL handle:
14     uintptr_t * handlePtr = (uintptr_t *)mxGetData(prhs[0]);
15     void * temp = (void *)*handlePtr;
16     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
17     //get vector handle:
18     size_t * tempPtr = (size_t *)mxGetData(prhs[1]);
19     size_t type = tempPtr[1];
20     if (type == 0) //float
21     {
22         SparseMatrix<float> * mat = (SparseMatrix<float> *)((void
            *)tempPtr[0]);
23         __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
24         __OpenCLManager__->DeleteIndex(mat->GetRowControl());
25         __OpenCLManager__->DeleteIndex(mat->GetColumnControl());
26         mxFree(mat);
27     }
28     else if (type == 1) //double
29     {
30         SparseMatrix<double> * mat = (SparseMatrix<double> *)((
            void *)tempPtr[0]);
31         __OpenCLManager__->DeleteMemory(mat->GetMemoryControl());
32         __OpenCLManager__->DeleteIndex(mat->GetRowControl());
33         __OpenCLManager__->DeleteIndex(mat->GetColumnControl());
34         mxFree(mat);
35     }
36 }
```

10.2.12 MexBandMatrixVector.cpp

```

1 #include "mex.h"
```

```

2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "Matrix.h"
5 #include "BandMatrix.h"
6
7 //
8 //Function for using jacobi and gauss-seidel smoothers.
9 //Arguments:
10 // 0: OpenCL handle.
11 // 1: uh
12 // 2: RHS
13 // 3: h (optional).
14 //
15 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
    mxArray *prhs[])
16 {
17     float time;
18     cl_event event;
19     double h = -1.0;
20     if (nrhs < 2 || nrhs > 6)
21     {
22         mexPrintf("Handle or arrays not provided. Input: Handle,
            Approximation, rhs\n");
23         return;
24     }
25     //get opencl handle:
26     size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
27     void * temp = (void *)tempPtr;
28     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
29
30     //Check if RPT and TPG are provided:
31     if (nrhs ≥ 6)
32     {
33         size_t RowsPerThread = (size_t)(*mxGetPr(prhs[4]) + 0.1);
34         size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[5]) + 0.1);
35         __OpenCLManager__->SetBandMatrixVectorRowsPerThread(
            RowsPerThread);
36         __OpenCLManager__->SetBandMatrixVectorThreadsPerGroup(
            ThreadsPerGroup);
37     }
38
39
40     //get vector handle:
41     size_t * tempBandPtr = (size_t *)mxGetData(prhs[1]);
42     size_t type = tempBandPtr[1];
43
44     size_t * tempVecPtr = (size_t *)mxGetData(prhs[2]);
45     size_t typeRHS = tempVecPtr[1];
46

```

```

47
48     if (type == 0 && typeRHS == 0) //float
49     {
50         BandMatrix<float> * band = (BandMatrix<float> *)((void *)tempBandPtr[0]);
51         Matrix<float> * vec = (Matrix<float> *)((void *)tempVecPtr[0]);
52
53         Matrix<float> * out = vec;
54         if (nrhs >= 3)
55         {
56             size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
57             out = (Matrix<float> *)((void *)tempOutPtr[0]);
58         }
59         else if (nlhs == 1) //need to create a matrix to send
60             out:
61             {
62                 //populate gpu:
63                 out = (Matrix<float> *)mxMalloc(sizeof(Matrix<float>));
64                 out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec->GetHeight(), (float *)NULL);
65                 mexMakeMemoryPersistent(out);
66
67                 //return handle to gpu vector:
68                 const mwSize rows = 2;
69                 plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
70                                               mxREAL);
71                 size_t * data = (size_t *)mxGetData(plhs[0]);
72                 data[0] = (size_t)out;
73                 data[1] = 0; //float
74             }
75
76         __OpenCLManager__->BandMatrixVectorFF(band->GetDataBuffer()
77                                                 (), vec->GetDataBuffer(), out->GetDataBuffer(), band->GetHeight(),
78                                                 band->GetBandwidth(), band->GetLength(),
79                                                 &event);
80
81     } else if (type == 1 && typeRHS == 1) //doubles
82     {
83         BandMatrix<double> * band = (BandMatrix<double> *)((void *)tempBandPtr[0]);
84         Matrix<double> * vec = (Matrix<double> *)((void *)tempVecPtr[0]);
85
86         Matrix<double> * out = vec;
87         if (nrhs >= 3)
88         {
89             size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);

```

```

85      out = (Matrix<double> *)(( void *)tempOutPtr[0]);
86    }
87    else if (nlhs == 1) //need to create a matrix to send
88    {
89      //populate gpu:
90      out = (Matrix<double> *)mxMalloc(sizeof(Matrix<
91      double>));
91      out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec
92      ->GetHeight(), (double *)NULL);
92      mexMakeMemoryPersistent(out);
93
94      //return handle to gpu vector:
95      const mwSize rows = 2;
96      plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS
97      ,mxREAL);
98      size_t * data = (size_t *)mxGetData(plhs[0]);
99      data[0] = (size_t)out;
100     data[1] = 1; //double
100   }
101
102   __OpenCLManager__->BandMatrixVectorDD(band->GetDataBuffer
103   (), vec->GetDataBuffer(), out->GetDataBuffer(), band->
104   GetHeight(), band->GetBandwidth(), vec->GetLength(), &
105   event);
106
106 //Outgoing matrix provided and we return time instead.
107 if (nlhs == 1 && nrhs ≥ 3)
107 {
108   __OpenCLManager__->WaitForCPU();
109   float time = __OpenCLManager__->GetExecutionTime(&event);
110   //return handle to gpu vector:
111   const mwSize rows = 1;
112   plhs[0] = mxCreateNumericArray(1,&rows,mxDLUBLE_CLASS,
113   mxREAL);
113   double * data = (double *)mxGetData(plhs[0]);
114   *data = (double)time;
115 }
116   clReleaseEvent(event);
117 }
```

10.2.13 MexSparseMatrixVector.cpp

```

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

```

3 #include "MathVector.h"
4 #include "Matrix.h"
5 #include "SparseMatrix.h"
6
7 /**
8 //Function for using jacobi and gauss-seidel smoothers.
9 //Arguments:
10 //0: OpenCL handle.
11 //1: uh
12 //2: RHS
13 //3: h (optional).
14 /**
15 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
    mxArray *prhs[])
16 {
17     double h = -1.0;
18     if (nrhs < 3 || nrhs > 7)
19     {
20         mexPrintf("Handle or arrays not provided. Input: Handle,
            Approximation, rhs\n");
21         return;
22     }
23     //get opencl handle:
24     size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
25     void * temp = (void *)*tempPtr;
26     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
27
28     //get vector handle:
29     size_t * tempSparsePtr = (size_t *)mxGetData(prhs[1]);
30     size_t type = tempSparsePtr[1];
31
32     size_t * tempVecPtr = (size_t *)mxGetData(prhs[2]);
33     size_t typeRHS = tempVecPtr[1];
34
35     if (nrhs ≥ 6)
36     {
37         size_t RowsPerThread = (size_t)(*mxGetPr(prhs[4]) + 0.1);
38         size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[5]) + 0.1);
39         __OpenCLManager__->SetSparseMatrixVectorRowsPerThread(
            RowsPerThread);
40         __OpenCLManager__->SetSparseMatrixVectorThreadsPerGroup(
            ThreadsPerGroup);
41     }
42
43
44     cl_event event;
45     if (type == 0 && typeRHS == 0) //float
46 {

```

```

47     SparseMatrix<float> * sparse = (SparseMatrix<float> *)((  
48         void *)tempSparsePtr[0]);  
49     Matrix<float> * vec = (Matrix<float> *)((void *)tempVecPtr  
50         [0]);  
51     //populate gpu:  
52     Matrix<float> * out;  
53     if (nrhs≥ 4)  
54     {  
55         size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);  
56         out = (Matrix<float> *)((void *)tempOutPtr[0]);  
57     }  
58     else  
59     {  
60         out = (Matrix<float> *)mxMalloc(sizeof(Matrix<float>)  
61             );  
62         out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec  
63             ->GetHeight(), (float *)NULL);  
64         mexMakeMemoryPersistent(out);  
65     }  
66     __OpenCLManager__->SparseMatrixVectorFF(sparse->  
67         GetDataBuffer(), sparse->GetColumnIndexBuffer(),  
68         sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out  
69         ->GetDataBuffer(), sparse->GetHeight(), sparse->  
70         GetWidth(), sparse->GetLength(), sparse->  
71         GetRowVectorLength(), &event);  
72     __OpenCLManager__->WaitForCPU();  
73     if (nrhs<4)  
74     {  
75         if (nlhs == 1) //need to create a matrix to send out:  
76         {  
77             //return handle to gpu vector:  
78             const mwSize rows = 2;  
79             plhs[0] = mxCreateNumericArray(1,&rows,  
80                 mxUINT64_CLASS,mxREAL);  
81             size_t * data = (size_t *)mxGetData(plhs[0]);  
82             data[0] = (size_t)out;  
83             data[1] = 0; //float  
84         }  
85     }  
86     else  
87     {  
88         __OpenCLManager__->DeleteMemory(vec->  
89             GetMemoryControl());  
90         vec->SetMemoryControl(out->GetMemoryControl());  
91         mxFree(out);  
92     }  
93 }
```

```

85      else if (nlhs == 1) //send out timing:
86    {
87      //return handle to gpu vector:
88      const mwSize rows = 1;
89      plhs[0] = mxCreateNumericArray(1,&rows ,mxDOUBLE_CLASS
90                                     ,mxREAL);
91      double * data = (double *)mxGetData(plhs[0]);
92      __OpenCLManager__->WaitForCPU();
93      float time = __OpenCLManager__->GetExecutionTime(&event)
94      ;
95      data[0] = time;
96    }
97  else if (type == 1 && typeRHS == 1) //doubles
98  {
99    SparseMatrix<double> * sparse = (SparseMatrix<double> *)((
100      void *)tempSparsePtr[0]);
101    Matrix<double> * vec = (Matrix<double> *)((void *)
102      tempVecPtr[0]);
103    Matrix<double> * out;
104    if (nrhs >= 4)
105    {
106      size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
107      out = (Matrix<double> *)((void *)tempOutPtr[0]);
108    }
109    else
110    {
111      out = (Matrix<double> *)mxMalloc(sizeof(Matrix<double
112                                     >));
113      out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec
114                        ->GetHeight(), (double *)NULL);
115      mexMakeMemoryPersistent(out);
116    }
117    __OpenCLManager__->SparseMatrixVectorDD(sparse->
118                                              GetDataBuffer(), sparse->GetColumnIndexBuffer(),
119                                              sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out
120                                              ->GetDataBuffer(), sparse->GetHeight(), sparse->
121                                              GetWidth(), sparse->GetLength(), sparse->
122                                              GetRowVectorLength(), &event);
123    __OpenCLManager__->WaitForCPU();
124    if (nrhs < 4)
125    {
126      if (nlhs == 1) //need to create a matrix to send out:
127      {
128        //return handle to gpu vector:
129        const mwSize rows = 2;

```

```

123         plhs[0] = mxCreateNumericArray(1,&rows ,
124                                         mxUINT64_CLASS,mxREAL);
125         size_t * data = (size_t *)mxGetData( plhs[0]) ;
126         data[0] = (size_t)out;
127         data[1] = 1; //double
128     }
129     {
130         //Clean up:
131         __OpenCLManager__->DeleteMemory( vec->
132                                         GetMemoryControl()) ;
133         vec->SetMemoryControl(out->GetMemoryControl());
134         mxFree(out);
135     }
136     else if ( nlhs == 1) //send out timing:
137     {
138         //return handle to gpu vector:
139         const mwSize rows = 1;
140         plhs[0] = mxCreateNumericArray(1,&rows ,mxDOUBLE_CLASS
141                                         ,mxREAL);
142         double * data = (double *)mxGetData( plhs[0]) ;
143         float time = __OpenCLManager__->GetExecutionTime(&event)
144         ;
145         data[0] = time;
146     }
147     else if (type == 1 && typeRHS == 0) //double sparse and
148         float vector given. Return should be double.
149     {
150         SparseMatrix<double> * sparse = (SparseMatrix<double> *)((
151             void *)tempSparsePtr[0]);
152         Matrix<float> * vec = (Matrix<float> *)((void *)tempVecPtr
153             [0]);
154         Matrix<double> * out;
155         if (nrhs≥ 4)
156         {
157             size_t * tempOutPtr = (size_t *)mxGetData( prhs[3]);
158             out = (Matrix<double> *)((void *)tempOutPtr[0]);
159         }
160         else
161         {
162             out = (Matrix<double> * )mxMalloc( sizeof(Matrix<double
163                 >));
164             out->InitMatrix(__OpenCLManager__, vec->GetWidth() , vec
165                 ->GetHeight() , (double *)NULL);
166             mexMakeMemoryPersistent(out);

```

```

163     }
164
165     __OpenCLManager__->SparseMatrixVectorDF( sparse->
166         GetDataBuffer() , sparse->GetColumnIndexBuffer() ,
167         sparse->GetRowIndexBuffer() , vec->GetDataBuffer() , out
168         ->GetDataBuffer() , sparse->GetHeight() , sparse->
169         GetWidth() , sparse->GetLength() , sparse->
170         GetRowVectorLength() , &event );
171
172     __OpenCLManager__->WaitForCPU();
173
174     if ( nrhs < 4 )
175     {
176         if ( nlhs == 1 ) //need to create a matrix to send out:
177         {
178             //return handle to gpu vector:
179             const mwSize rows = 2;
180             plhs[0] = mxCreateNumericArray(1,&rows,
181                 mxUINT64_CLASS,mxREAL);
182             size_t * data = (size_t *)mxGetData( plhs[0] );
183             data[0] = (size_t)out;
184             data[1] = 1; //double
185         }
186         else if ( nlhs == 1 ) //send out timing:
187         {
188             //return handle to gpu vector:
189             const mwSize rows = 1;
190             plhs[0] = mxCreateNumericArray(1,&rows ,mxDOUBLE_CLASS
191                 ,mxREAL);
192             double * data = (double *)mxGetData( plhs[0] );
193             float time = __OpenCLManager__->GetExecutionTime(&event)
194             ;
195             data[0] = time;
196         }
197     }
198     else if ( type == 0 && typeRHS == 1 ) //float sparse and
199         double vector given. Return should be float.
200     {
201         SparseMatrix<float> * sparse = (SparseMatrix<float> *)((
202             void *)tempSparsePtr[0]);
203         Matrix<double> * vec = (Matrix<double> *)((void *)
204             tempVecPtr[0]);
205
206         Matrix<float> * out;
207         if ( nrhs ≥ 4 )
208         {
209             size_t * tempOutPtr = (size_t *)mxGetData( prhs[3] );
210             out = (Matrix<float> *)((void *)tempOutPtr[0]);
211         }

```

```

201     else
202     {
203         out = (Matrix<float> * )mxMalloc( sizeof(Matrix<float>)
204                                         );
204         out->InitMatrix(__OpenCLManager__, vec->GetWidth() , vec
205                           ->GetHeight() , (float *)NULL);
205         mexMakeMemoryPersistent(out);
206     }
207
208     __OpenCLManager__->SparseMatrixVectorFD( sparse->
209                                               GetDataBuffer() , sparse->GetColumnIndexBuffer() ,
210                                               sparse->GetRowIndexBuffer() , vec->GetDataBuffer() , out
211                                               ->GetDataBuffer() , sparse->GetHeight() , sparse->
212                                               GetWidth() , sparse->GetLength() , sparse->
213                                               GetRowVectorLength() , &event);
214     __OpenCLManager__->WaitForCPU();
215     if (nrhs<4)
216     {
217         if (nlhs == 1) //need to create a matrix to send out:
218         {
219             //return handle to gpu vector:
220             const mwSize rows = 2;
221             plhs[0] = mxCreateNumericArray(1,&rows ,
222                                           mxUINT64_CLASS,mxREAL);
223             size_t * data = (size_t *)mxGetData( plhs[0] );
224             data[0] = (size_t)out;
225             data[1] = 0; //double
226         }
227     }
228     else if (nlhs == 1) //send out timing:
229     {
230         //return handle to gpu vector:
231         const mwSize rows = 1;
232         plhs[0] = mxCreateNumericArray(1,&rows ,mxDOUBLE_CLASS
233                                       ,mxREAL);
234         double * data = (double *)mxGetData( plhs[0] );
235
236         float time = __OpenCLManager__->GetExecutionTime(&event )
237                     ;
238         data[0] = time;
239     }
240     clReleaseEvent(event);
241 }
```

10.2.14 MexSparseMatrixVector.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "Matrix.h"
5 #include "SparseMatrix.h"
6
7 /**
8 //Function for using jacobi and gauss-seidel smoothers.
9 //Arguments:
10 //0: OpenCL handle.
11 //1: uh
12 //2: RHS
13 //3: h (optional).
14 /**
15 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
16 mxArray *prhs[])
17 {
18     double h = -1.0;
19     if (nrhs < 3 || nrhs > 7)
20     {
21         mexPrintf("Handle or arrays not provided. Input: Handle,
22                 Approximation, rhs\n");
23         return;
24     }
25     //get opencl handle:
26     size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
27     void * temp = (void *)*tempPtr;
28     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
29
30     //get vector handle:
31     size_t * tempSparsePtr = (size_t *)mxGetData(prhs[1]);
32     size_t type = tempSparsePtr[1];
33
34     size_t * tempVecPtr = (size_t *)mxGetData(prhs[2]);
35     size_t typeRHS = tempVecPtr[1];
36
37     if (nrhs ≥ 6)
38     {
39         size_t RowsPerThread = (size_t)(*mxGetPr(prhs[4]) + 0.1);
40         size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[5]) + 0.1);
41         __OpenCLManager__->SetSparseMatrixVectorRowsPerThread(
42             RowsPerThread);
43         __OpenCLManager__->SetSparseMatrixVectorThreadsPerGroup(
44             ThreadsPerGroup);
45     }
46
47     cl_event event;

```

```

45 if (type == 0 && typeRHS == 0) //float
46 {
47     SparseMatrix<float> * sparse = (SparseMatrix<float> *)((
48         void *)tempSparsePtr[0]);
49     Matrix<float> * vec = (Matrix<float> *)((void *)tempVecPtr
50 [0]);
51
52     //populate gpu:
53     Matrix<float> * out;
54     if (nrhs >= 4)
55     {
56         size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
57         out = (Matrix<float> *)((void *)tempOutPtr[0]);
58     }
59     else
60     {
61         out = (Matrix<float> *)mxMalloc(sizeof(Matrix<float>));
62         out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec
63             ->GetHeight(), (float *)NULL);
64         mexMakeMemoryPersistent(out);
65     }
66
67     __OpenCLManager__->SparseMatrixVectorFF(sparse->
68         GetDataBuffer(), sparse->GetColumnIndexBuffer(),
69         sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out
70         ->GetDataBuffer(), sparse->GetHeight(), sparse->
71         GetWidth(), sparse->GetLength(), sparse->
72         GetRowVectorLength(), &event);
73     __OpenCLManager__->WaitForCPU();
74
75     if (nrhs < 4)
76     {
77         if (nlhs == 1) //need to create a matrix to send out:
78         {
79             //return handle to gpu vector:
80             const mwSize rows = 2;
81             plhs[0] = mxCreateNumericArray(1, &rows,
82                 mxUINT64_CLASS, mxREAL);
83             size_t * data = (size_t *)mxGetData(plhs[0]);
84             data[0] = (size_t)out;
85             data[1] = 0; //float
86         }
87         else
88         {
89             __OpenCLManager__->DeleteMemory(vec->
90                 GetMemoryControl());
91             vec->SetMemoryControl(out->GetMemoryControl());
92             mxFree(out);
93         }
94     }
95 }
```

```

83         }
84     }
85     else if (nlhs == 1) //send out timing:
86     {
87         //return handle to gpu vector:
88         const mwSize rows = 1;
89         plhs[0] = mxCreateNumericArray(1,&rows,mxDLBLE_CLASS
90                                         ,mxREAL);
91         double * data = (double *)mxGetData(plhs[0]);
92         __OpenCLManager__->WaitForCPU();
93         float time = __OpenCLManager__->GetExecutionTime(&event)
94             ;
95         data[0] = time;
96     }
97     else if (type == 1 && typeRHS == 1) //doubles
98     {
99         SparseMatrix<double> * sparse = (SparseMatrix<double> *)((
100           void *)tempSparsePtr[0]);
101         Matrix<double> * vec = (Matrix<double> *)((void *)
102           tempVecPtr[0]);
103         Matrix<double> * out;
104         if (nrhs >= 4)
105         {
106             size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
107             out = (Matrix<double> *)((void *)tempOutPtr[0]);
108         }
109         else
110         {
111             out = (Matrix<double> *)mxMalloc(sizeof(Matrix<double
112                                         >));
113             out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec
114                             ->GetHeight(), (double *)NULL);
115             mexMakeMemoryPersistent(out);
116         }
117         __OpenCLManager__->SparseMatrixVectorDD(sparse->
118             GetDataBuffer(), sparse->GetColumnIndexBuffer(),
119             sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out
120             ->GetDataBuffer(), sparse->GetHeight(), sparse->
121             GetWidth(), sparse->GetLength(), sparse->
122             GetRowVectorLength(), &event);
123         __OpenCLManager__->WaitForCPU();
124         if (nrhs < 4)
125         {
126             if (nlhs == 1) //need to create a matrix to send out:
127             {

```

```

121         //return handle to gpu vector:
122         const mwSize rows = 2;
123         plhs[0] = mxCreateNumericArray(1,&rows,
124             mxUINT64_CLASS,mxREAL);
125         size_t * data = (size_t *)mxGetData( plhs[0] );
126         data[0] = (size_t)out;
127         data[1] = 1; //double
128     }
129     else
130     {
131         //Clean up:
132         __OpenCLManager__->DeleteMemory( vec->
133             GetMemoryControl() );
134         vec->SetMemoryControl(out->GetMemoryControl());
135         mxFree(out);
136     }
137     else if ( nlhs == 1 ) //send out timing:
138     {
139         //return handle to gpu vector:
140         const mwSize rows = 1;
141         plhs[0] = mxCreateNumericArray(1,&rows ,mxDOUBLE_CLASS
142             ,mxREAL);
143         double * data = (double *)mxGetData( plhs[0] );
144         float time = __OpenCLManager__->GetExecutionTime(&event)
145             ;
146         data[0] = time;
147     }
148     else if ( type == 1 && typeRHS == 0 ) //double sparse and
149         float vector given. Return should be double.
150     {
151         SparseMatrix<double> * sparse = (SparseMatrix<double> *)((
152             void *)tempSparsePtr[0]);
153         Matrix<float> * vec = (Matrix<float> *)((void *)tempVecPtr
154             [0]);
155         Matrix<double> * out;
156         if ( nrhs≥ 4)
157         {
158             size_t * tempOutPtr = (size_t *)mxGetData( prhs[3] );
159             out = (Matrix<double> *)((void *)tempOutPtr[0]);
160         }
161         else
162         {
163             out = (Matrix<double> * )mxMalloc( sizeof( Matrix<double
164                 >));

```

```

161     out->InitMatrix(_OpenCLManager_, vec->GetWidth() , vec
162                     ->GetHeight() , (double *)NULL);
163     mexMakeMemoryPersistent(out);
164 }
165 __OpenCLManager__->SparseMatrixVectorDF(sparse->
166                                         GetDataBuffer() , sparse->GetColumnIndexBuffer() ,
167                                         sparse->GetRowIndexBuffer() , vec->GetDataBuffer() , out
168                                         ->GetDataBuffer() , sparse->GetHeight() , sparse->
169                                         GetWidth() , sparse->GetLength() , sparse->
170                                         GetRowVectorLength() , &event);
171 __OpenCLManager__->WaitForCPU();
172 if (nrhs<4)
173 {
174     if (nlhs == 1) //need to create a matrix to send out:
175     {
176         //return handle to gpu vector:
177         const mwSize rows = 2;
178         plhs[0] = mxCreateNumericArray(1,&rows ,
179                                       mxUINT64_CLASS,mxREAL);
180         size_t * data = (size_t *)mxGetData(plhs[0]);
181         data[0] = (size_t)out;
182         data[1] = 1; //double
183     }
184     else if (nlhs == 1) //send out timing:
185     {
186         //return handle to gpu vector:
187         const mwSize rows = 1;
188         plhs[0] = mxCreateNumericArray(1,&rows ,mxDOUBLE_CLASS
189                                       ,mxREAL);
190         double * data = (double *)mxGetData(plhs[0]);
191         float time = __OpenCLManager__->GetExecutionTime(&event)
192         ;
193         data[0] = time;
194     }
195 }
196 else if (type == 0 && typeRHS == 1) //float sparse and
197     double vector given. Return should be float.
198 {
199     SparseMatrix<float> * sparse = (SparseMatrix<float> *)((
200         void *)tempSparsePtr[0]);
201     Matrix<double> * vec = (Matrix<double> *)(( void *)
202         tempVecPtr[0]);
203
204     Matrix<float> * out;
205     if (nrhs≥ 4)
206     {

```

```

198     size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
199     out = (Matrix<float> *)((void *)tempOutPtr[0]);
200 }
201 else
202 {
203     out = (Matrix<float> *)mxMalloc(sizeof(Matrix<float>));
204     out->InitMatrix(__OpenCLManager__, vec->GetWidth(), vec
205                     ->GetHeight(), (float *)NULL);
206     mexMakeMemoryPersistent(out);
207 }
208 __OpenCLManager__->SparseMatrixVectorFD(sparse-
209                                         ->GetDataBuffer(), sparse->GetColumnIndexBuffer(),
210                                         sparse->GetRowIndexBuffer(), vec->GetDataBuffer(), out
211                                         ->GetDataBuffer(), sparse->GetHeight(), sparse->
212                                         GetWidth(), sparse->GetLength(), sparse->
213                                         GetRowVectorLength(), &event);
214 __OpenCLManager__->WaitForCPU();
215 if (nrhs<4)
216 {
217     if (nlhs == 1) //need to create a matrix to send out:
218     {
219         //return handle to gpu vector:
220         const mwSize rows = 2;
221         plhs[0] = mxCreateNumericArray(1,&rows,
222                                       mxUINT64_CLASS,mxREAL);
223         size_t * data = (size_t *)mxGetData(plhs[0]);
224         data[0] = (size_t)out;
225         data[1] = 0; //double
226     }
227     else if (nlhs == 1) //send out timing:
228     {
229         //return handle to gpu vector:
230         const mwSize rows = 1;
231         plhs[0] = mxCreateNumericArray(1,&rows,mxDLUBLE_CLASS
232                                       ,mxREAL);
233         double * data = (double *)mxGetData(plhs[0]);
234         float time = __OpenCLManager__->GetExecutionTime(&event)
235         ;
236         data[0] = time;
237     }
238 }
239 clReleaseEvent(event);
240 }
```

10.2.15 MexVectorMinusVector.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "Matrix.h"
5
6 //Not done, needs DD and FD
7
8 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
      mxArray *prhs[])
9 {
10    cl_event event;
11    if (nrhs < 3 || nrhs > 7)
12    {
13        mexPrintf("Handle or arrays not provided. Input: Handle,
                  vector1, vector2\n");
14        return;
15    }
16    //get opencl handle:
17    size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
18    void * temp = (void *)tempPtr;
19    OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
20
21    //get vector1 handle:
22    size_t * tempPtr1 = (size_t *)mxGetData(prhs[1]);
23    size_t type1 = tempPtr1[1];
24
25    //get vector2 handle:
26    size_t * tempPtr2 = (size_t *)mxGetData(prhs[2]);
27    size_t type2 = tempPtr2[1];
28
29    if (nrhs ≥ 6)
30    {
31        size_t RowsPerThread = (size_t)(*mxGetPr(prhs[4])+0.1);
32        size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[5])+0.1);
33        __OpenCLManager__->SetVectorAndVectorRowsPerThread(
            RowsPerThread);
34        __OpenCLManager__->SetVectorAndVectorThreadsPerGroup(
            ThreadsPerGroup);
35    }
36
37    if (type1 == 0 && type2 == 0) //float & float
38    {
39        Matrix<float> * vec1 = (Matrix<float> *)((void *)tempPtr1
            [0]);

```

```

40     Matrix<float> * vec2 = (Matrix<float> *)((void *)tempPtr2
41         [0]);
42     Matrix<float> * out = vec1;
43     if (nrhs > 3)
44     {
45         size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
46         out = (Matrix<float> *)((void *)tempOutPtr[0]);
47     }
48     else if (nlhs == 1) //need to create a matrix to send out:
49     {
50         //populate gpu:
51         void * tempMat = mxMalloc(sizeof(Matrix<float>));
52         out = new(tempMat) Matrix<float>(_OpenCLManager__,
53             vec1
54             ->GetWidth(), vec1->GetHeight());
55         mexMakeMemoryPersistent(tempMat);
56
57         //return handle to gpu vector:
58         const mwSize rows = 2;
59         plhs[0] = mxCreateNumericArray(1, &rows, mxUINT64_CLASS,
60             mxREAL);
61         size_t * data = (size_t *)mxGetData(plhs[0]);
62         data[0] = (size_t)tempMat;
63         data[1] = 0; //float
64     }
65     __OpenCLManager__->VectorMinusVectorFF(vec1->GetDataBuffer()
66         (), vec2->GetDataBuffer(), out->GetDataBuffer(), vec1
67         ->GetLength(), &event);
68 }
69 else if (type1 == 1 && type2 == 0) //double & float
70 {
71     Matrix<double> * vec1 = (Matrix<double> *)((void *)
72         tempPtr1[0]);
73     Matrix<float> * vec2 = (Matrix<float> *)((void *)tempPtr2
74         [0]);
75     Matrix<double> * out = vec1;
76     if (nrhs > 3)
77     {
78         size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
79         out = (Matrix<double> *)((void *)tempOutPtr[0]);
80     }
81     else if (nlhs == 1) //need to create a matrix to send out:
82     {
83         //populate gpu:
84         void * tempMat = mxMalloc(sizeof(Matrix<double>));
85         out = new(tempMat) Matrix<double>(_OpenCLManager__,
86             vec1->GetWidth(), vec1->GetHeight());
87         mexMakeMemoryPersistent(tempMat);
88
89         //return handle to gpu vector:

```

```

81     const mwSize rows = 2;
82     plhs[0] = mxCreateNumericArray(1,&rows ,mxUINT64_CLASS,
83                                   mxREAL);
84     size_t * data = (size_t *)mxGetData(plhs[0]);
85     data[0] = (size_t)tempMat;
86     data[1] = 1; //double
87 }
88 --OpenCLManager__->VectorMinusVectorDF(vec1->GetDataBuffer()
89   (), vec2->GetDataBuffer(), out->GetDataBuffer(), vec1
90   ->GetLength(), &event);
91 }
92 else if (type1 == 1 && type2 == 1) //double & double
93 {
94     Matrix<double> * vec1 = (Matrix<double> *)((void *)
95       tempPtr1[0]);
96     Matrix<double> * vec2 = (Matrix<double> *)((void *)
97       tempPtr2[0]);
98     Matrix<double> * out = vec1;
99
100    if (nrhs > 3)
101    {
102        size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
103        out = (Matrix<double> *)((void *)tempOutPtr[0]);
104    }
105    else if (nlhs == 1) //need to create a matrix to send out:
106    {
107        //populate gpu:
108        void * tempMat = mxMalloc(sizeof(Matrix<double>));
109        out = new(tempMat) Matrix<double>(__OpenCLManager__,
110          vec1->GetWidth(), vec1->GetHeight());
111        mexMakeMemoryPersistent(tempMat);
112
113        //return handle to gpu vector:
114        const mwSize rows = 2;
115        plhs[0] = mxCreateNumericArray(1,&rows ,mxUINT64_CLASS,
116                                   mxREAL);
117        size_t * data = (size_t *)mxGetData(plhs[0]);
118        data[0] = (size_t)tempMat;
119        data[1] = 1; //double
120    }
121 --OpenCLManager__->VectorMinusVectorDD(vec1->GetDataBuffer()
122   (), vec2->GetDataBuffer(), out->GetDataBuffer(), vec1
123   ->GetLength(), &event);
124 }
125 --OpenCLManager__->WaitForCPU();
126 //if an outgoing matrix is provided already:
127 if (nlhs >= 1 && nrhs >= 4)
128 {
129     float time = __OpenCLManager__->GetExecutionTime(&event);

```

```

121     const mwSize rows = 1;
122     plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,
123                                   mxREAL);
123     float * data = (float *) mxGetData( plhs[0] );
124     *data = (float)time;
125 }
126 clReleaseEvent( event );
127 }
```

10.2.16 MexVectorMinusVectorConstant.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "Matrix.h"
5
6 //Not done, needs DD and FD
7
8 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
                  mxArray *prhs[])
9 {
10    cl_event event;
11    if (nrhs < 4 || nrhs > 7)
12    {
13        mexPrintf("Handle or arrays not provided. Input: Handle,
14                  vector1, vector2\n");
15    }
16    //get opencl handle:
17    size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
18    void * temp = (void *)tempPtr;
19    OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
20
21    //get vector1 handle:
22    size_t * tempPtr1 = (size_t *)mxGetData(prhs[1]);
23    size_t type1 = tempPtr1[1];
24
25    //get vector2 handle:
26    size_t * tempPtr2 = (size_t *)mxGetData(prhs[2]);
27    size_t type2 = tempPtr2[1];
28
29    //get contant:
30    double * tempConst = (double *)mxGetData(prhs[3]);
31    double con = *tempConst;
32
33    if (nrhs >= 7)
```

```

34    {
35        size_t RowsPerThread = (size_t)(*mxGetPr(prhs[5]) + 0.1);
36        size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[6]) + 0.1);
37        __OpenCLManager__->SetVectorAndVectorRowsPerThread(
38            RowsPerThread);
39        __OpenCLManager__->SetVectorAndVectorThreadsPerGroup(
40            ThreadsPerGroup);
41    }
42
43    if (type1 == 0 && type2 == 0) //float & float
44    {
45        Matrix<float> * vec1 = (Matrix<float> *)((void *)tempPtr1
46            [0]);
47        Matrix<float> * vec2 = (Matrix<float> *)((void *)tempPtr2
48            [0]);
49        Matrix<float> * out = vec1;
50        if (nrhs > 3)
51        {
52            size_t * tempOutPtr = (size_t *)mxGetData(prhs[4]);
53            out = (Matrix<float> *)((void *)tempOutPtr[0]);
54        }
55        else if (nlhs == 1) //need to create a matrix to send out:
56        {
57            //populate gpu:
58            void * tempMat = mxMalloc(sizeof(Matrix<float>));
59            out = new(tempMat) Matrix<float>(__OpenCLManager__,
60                vec1
61                    ->GetWidth(), vec1->GetHeight());
62            mexMakeMemoryPersistent(tempMat);
63
64            //return handle to gpu vector:
65            const mwSize rows = 2;
66            plhs[0] = mxCreateNumericArray(1, &rows, mxUINT64_CLASS,
67                mxREAL);
68            size_t * data = (size_t *)mxGetData(plhs[0]);
69            data[0] = (size_t)tempMat;
70            data[1] = 0; //float
71        }
72        __OpenCLManager__->VectorMinusVectorConstantFF(vec1-
73            >GetDataBuffer(), vec2->GetDataBuffer(), out->
74            GetDataBuffer(), (float) con, vec1->GetLength(), &
75            event);
76    }
77    else if (type1 == 1 && type2 == 0) //double & float
78    {
79        Matrix<double> * vec1 = (Matrix<double> *)((void *)
80            tempPtr1[0]);
81        Matrix<float> * vec2 = (Matrix<float> *)((void *)tempPtr2
82            [0]);
83        Matrix<double> * out = vec1;

```

```

72     if (nrhs > 3)
73     {
74         size_t * tempOutPtr = (size_t *)mxGetData(prhs[4]);
75         out = (Matrix<double> *)((void *)tempOutPtr[0]);
76     }
77     else if (nlhs == 1) //need to create a matrix to send out:
78     {
79         //populate gpu:
80         void * tempMat = mxMalloc(sizeof(Matrix<double>));
81         out = new(tempMat) Matrix<double>(_OpenCLManager__,
82             vec1->GetWidth(), vec1->GetHeight());
83         mexMakeMemoryPersistent(tempMat);
84
85         //return handle to gpu vector:
86         const mwSize rows = 2;
87         plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
88             mxREAL);
89         size_t * data = (size_t *)mxGetData(plhs[0]);
90         data[0] = (size_t)tempMat;
91         data[1] = 1; //double
92     }
93     --OpenCLManager__->VectorMinusVectorConstantDF(vec1->
94         GetDataBuffer(), vec2->GetDataBuffer(), out->
95         GetDataBuffer(), con, vec1->GetLength(), &event);
96 }
97 else if (type1 == 1 && type2 == 1) //double & double
98 {
99     Matrix<double> * vec1 = (Matrix<double> *)((void *)
100         tempPtr1[0]);
101     Matrix<double> * vec2 = (Matrix<double> *)((void *)
102         tempPtr2[0]);
103     Matrix<double> * out = vec1;
104
105     if (nrhs > 3)
106     {
107         size_t * tempOutPtr = (size_t *)mxGetData(prhs[4]);
108         out = (Matrix<double> *)((void *)tempOutPtr[0]);
109     }
110     else if (nlhs == 1) //need to create a matrix to send out:
111     {
112         //populate gpu:
113         void * tempMat = mxMalloc(sizeof(Matrix<double>));
114         out = new(tempMat) Matrix<double>(_OpenCLManager__,
115             vec1->GetWidth(), vec1->GetHeight());
116         mexMakeMemoryPersistent(tempMat);
117
118         //return handle to gpu vector:
119         const mwSize rows = 2;

```

```

113     plhs[0] = mxCreateNumericArray(1,&rows,mxUINT64_CLASS,
114         mxREAL);
115     size_t * data = (size_t *)mxGetData(plhs[0]);
116     data[0] = (size_t)tempMat;
117     data[1] = 1; //double
118     --OpenCLManager__->VectorMinusVectorConstantDD(vec1->
119             GetDataBuffer(), vec2->GetDataBuffer(), out->
120             GetDataBuffer(), con, vec1->GetLength(), &event);
121 }
122 //if an outgoing matrix is provided already:
123 __OpenCLManager__->WaitForCPU();
124 if (nlhs >= 1 && nrhs >= 5)
125 {
126     float time = __OpenCLManager__->GetExecutionTime(&event);
127     const mxArray *rows = 1;
128     plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,
129         mxREAL);
130     float * data = (float *) mxGetData(plhs[0]);
131     *data = (float)time;
132 }
133 clReleaseEvent(event);
134 }
```

10.2.17 MexVectorTimesConstant.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "Matrix.h"
5
6 //Not done, needs DD and FD
7
8 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
9                 mxArray *prhs[])
10 {
11     cl_event event;
12     if (nrhs > 7 || nrhs < 3)
13     {
14         mexPrintf("Handle or arrays not provided. Input:
15                   Handle, matrix, element\n");
16         return;
17     }
18     //get opencl handle:
19     size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
```

```

18     void * temp = (void *)tempPtr;
19     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
20
21     if (nrhs >= 6)
22     {
23         size_t RowsPerThread = (size_t)(*mxGetPr(prhs[4]) + 0.1);
24         size_t ThreadsPerGroup = (size_t)(*mxGetPr(prhs[5]) + 0.1);
25         __OpenCLManager__->SetVectorConstantRowsPerThread(
26             RowsPerThread);
27         __OpenCLManager__->SetVectorConstantThreadsPerGroup(
28             ThreadsPerGroup);
29
30         //get vector handle:
31         tempPtr = (size_t *)mxGetData(prhs[1]);
32         size_t type = tempPtr[1];
33
34         double * tempPtrEl = (double *)mxGetData(prhs[2]);
35         double element = *tempPtrEl;
36         if (type == 0) //float
37         {
38             Matrix<float> * uh = (Matrix<float> *)((void *)tempPtr
39                 [0]);
40             Matrix<float> * out = uh;
41
42             if (nrhs > 3)
43             {
44                 size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]);
45                 out = (Matrix<float> *)((void *)tempOutPtr[0]);
46             }
47             else if (nlhs == 1) //need to create a matrix to send
48                 out;
49
50             //populate gpu:
51             void * tempMat = mxMalloc(sizeof(Matrix<float>));
52             Matrix<float> * tempGPUvec = new(tempMat) Matrix<
53                 float>(__OpenCLManager__, uh->GetWidth(), uh->
54                 GetHeight());
55             mexMakeMemoryPersistent(tempMat);
56             out = tempGPUvec;
57
58             //return handle to gpu vector:
59             const mwSize rows = 2;
60             plhs[0] = mxCreateNumericArray(1, &rows,
61                 mxUINT64_CLASS, mxREAL);
62             size_t * data = (size_t *)mxGetData(plhs[0]);
63             data[0] = (size_t)tempMat;
64             data[1] = 0; //float
65         }

```

```

60      __OpenCLManager__->VectorTimesConstantFF(uh->
61          GetDataBuffer() , out->GetDataBuffer() , (float)
62              element , uh->GetLength() , &event) ;
63      }
64      else if (type == 1) //double
65      {
66          Matrix<double> * uh = (Matrix<double> *)((void *)
67              tempPtr[0]) ;
68          Matrix<double> * out = uh ;
69          if (nrhs > 3)
70          {
71              size_t * tempOutPtr = (size_t *)mxGetData(prhs[3]) ;
72              out = (Matrix<double> *)((void *)tempOutPtr[0]) ;
73          }
74          else if (nlhs == 1) //need to create a matrix to send
75              out : //populate gpu:
76          {
77              void * tempMat = mxMalloc(sizeof(Matrix<double>)) ;
78              Matrix<double> * tempGPUvec = new(tempMat) Matrix<
79                  double>(__OpenCLManager__ , uh->GetWidth() , uh
80                      ->GetHeight()) ;
81              mexMakeMemoryPersistent(tempMat) ;
82              out = tempGPUvec ;
83
84              //return handle to gpu vector:
85              const mwSize rows = 2;
86              plhs[0] = mxCreateNumericArray(1,&rows ,
87                  mxUINT64_CLASS,mxREAL) ;
88              size_t * data = (size_t *)mxGetData(plhs[0]) ;
89              data[0] = (size_t)tempMat;
90              data[1] = 1; //double
91
92              __OpenCLManager__->VectorTimesConstantDD(uh->
93                  GetDataBuffer() , out->GetDataBuffer() , (double)
94                      element , uh->GetLength() , &event) ;
95      }
96
97      //if an outgoing matrix is provided already:
98      __OpenCLManager__->WaitForCPU() ;
99      if (nlhs >= 1 && nrhs >= 4)
100      {
101          float time = __OpenCLManager__->GetExecutionTime(&event) ;
102          const mwSize rows = 1;
103          plhs[0] = mxCreateNumericArray(1,&rows ,mxSINGLE_CLASS,
104              mxREAL) ;
105          float * data = (float *) mxGetData(plhs[0]) ;
106          *data = (float)time ;

```

```
99    }
100   clReleaseEvent( event );
101
102 }
```

10.2.18 MexSum.cpp

```
1 #include "mex.h"
2 #include "OpenCLHigh.h"
3
4 /**
5 //Norm Inf:
6 //Arguments:
7 //0: OpenCL handle
8 //1: Matrix/vector to operate on.
9 /**
10 void mexFunction( int nlhs , mxArray *plhs[ ] , int nrhs , const
11                  mxArray *prhs[ ] )
12 {
13     if ( nrhs != 2 && nrhs != 3 )
14     {
15         mexPrintf( "MexSum: Handle or array not provided\n" );
16         return ;
17     }
18     //timing:
19     float time;
20
21     //get opencl handle:
22     size_t * tempPtr = (size_t *)mxGetData( prhs[0] );
23     void * temp = (void *)*tempPtr;
24     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
25
26     //get vector handle:
27     tempPtr = (size_t *)mxGetData( prhs[1] );
28     size_t type = tempPtr[1];
29
30     //get optional size handle:
31     unsigned int size = 256;
32     if ( nrhs ≥ 3 )
33     {
34         double * tempSizePtr = (double *)mxGetData( prhs[2] );
35         size = (unsigned int)(*tempSizePtr + 0.1);
36     }
37     if ( type == 0 ) //float
38     {
```

```

38     MathVector<float> * mat = (MathVector<float> *)(( void *)  
39         tempPtr[0]);  
40     float tempsum = sum(__OpenCLManager__, *mat, size, &time);  
41  
42     //return sum:  
43     const mwSize rows = 1;  
44     plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,  
45         mxREAL);  
46     float * data = (float *) mxGetData(plhs[0]);  
47     *data = (float)tempsum;  
48 }  
49 else if (type == 1) //double  
50 {  
51     MathVector<double> * mat = (MathVector<double> *)(( void *)  
52         tempPtr[0]);  
53     double tempsum = sum(__OpenCLManager__, *mat, size, &time)  
54         ;  
55  
56     //return sum:  
57     const mwSize rows = 1;  
58     plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,  
59         mxREAL);  
60     double * data = (double *) mxGetData(plhs[0]);  
61     *data = (double)tempsum;  
62 }  
63 __OpenCLManager__->WaitForCPU();  
64 if (nlhs > 1)  
65 {  
66     const mwSize rows = 1;  
67     plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,  
68         mxREAL);  
69     float * data = (float *) mxGetData(plhs[0]);  
70     *data = (float)time;  
71 }

```

10.2.19 MexNorm2.cpp

```

1 #include "mex.h"  
2 #include "OpenCLHigh.h"  
3  
4 //  
5 //Norm Inf:  
6 //Arguments:

```

```

7 //0: OpenCL handle
8 //1: Matrix/vector to operate on.
9 //
10 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
11   mxArray *prhs[])
12 {
13   float time;
14   if (nrhs != 2 && nrhs != 3)
15   {
16     mexPrintf("MexNorm2: Handle or array not provided\n");
17     return;
18   }
19   //get opencl handle:
20   size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
21   void * temp = (void *)*tempPtr;
22   OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
23
24   //get vector handle:
25   tempPtr = (size_t *)mxGetData(prhs[1]);
26   size_t type = tempPtr[1];
27
28   unsigned int size = 256;
29   if (nrhs >= 3)
30   {
31     double * tempSizePtr = (double *)mxGetData(prhs[2]);
32     size = (unsigned int)(*tempSizePtr + 0.1);
33   }
34   if (type == 0) //float
35   {
36     MathVector<float> * mat = (MathVector<float> *)((void *)
37       tempPtr[0]);
38     float tempsum = Norm2(__OpenCLManager__, *mat, size, &time
39 );
40
41   //return sum:
42   const mwSize rows = 1;
43   plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,
44     mxREAL);
45   float * data = (float *) mxGetData( plhs[0] );
46   *data = (float)tempsum;
47
48   else if (type == 1) //double
49   {
50     MathVector<double> * mat = (MathVector<double> *)((void *)
51       tempPtr[0]);
52     double tempsum = Norm2(__OpenCLManager__, *mat, size, &
53       time);
54
55   //return sum:

```

```

50     const mwSize rows = 1;
51     plhs[0] = mxCreateNumericArray(1,&rows,mxDYNAMIC_CLASS,
52                                     mxREAL);
53     double * data = (double *) mxGetData(plhs[0]);
54     *data = (double)tempsum;
55 }
56 __OpenCLManager__->WaitForCPU();
57 if (nlhs > 1)
58 {
59     const mwSize rows = 1;
60     plhs[0] = mxCreateNumericArray(1,&rows,mxDYNAMIC_CLASS,
61                                     mxREAL);
62     float * data = (float *) mxGetData(plhs[0]);
63     *data = (float)time;
64 }
65 }
```

10.2.20 MexNormInf.cpp

```

1 #include "mex.h"
2 #include "OpenCLHigh.h"
3
4 /**
5 //Norm Inf:
6 //Arguments:
7 //0: OpenCL handle
8 //1: Matrix/vector to operate on.
9 /**
10 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
11                  mxArray *prhs[])
12 {
13     float time;
14     if (nrhs != 2 && nrhs != 3)
15     {
16         mexPrintf("MexNormInf: Handle or array not provided\n");
17         return;
18     }
19     //get opencl handle:
20     size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
21     void * temp = (void *)tempPtr;
22     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
23
24     //get vector handle:
25     tempPtr = (size_t *)mxGetData(prhs[1]);
```

```

25     size_t type = tempPtr[1];
26
27     //Get optional size handle:
28     unsigned int size = 256;
29     if (nrhs >= 3)
30     {
31         double * tempSizePtr = (double *)mxGetData(prhs[2]);
32         size = (unsigned int)(*tempSizePtr + 0.1);
33     }
34     if (type == 0) //float
35     {
36         MathVector<float> * mat = (MathVector<float> *)((void *)tempPtr[0]);
37         float tempsum = NormInf(__OpenCLManager__, *mat, size, &time);
38
39         //return sum:
40         const mwSize rows = 1;
41         plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,
42                                       mxREAL);
43         float * data = (float *) mxGetData(plhs[0]);
44         *data = (float)tempsum;
45     }
46     else if (type == 1) //double
47     {
48         MathVector<double> * mat = (MathVector<double> *)((void *)tempPtr[0]);
49         double tempsum = NormInf(__OpenCLManager__, *mat, size, &time);
50
51         //return sum:
52         const mwSize rows = 1;
53         plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS,
54                                       mxREAL);
55         double * data = (double *) mxGetData(plhs[0]);
56         *data = (double)tempsum;
57     }
58     __OpenCLManager__->WaitForCPU();
59     if (nlhs > 1)
60     {
61         const mwSize rows = 1;
62         plhs[0] = mxCreateNumericArray(1,&rows,mxSINGLE_CLASS,
63                                       mxREAL);
64         float * data = (float *) mxGetData(plhs[0]);
65         *data = (float)time;
66     }
67 }
```

10.2.21 MexCoarseToFine.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "Matrix.h"
5
6 /**
7 //Function for turning a fine grid into a coarser grid.
8 //Arguments:
9 //0: OpenCL handle
10 //1: Fine matrix handle
11 //2: type of returned matrix (0: float , 1: double)
12 //output:
13 //0: Coarse matrix handle
14 /**
15 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
16                  mxArray *prhs[])
17 {
18     cl_event event;
19     if (nrhs < 3)
20     {
21         mexPrintf("Handle or arrays not provided. Input: Handle,
22                   Approximation, rhs\n");
23         return;
24     }
25     //get opencl handle:
26     size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
27     void * temp = (void *)tempPtr;
28     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
29
30     __OpenCLManager__->SetCTFRowsPerThread(64);
31     __OpenCLManager__->SetCTFThreadsPerGroup(64);
32
33     //get matrix handle:
34     tempPtr = (size_t *)mxGetData(prhs[1]);
35     size_t type = tempPtr[1];
36
37     //Determine return type:
38     size_t * tempFinePtr = (size_t *)mxGetData(prhs[2]);
39     size_t typeFineType = tempFinePtr[1];
40
41     if (type == 0 && typeFineType == 0) //float and return float
42     {
43         Matrix<float> * coarse = (Matrix<float> *)((void *)tempPtr
44             [0]);

```

```

42     Matrix<float> * fine = (Matrix<float> *)(( void *)tempFinePtr[0]);
43     __OpenCLManager__->CoarseToFineFF(fine->GetDataBuffer(), coarse->GetDataBuffer(), fine->GetWidth(), fine->GetHeight(), &event);
44 }
45 else if (type == 1 && typeFineType == 1) //double
46 {
47     Matrix<double> * coarse = (Matrix<double> *)(( void *)tempPtr[0]);
48     Matrix<double> * fine = (Matrix<double> *)(( void *)tempFinePtr[0]);
49     __OpenCLManager__->CoarseToFineDD(fine->GetDataBuffer(), coarse->GetDataBuffer(), fine->GetWidth(), fine->GetHeight(), &event);
50 }
51 else if (type == 0 && typeFineType == 1) //float to double
52 {
53     Matrix<float> * coarse = (Matrix<float> *)(( void *)tempPtr[0]);
54     Matrix<double> * fine = (Matrix<double> *)(( void *)tempFinePtr[0]);
55     __OpenCLManager__->CoarseToFineFD(fine->GetDataBuffer(), coarse->GetDataBuffer(), fine->GetWidth(), fine->GetHeight(), &event);
56 }
57
58 if (nlhs == 1) //send out timing:
59 {
60     //return handle to gpu vector:
61     const mwSize rows = 1;
62     plhs[0] = mxCreateNumericArray(1,&rows,mxDOUBLE_CLASS, mxREAL);
63     double * data = (double *)mxGetData(plhs[0]);
64     __OpenCLManager__->WaitForCPU();
65     float time = __OpenCLManager__->GetExecutionTime(&event);
66     data[0] = time;
67 }
68
69 clReleaseEvent(event);
70 }

```

10.2.22 MexFineToCoarse.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"

```

```

3 #include "MathVector.h"
4 #include "Matrix.h"
5
6 /**
7 //Function for turning a fine grid into a coarser grid.
8 //Arguments:
9 //0: OpenCL handle
10 //1: Fine matrix handle
11 //2: type of returned matrix (0: float , 1: double)
12 //output:
13 //0: Coarse matrix handle
14 //
15 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
    mxArray *prhs[])
16 {
17     cl_event event;
18     if (nrhs < 3)
19     {
20         mexPrintf("Handle or arrays not provided. Input: Handle,
            Approximation , rhs\n");
21         return ;
22     }
23     //get opencl handle:
24     size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
25     void * temp = (void *)*tempPtr;
26     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
27
28     __OpenCLManager__->SetFTCRowsPerThread(64);
29     __OpenCLManager__->SetFTCThreadsPerGroup(64);
30
31     //get matrix handle:
32     tempPtr = (size_t *)mxGetData(prhs[1]);
33     size_t type = tempPtr[1];
34
35     //get matrix handle:
36     size_t * tempCorPtr = (size_t *)mxGetData(prhs[2]);
37     size_t typeCorType = tempCorPtr[1];
38
39     if (type == 0 && typeCorType == 0) //float and return float .
40     {
41         Matrix<float> * fine = (Matrix<float> *)((void *)tempPtr
42             [0]);
43         Matrix<float> * coarse = (Matrix<float> *)((void *)
44             tempCorPtr[0]);
45         __OpenCLManager__->FineToCoarseFF(fine->GetDataBuffer(),
46             coarse->GetDataBuffer(), coarse->GetWidth(),
47             coarse->GetHeight(), &event);
48     }
49     else if (type == 1 && typeCorType == 1) //double

```

```

46    {
47        Matrix<double> * fine = (Matrix<double> *)((void *)tempPtr
48            [0]);
49        Matrix<double> * coarse = (Matrix<double> *)((void *)tempCorPtr[0]);
50        __OpenCLManager__->FineToCoarseDD(fine->GetDataBuffer(),
51            coarse->GetDataBuffer(), coarse->GetWidth(), coarse->
52            GetHeight(), &event);
53    }
54    else if (type == 1 && typeCorType == 0) //double to float
55    {
56        Matrix<double> * fine = (Matrix<double> *)((void *)tempPtr
57            [0]);
58        Matrix<float> * coarse = (Matrix<float> *)((void *)tempCorPtr[0]);
59        __OpenCLManager__->FineToCoarseDF(fine->GetDataBuffer(),
60            coarse->GetDataBuffer(), coarse->GetWidth(), coarse->
61            GetHeight(), &event);
62    }
63    if (nlhs == 1) //send out timing:
64    {
65        //return handle to gpu vector:
66        const mwSize rows = 1;
67        plhs[0] = mxCreateNumericArray(1,&rows,mxDDOUBLE_CLASS,
68            mxREAL);
69        double * data = (double *)mxGetData(plhs[0]);
70        __OpenCLManager__->WaitForCPU();
71        float time = __OpenCLManager__->GetExecutionTime(&event);
72        data[0] = time;
73    }
74    clReleaseEvent(event);
75 }
```

10.2.23 MexRBGS.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "Matrix.h"
5
6 /**
7 //Function for using jacobi and gauss-seidel smoothers.
8 //Arguments:
9 //0: OpenCL handle.
```

```

10 // 1: uh
11 // 2: RHS
12 // 3: h (optional).
13 //
14 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
                  mxArray *prhs[])
15 {
16     cl_event event;
17     double h = -1.0;
18     if (nrhs < 3 || nrhs > 8)
19     {
20         mexPrintf("Handle or arrays not provided. Input: Handle,
                  Approximation, rhs\n");
21         return;
22     }
23     //get opencl handle:
24     size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
25     void * temp = (void *)*tempPtr;
26     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
27
28     //get vector handle:
29     tempPtr = (size_t *)mxGetData(prhs[1]);
30     size_t type = tempPtr[1];
31
32     size_t * tempPtrRHS = (size_t *)mxGetData(prhs[2]);
33     size_t typeRHS = tempPtrRHS[1];
34
35     __OpenCLManager__->SetJacobiRowsPerThread(8);
36     __OpenCLManager__->SetJacobiThreadsPerGroup(8);
37
38     //Get h if given:
39     if (nrhs ≥ 4)
40     {
41         double * tempHPtr = (double *)mxGetData(prhs[3]);
42         h = *tempHPtr;
43     }
44
45     //if nmax is given:
46     unsigned int nmax = 1;
47     if (nrhs ≥ 5)
48     {
49         double * tempNmaxPtr = (double *)mxGetData(prhs[4]);
50         nmax = (unsigned int)(*tempNmaxPtr + 0.1);
51     }
52
53     unsigned int grid = 8;
54     if (nrhs ≥ 7)
55     {
56         double * tempRPTPtr = (double *)mxGetData(prhs[6]);

```

```

57     size_t RPT = (size_t)(*tempRPTPtr + 0.1);
58     double * tempTPGPtr = (double*)mxGetData(prhs[7]);
59     size_t TPG = (size_t)(*tempTPGPtr + 0.1);
60     __OpenCLManager__->SetJacobiThreadsPerGroup(TPG);
61     __OpenCLManager__->SetJacobiRowsPerThread(RPT);
62 }
63
64 if (type == 0 && typeRHS == 0) //float
65 {
66     Matrix<float> * uh = (Matrix<float> *)((void *)tempPtr[0])
67         ;
68     Matrix<float> * rhs = (Matrix<float> *)((void *)tempPtrRHS
69         [0]);
70     Matrix<float> * output;
71     if (nrhs >= 6) //swap matrix given.
72     {
73         size_t * tempOutPtr = (size_t *)mxGetData(prhs[5]);
74         output = (Matrix<float> *)((void *)tempOutPtr[0]);
75     }
76     else
77     {
78         output = (Matrix<float> *)mxMalloc(sizeof(Matrix<
79             float>));
79         output->InitMatrix(__OpenCLManager__, uh->GetWidth(), uh
80             ->GetHeight(), (float *)NULL);
81         //mexMakeMemoryPersistent(output);
82     }
83     if (h < 0) //h not given:
84     {
85         h = 1.0 / (uh->GetWidth() - 1);
86     }
87     Matrix<float> * pOut = output, * pIn = uh, * pTemp;
88     for (unsigned int i = 0; i < nmax; i++)
89     {
90         __OpenCLManager__->JacobiF(pOut->GetDataBuffer(), pIn->
91             GetDataBuffer(), rhs->GetDataBuffer(), uh->GetWidth
92             (), uh->GetHeight(), (float)h, grid, &event);
93         pTemp = pOut;
94         pOut = pIn;
95         pIn = pTemp;
96     }
97     //Make sure we keep the right one:
98     __MemoryControl__<float> * memOut = pIn->GetMemoryControl
99         () , * memIn = pOut->GetMemoryControl();
100    uh->SetMemoryControl(memOut);
101    output->SetMemoryControl(memIn);
102    //delete output;
103 }
104 else if (type == 1 && typeRHS == 1) //double

```

```

99      {
100         Matrix<double> * uh = (Matrix<double> *)((void *)tempPtr
101             [0]);
102         Matrix<double> * rhs = (Matrix<double> *)((void *)tempPtrRHS[0]);
103         Matrix<double> * output;
104         if (nrhs >= 6) //swap matrix given.
105         {
106             size_t * tempOutPtr = (size_t *)mxGetData(prhs[5]);
107             output = (Matrix<double> *)((void *)tempOutPtr[0]);
108         }
109         else
110         {
111             output = (Matrix<double> *)mxMalloc(sizeof(Matrix<
112                 double>));
113             output->InitMatrix(_OpenCLManager_, uh->GetWidth(), uh
114                 ->GetHeight(), (double *)NULL);
115             //mexMakeMemoryPersistent(output);
116         }
117         if (h < 0) //h not given:
118         {
119             h = 1.0 / (uh->GetWidth() - 1);
120         }
121         Matrix<double> * pOut = output, * pIn = uh, * pTemp;
122         for (unsigned int i = 0; i < nmax; i++)
123         {
124             _OpenCLManager_->JacobiD(pOut->GetDataBuffer(), pIn->
125                 GetDataBuffer(), rhs->GetDataBuffer(), uh->GetWidth()
126                 (), uh->GetHeight(), (double)h, grid, &event);
127             pTemp = pOut;
128             pOut = pIn;
129             pIn = pTemp;
130         }
131         //Make sure we keep the right one:
132         __MemoryControl__<double> * memOut = pIn->GetMemoryControl()
133             (), * memIn = pOut->GetMemoryControl();
134         uh->SetMemoryControl(memOut);
135         output->SetMemoryControl(memIn);
136         // delete output;
137     }
138     if (nlhs == 1) //send out timing:
139     {
140         //return handle to gpu vector:
141         const mwSize rows = 1;
142         plhs[0] = mxCreateNumericArray(1, &rows, mxDOUBLE_CLASS,
143             mxREAL);
144         double * data = (double *)mxGetData(plhs[0]);
145         _OpenCLManager_->WaitForCPU();
146         float time = _OpenCLManager_->GetExecutionTime(&event);

```

```

140     data[0] = time;
141 }
142 clReleaseEvent(event);
143 }
```

10.2.24 MexJacobi.cpp

```

1 #include "mex.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include "Matrix.h"
5
6 /**
7 //Function for using jacobi and gauss-seidel smoothers.
8 //Arguments:
9 //0: OpenCL handle.
10 //1: uh
11 //2: RHS
12 //3: h (optional).
13 /**
14 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const
15                  mxArray *prhs[])
16 {
17     cl_event event1;
18     cl_event event2;
19     double h = -1.0;
20     if (nrhs < 3 || nrhs > 7)
21     {
22         mexPrintf("Handle or arrays not provided. Input: Handle,
23                   Approximation, rhs\n");
24         return;
25     }
26     //get opencl handle:
27     size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
28     void * temp = (void *)tempPtr;
29     OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
30
31     //get vector handle:
32     tempPtr = (size_t *)mxGetData(prhs[1]);
33     size_t type = tempPtr[1];
34
35     size_t * tempPtrRHS = (size_t *)mxGetData(prhs[2]);
36     size_t typeRHS = tempPtrRHS[1];
37
38     //Get h if given:
39     if (nrhs >= 4)
```

```

38  {
39      double * tempHPtr = (double *)mxGetData(prhs[3]);
40      h = *tempHPtr;
41  }
42
43 //if nmax is given:
44 unsigned int nmax = 1;
45 if (nrhs >= 5)
46 {
47     double * tempNmaxPtr = (double *)mxGetData(prhs[4]);
48     nmax = (unsigned int)(*tempNmaxPtr + 0.1);
49 }
50 unsigned int grid = 8;
51 if (nrhs >= 7)
52 {
53     double * tempRPTPtr = (double *)mxGetData(prhs[5]);
54     size_t RPT = (size_t)(*tempRPTPtr + 0.1);
55     double * tempTPGPtr = (double *)mxGetData(prhs[6]);
56     size_t TPG = (size_t)(*tempTPGPtr + 0.1);
57     __OpenCLManager__->SetRBGSThreadsPerGroup(TPG);
58     __OpenCLManager__->SetRBGSRowsPerThread(RPT);
59 }
60
61 if (type == 0 && typeRHS == 0) //float
62 {
63     Matrix<float> * uh = (Matrix<float> *)((void *)tempPtr[0])
64         ;
65     Matrix<float> * rhs = (Matrix<float> *)((void *)tempPtrRHS
66         [0]);
67     if (h < 0) //h not given:
68     {
69         h = 1.0 / (uh->GetWidth() - 1);
70     }
71     for (unsigned int i = 0; i < nmax; i++)
72     {
73         __OpenCLManager__->JacobiMethodOddF(uh->GetDataBuffer(),
74             rhs->GetDataBuffer(), uh->GetWidth(), uh->GetHeight
75             (), (float)h, grid, &event1);
76         __OpenCLManager__->WaitForCPU();
77         __OpenCLManager__->JacobiMethodEvenF(uh->GetDataBuffer()
78             , rhs->GetDataBuffer(), rhs->GetWidth(), rhs->
79                 GetHeight(), (float)h, grid, &event2);
80     }
81 }
82 else if (type == 1 && typeRHS == 1) //double
83 {
84     Matrix<double> * uh = (Matrix<double> *)((void *)tempPtr
85         [0]);

```

```

79     Matrix<double> * rhs = (Matrix<double> *)((void *)  
80         tempPtrRHS[0]);  
81     if (h < 0) //h not given:  
82     {  
83         h = 1.0/(uh->GetWidth()-1);  
84     }  
85     for (unsigned int i = 0; i < nmax; i++)  
86     {  
87         __OpenCLManager__->JacobiMethodOddD(uh->GetDataBuffer(),  
88             rhs->GetDataBuffer(), uh->GetWidth(), uh->GetHeight  
             (), (double)h, grid, &event1);  
89         __OpenCLManager__->WaitForCPU();  
90         __OpenCLManager__->JacobiMethodEvenD(uh->GetDataBuffer()  
91             , rhs->GetDataBuffer(), uh->GetWidth(), uh->  
92                 GetHeight(), (double)h, grid, &event2);  
93     }  
94     if (nlhs == 1) //send out timing:  
95     {  
96         //return handle to gpu vector:  
97         const mxArray *rows = mxCreateNumericArray(1, &rows, mxDOUBLE_CLASS,  
98             mxREAL);  
99         double * data = (double *)mxGetData(rows);  
100        __OpenCLManager__->WaitForCPU();  
101        float time = __OpenCLManager__->GetExecutionTime(&event1)  
102            + __OpenCLManager__->GetExecutionTime(&event2);  
103        data[0] = time;  
104    }  
105    clReleaseEvent(event1);  
106    clReleaseEvent(event2);  
107 }
```

10.2.25 MexPoissonDefect.cpp

```

1 #include "mex.h"  
2 #include "OpenCLManager.h"  
3 #include "MathVector.h"  
4 #include "Matrix.h"  
5  
6 void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const  
    mxArray *prhs[])
7 {
8     cl_event event;
9     double h = -1.0;
10    if (nrhs < 3 || nrhs > 8)
```

```

11  {
12      mexPrintf("Handle or arrays not provided. Input: Handle ,
13          Approximation , rhs\n");
14  }
15 //get opencl handle:
16 size_t * tempPtr = (size_t *)mxGetData(prhs[0]);
17 void * temp = (void *)*tempPtr;
18 OpenCLManager * __OpenCLManager__ = (OpenCLManager *)temp;
19
20 //get uh handle:
21 tempPtr = (size_t *)mxGetData(prhs[1]);
22 size_t type = tempPtr[1];
23
24 //Get RHS handle:
25 size_t * tempPtrRHS = (size_t *)mxGetData(prhs[2]);
26 size_t typeRHS = tempPtrRHS[1];
27
28     //Get h if given:
29 if (nrhs >= 4)
30 {
31     double * tempHPtr = (double *)mxGetData(prhs[3]);
32     h = *tempHPtr;
33 }
34 if (nrhs >= 7)
35 {
36     double * tempRPTPtr = (double *)mxGetData(prhs[5]);
37     size_t RPT = (size_t)(*tempRPTPtr + 0.1);
38     double * tempTPGPtr = (double *)mxGetData(prhs[6]);
39     size_t TPG = (size_t)(*tempTPGPtr + 0.1);
40     __OpenCLManager__->SetDefectThreadsPerGroup(TPG);
41     __OpenCLManager__->SetDefectRowsPerThread(RPT);
42 }
43
44 if (type == 0 && typeRHS == 0) //float
45 {
46     Matrix<float> * uh = (Matrix<float> *)((void *)tempPtr[0])
47         ;
48     Matrix<float> * rhs = (Matrix<float> *)((void *)tempPtrRHS
49         [0]);
50     Matrix<float> * defect;
51     if (nlhs == 1 && nrhs < 5) //need to create a matrix to
52         send out:
53     {
54         //populate gpu:
55         defect = (Matrix<float> *)mxMalloc(sizeof(Matrix<float
56             >));
57         defect->InitMatrix(__OpenCLManager__, uh->GetWidth() , uh
58             ->GetHeight() , (float *)NULL);

```

```

54     mexMakeMemoryPersistent(defect);
55
56     //return handle to gpu vector:
57     const mwSize rows = 2;
58     plhs[0] = mxCreateNumericArray(1,&rows ,mxUINT64_CLASS,
59                                   mxREAL);
60     size_t * data = (size_t *)mxGetData(plhs[0]);
61     data[0] = (size_t)defect;
62     data[1] = 0; //float
63 }
64 else //Matrix provided:
65 {
66     size_t * tempPtrDef = (size_t *)mxGetData(prhs[4]);
67     defect = (Matrix<float> *)((void *)tempPtrDef[0]);
68 }
69 if (h < 0) //h not given:
70 {
71     h = 1.0/(uh->GetWidth()-1);
72 }
73 __OpenCLManager__->JacobiDefectF(uh->GetDataBuffer(), rhs
74                                     ->GetDataBuffer(), defect->GetDataBuffer(), uh->
75                                     GetWidth(), uh->GetHeight(), (float)h, &event);
76 }
77 else if (type == 1 && typeRHS == 1) //double
78 {
79     Matrix<double> * uh = (Matrix<double> *)((void *)tempPtr
80 [0]);
81     Matrix<double> * rhs = (Matrix<double> *)((void *)tempPtrRHS[0]);
82     Matrix<double> * defect;
83     if (nlhs == 1 && nrhs < 5) //need to create a matrix to
84         send out:
85     {
86         //populate gpu:
87         defect = (Matrix<double> *)mxMalloc(sizeof(Matrix<
88                                         double>));
89         defect->InitMatrix(__OpenCLManager__, uh->GetWidth(), uh
90                           ->GetHeight(), (double *)NULL);
91         mexMakeMemoryPersistent(defect);
92     }
93 else //Matrix provided:

```

```

94      {
95          size_t * tempPtrDef = (size_t *)mxGetData(prhs[4]);
96          defect = (Matrix<double> *)((void *)tempPtrDef[0]);
97      }
98      if (h < 0) //h not given:
99      {
100         h = 1.0/(uh->GetWidth()-1);
101     }
102     __OpenCLManager__->JacobiDefectD(uh->GetDataBuffer(), rhs
103                                         ->GetDataBuffer(), defect->GetDataBuffer(), uh->
104                                         GetWidth(), uh->GetHeight(), (double)h, &event);
105 }
106 if (nlhs == 1 && nrhs > 5) //send out timing:
107 {
108     //return handle to gpu vector:
109     const mwSize rows = 1;
110     plhs[0] = mxCreateNumericArray(1,&rows,mxDDOUBLE_CLASS,
111                                   mxREAL);
112     double * data = (double *)mxGetData(plhs[0]);
113     __OpenCLManager__->WaitForCPU();
114     float time = __OpenCLManager__->GetExecutionTime(&event);
115     data[0] = time;
116 }
117 clReleaseEvent(event);
118 }
```

10.3 API code

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

10.3.1 OpenCLManager.h

```

1 #include "MemoryControl.h"
2 #include "preprocessor.h"
3 #include <CL/cl.h>
4
5 #ifndef __OPENCLMANAGER__
6 #define __OPENCLMANAGER__
7 class OpenCLManager
8 {
9 public:
10    OpenCLManager();
11    ~OpenCLManager();
12    void ReleaseOpenCL();
13    void PrintGPUs();
```

```

14 void SetActiveGPU(unsigned int index);
15 void AddSource( char * name );
16 void AllowDouble();
17
18 //Memory management:
19 __MemoryControl__<float> * AllocateMemory( float * real ,
20   unsigned int size);
21 __MemoryControl__<double> * AllocateMemory( double * real ,
22   unsigned int size);
23 __IndexControl__ * AllocateIndex( unsigned int * index ,
24   unsigned int size);
25
26 //vector standard operations:
27 void VectorTimesConstantD(cl_kernel kernel , cl_mem & vector ,
28   cl_mem & output , double constant , unsigned int
29   vectorSize , cl_event * event);
30 void VectorTimesConstantFF(cl_mem & vector , cl_mem & output ,
31   float constant , unsigned int vectorSize , cl_event *
32   event);
33 void VectorTimesConstantFD(cl_mem & vector , cl_mem & output ,
34   double constant , unsigned int vectorSize , cl_event *
35   event);
36 void VectorTimesConstantDD(cl_mem & vector , cl_mem & output ,
37   double constant , unsigned int vectorSize , cl_event *
38   event);
39
40 void VectorOperatorVector(cl_kernel kernel , cl_mem & vector1
41   , cl_mem & vector2 , cl_mem & output , unsigned int length
42   , cl_event * event);
43 void VectorMinusVectorFF(cl_mem & vector1 , cl_mem & vector2 ,
44   cl_mem & output , unsigned int length , cl_event * event)
45 ;
46 void VectorMinusVectorFD(cl_mem & vector1 , cl_mem & vector2 ,
47   cl_mem & output , unsigned int length , cl_event * event)
48 ;
49 void VectorMinusVectorDF(cl_mem & vector1 , cl_mem & vector2 ,
50   cl_mem & output , unsigned int length , cl_event * event)
51 ;
52 void VectorMinusVectorDD(cl_mem & vector1 , cl_mem & vector2 ,
53   cl_mem & output , unsigned int length , cl_event * event)
54 ;
55 void VectorPlusVectorFF(cl_mem & vector1 , cl_mem & vector2 ,
56   cl_mem & output , unsigned int length , cl_event * event);
57 void VectorPlusVectorFD(cl_mem & vector1 , cl_mem & vector2 ,
58   cl_mem & output , unsigned int length , cl_event * event);
59 void VectorPlusVectorDF(cl_mem & vector1 , cl_mem & vector2 ,
60   cl_mem & output , unsigned int length , cl_event * event);
61 void VectorPlusVectorDD(cl_mem & vector1 , cl_mem & vector2 ,
62   cl_mem & output , unsigned int length , cl_event * event);

```

```

38
39 void VectorOperatorVectorConstant(cl_kernel kernel , cl_mem &
40     vector1 , cl_mem & vector2 , cl_mem & output , double con ,
41     unsigned int length , cl_event * event );
42 void VectorMinusVectorConstantFF(cl_mem & vector1 , cl_mem &
43     vector2 , cl_mem & output , float con , unsigned int length
44     , cl_event * event );
45 void VectorMinusVectorConstantFD(cl_mem & vector1 , cl_mem &
46     vector2 , cl_mem & output , double con , unsigned int
47     length , cl_event * event );
48 void VectorMinusVectorConstantDF(cl_mem & vector1 , cl_mem &
49     vector2 , cl_mem & output , double con , unsigned int
50     length , cl_event * event );
51 void VectorMinusVectorConstantDD(cl_mem & vector1 , cl_mem &
52     vector2 , cl_mem & output , double con , unsigned int
53     length , cl_event * event );
54
55 //sum operations:
56 void Norm(cl_kernel kernel , cl_mem & input , cl_mem & output ,
57     unsigned int threadsize , unsigned int problemsize ,
58     cl_event * event );
59 void ParallelSumReductionF(cl_mem & input , cl_mem & output ,
60     unsigned int threadsize , unsigned int problemsize ,
61     cl_event * event );
62 void ParallelSumReductionD(cl_mem & input , cl_mem & output ,
63     unsigned int threadsize , unsigned int problemsize ,
64     cl_event * event );
65 void Norm2F(cl_mem & input , cl_mem & output , unsigned int
66     threadsize , unsigned int problemsize , cl_event * event );
67 void Norm2D(cl_mem & input , cl_mem & output , unsigned int
68     threadsize , unsigned int problemsize , cl_event * event );
69 void NormInfF(cl_mem & input , cl_mem & output , unsigned int
70     threadsize , unsigned int problemsize , cl_event * event );
71 void NormInfD(cl_mem & input , cl_mem & output , unsigned int
72     threadsize , unsigned int problemsize , cl_event * event );
73
74 //Matrix vector operations:
75 void SparseMatrixVector(cl_kernel kernel , cl_mem & matData ,
76     cl_mem & matCol , cl_mem & matRow , cl_mem & vecData ,
77     cl_mem & returnData , unsigned int height , unsigned int
78     width ,
79     unsigned int numIndexes ,
80     unsigned int rowVectorLength , cl_event * event );
81 void SparseMatrixVectorDF(cl_mem & matData , cl_mem & matCol ,
82     cl_mem & matRow , cl_mem & vecData , cl_mem & returnData ,
83     unsigned int height , unsigned int width ,
84     unsigned int numIndexes , unsigned int
85     rowVectorLength , cl_event * event );

```

```

58 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);
59 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);
60 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);
61 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);
62 void BandMatrixVectorFF(cl_mem & matData, cl_mem & vecData,
   cl_mem & returnData, unsigned int height, unsigned int
   bandwidth,
   unsigned int length, cl_event *
   event);
63 void BandMatrixVectorFD(cl_mem & matData, cl_mem & vecData,
   cl_mem & returnData, unsigned int height, unsigned int
   bandwidth,
   unsigned int length, cl_event *
   event);
64 void BandMatrixVectorDF(cl_mem & matData, cl_mem & vecData,
   cl_mem & returnData, unsigned int height, unsigned int
   bandwidth,
   unsigned int length, cl_event *
   event);
65 void BandMatrixVectorDD(cl_mem & matData, cl_mem & vecData,
   cl_mem & returnData, unsigned int height, unsigned int
   bandwidth,
   unsigned int length, cl_event *
   event);
66 void FineToCoarse(cl_kernel kernel, cl_mem & fineData,
   cl_mem & corData, unsigned int corWidth, unsigned int
   corHeight, cl_event * event);
67 void FineToCoarseFF(cl_mem & fineData, cl_mem & corData,
   unsigned int corWidth, unsigned int corHeight, cl_event
   * event);
68 void FineToCoarseFD(cl_mem & fineData, cl_mem & corData,
   unsigned int corWidth, unsigned int corHeight, cl_event
   * event);
69 void FineToCoarseDF(cl_mem & fineData, cl_mem & corData,
   unsigned int corWidth, unsigned int corHeight, cl_event
   * event);
70 void FineToCoarseDD(cl_mem & fineData, cl_mem & corData,
   unsigned int corWidth, unsigned int corHeight, cl_event
   * event);
71 //Coarse-Fine operations:
72 void FineToCoarse(cl_kernel kernel, cl_mem & fineData,
   cl_mem & corData, unsigned int corWidth, unsigned int
   corHeight, cl_event * event);
73 void FineToCoarseFF(cl_mem & fineData, cl_mem & corData,
   unsigned int corWidth, unsigned int corHeight, cl_event
   * event);
74 void FineToCoarseFD(cl_mem & fineData, cl_mem & corData,
   unsigned int corWidth, unsigned int corHeight, cl_event
   * event);

```

```

75   void FineToCoarseDF(cl_mem & fineData , cl_mem & corData ,
76     unsigned int corWidth , unsigned int corHeight , cl_event
77     * event );
76   void FineToCoarseDD(cl_mem & fineData , cl_mem & corData ,
77     unsigned int corWidth , unsigned int corHeight , cl_event
78     * event );
77   void CoarseToFine(cl_kernel kernel , cl_mem & fineData ,
78     cl_mem & corData , unsigned int fineWidth , unsigned int
79     fineHeight , cl_event * event );
78   void CoarseToFineFF(cl_mem & fineData , cl_mem & corData ,
80     unsigned int fineWidth , unsigned int fineHeight ,
81     cl_event * event );
79   void CoarseToFineFD(cl_mem & fineData , cl_mem & corData ,
80     unsigned int fineWidth , unsigned int fineHeight ,
81     cl_event * event );
80   void CoarseToFineDD(cl_mem & fineData , cl_mem & corData ,
81     unsigned int fineWidth , unsigned int fineHeight ,
82     cl_event * event );
81
82 // Jacobi method:
83   void JacobiID(cl_mem & output , cl_mem & input , cl_mem &
84     rightData , unsigned int width , unsigned int height ,
85     double spacing , unsigned int grid , cl_event * event );
84   void JacobiF(cl_mem & output , cl_mem & input , cl_mem &
85     rightData , unsigned int width , unsigned int height ,
86     float spacing , unsigned int grid , cl_event * event );
85   void JacobiMethodF(cl_kernel kernel , cl_mem & leftData ,
86     cl_mem & rightData , unsigned int width , unsigned int
87     height , float spacing , unsigned int grid , cl_event *
88     event );
86   void JacobiMethodD(cl_kernel kernel , cl_mem & leftData ,
87     cl_mem & rightData , unsigned int width , unsigned int
88     height , double spacing , unsigned int grid , cl_event *
89     event );
87   void JacobiMethodOddF(cl_mem & leftData , cl_mem & rightData ,
88     unsigned int width , unsigned int height , float spacing ,
89     unsigned int grid , cl_event * event );
88   void JacobiMethodOddD(cl_mem & leftData , cl_mem & rightData ,
89     unsigned int width , unsigned int height , double spacing
90     , unsigned int grid , cl_event * event );
89   void JacobiMethodEvenF(cl_mem & leftData , cl_mem & rightData ,
90     unsigned int width , unsigned int height , float spacing
91     , unsigned int grid , cl_event * event );
90   void JacobiMethodEvenD(cl_mem & leftData , cl_mem & rightData ,
91     unsigned int width , unsigned int height , double
92     spacing , unsigned int grid , cl_event * event );
91   void JacobiDefectF(cl_mem & leftData , cl_mem & rightData ,
92     cl_mem & defect , unsigned int width , unsigned int height
93     , float spacing , cl_event * event );

```

```

92 void JacobiDefectD(cl_mem & leftData , cl_mem & rightData ,
93                     cl_mem & defect , unsigned int width , unsigned int height
94                     , double spacing , cl_event * event);
95
95 //Memory swapping:
96 void SwapGPUBufferData(const cl_mem & buffer , void * ptr ,
97                         unsigned int size , size_t sizeType);
96 void WriteGPUBufferData(const cl_mem & buffer , void * ptr ,
97                         unsigned int size , size_t sizeType);
97
98 //Memory resizing:
99 void ResizeGPUBuffer( __MemoryControl__<float> * control ,
100                      unsigned int size);
100 void ResizeGPUBuffer( __MemoryControl__<double> * control ,
100                      unsigned int size);
101 void ResizeGPUBuffer( __IndexControl__ * control , unsigned
101                      int size);
102
103 //Memory leak control:
104 void DeleteMemory( __MemoryControl__<float> * mem);
105 void DeleteMemory( __MemoryControl__<double> * mem);
106 void DeleteIndex( __IndexControl__ * mem);
107
108
109 //Autotuning:
110 void SetSparseMatrixVectorRowsPerThread(size_t );
111 void SetSparseMatrixVectorThreadsPerGroup(size_t );
112 void SetBandMatrixVectorRowsPerThread(size_t );
113 void SetBandMatrixVectorThreadsPerGroup(size_t );
114 void SetNormRowsPerThread(size_t );
115 void SetNormThreadsPerGroup(size_t );
116 void SetVectorAndVectorRowsPerThread(size_t );
117 void SetVectorAndVectorThreadsPerGroup(size_t );
118 void SetVectorConstantRowsPerThread(size_t );
119 void SetVectorConstantThreadsPerGroup(size_t );
120 void SetJacobiRowsPerThread(size_t );
121 void SetJacobiThreadsPerGroup(size_t );
122 void SetRBGSRowsPerThread(size_t );
123 void SetRBGSThreadsPerGroup(size_t );
124 void SetDefectRowsPerThread(size_t );
125 void SetDefectThreadsPerGroup(size_t );
126 void SetFTCRowsPerThread(size_t );
127 void SetFTCThreadsPerGroup(size_t );
128 void SetCTFRowsPerThread(size_t );
129 void SetCTFThreadsPerGroup(size_t );
130 void WaitForCPU();
131 float GetExecutionTime(cl_event * event);
132
133

```

```

134 private:
135     //Shortcut functions:
136     cl_kernel CreateKernel(char * name);
137     //Platform and Device control:
138     cl_platform_id * vectorPlatforms;
139     unsigned int numPlatforms;
140     cl_device_id ** vectorDevices;
141     unsigned int * numDevices;
142     cl_platform_id platform;
143     cl_device_id device;
144
145     //Program control:
146     cl_program program;
147     cl_context context;
148     char ** vectorSourceFiles;
149     unsigned int numSourceFiles;
150     cl_command_queue queue;
151     bool EnableDouble;
152     char ** program_strings;
153     size_t * program_sizes;
154
155     //Autotuning constants:
156     size_t SparseMatrixVectorRowsPerThread;
157     size_t SparseMatrixVectorThreadsPerGroup;
158     size_t BandMatrixVectorRowsPerThread;
159     size_t BandMatrixVectorThreadsPerGroup;
160     size_t NormRowsPerThread;
161     size_t NormThreadsPerGroup;
162     size_t VectorAndVectorRowsPerThread;
163     size_t VectorAndVectorThreadsPerGroup;
164     size_t VectorConstantRowsPerThread;
165     size_t VectorConstantThreadsPerGroup;
166     size_t JacobiRowsPerThread;
167     size_t JacobiThreadsPerGroup;
168     size_t RBGSRowsPerThread;
169     size_t RBGSThreadsPerGroup;
170     size_t DefectRowsPerThread;
171     size_t DefectThreadsPerGroup;
172     size_t FTCRowsPerThread;
173     size_t FTCThreadsPerGroup;
174     size_t CTFRowsPerThread;
175     size_t CTFThreadsPerGroup;
176
177
178
179     //Memory Control:
180     __MemoryControl__<float> ** vectorMemoryF;
181     __MemoryControl__<double> ** vectorMemoryD;
182     unsigned int numMemoryF;

```

```
183     unsigned int numMemoryD;
184     unsigned int capMemoryF;
185     unsigned int capMemoryD;
186     __IndexControl__ ** vectorIndex ;
187     unsigned int numIndex;
188     unsigned int capIndex;
189
190 //Kernels:
191 cl_kernel kernelReductionF ;
192 cl_kernel kernelReductionD ;
193 cl_kernel kernelSparseMatrixVectorFF ;
194 cl_kernel kernelSparseMatrixVectorDF ;
195 cl_kernel kernelSparseMatrixVectorDD ;
196 cl_kernel kernelSparseMatrixVectorFD ;
197 cl_kernel kernelBandMatrixVectorFF ;
198 cl_kernel kernelBandMatrixVectorFD ;
199 cl_kernel kernelBandMatrixVectorDF ;
200 cl_kernel kernelBandMatrixVectorDD ;
201 cl_kernel kernelJacobiMethodOddF ;
202 cl_kernel kernelJacobiMethodOddD ;
203 cl_kernel kernelJacobiMethodEvenF ;
204 cl_kernel kernelJacobiMethodEvenD ;
205 cl_kernel kernelJacobiDefectF ;
206 cl_kernel kernelJacobiDefectD ;
207 cl_kernel kernelJacobiF ;
208 cl_kernel kernelJacobiD ;
209 cl_kernel kernelRefineCTFFF ;
210 cl_kernel kernelRefineCTFFD ;
211 cl_kernel kernelRefineCTFDD ;
212 cl_kernel kernelRefineFTCFF ;
213 cl_kernel kernelRefineFTCDF ;
214 cl_kernel kernelRefineFTCDD ;
215 cl_kernel kernelVectorTimesConstantFF ;
216 cl_kernel kernelVectorTimesConstantFD ;
217 cl_kernel kernelVectorTimesConstantDD ;
218 cl_kernel kernelVectorPlusVectorFF ;
219 cl_kernel kernelVectorPlusVectorFD ;
220 cl_kernel kernelVectorPlusVectorDF ;
221 cl_kernel kernelVectorPlusVectorDD ;
222 cl_kernel kernelVectorMinusVectorFF ;
223 cl_kernel kernelVectorMinusVectorFD ;
224 cl_kernel kernelVectorMinusVectorDF ;
225 cl_kernel kernelVectorMinusVectorDD ;
226 cl_kernel kernelVectorMinusVectorConstantFF ;
227 cl_kernel kernelVectorMinusVectorConstantFD ;
228 cl_kernel kernelVectorMinusVectorConstantDF ;
229 cl_kernel kernelVectorMinusVectorConstantDD ;
230 cl_kernel kernelNormInff ;
231 cl_kernel kernelNormInfD ;
```

```

232     cl_kernel kernelNorm2F;
233     cl_kernel kernelNorm2D;
234
235     //conditional kernel statements:
236     bool NVIDIA;
237
238     //functions:
239     void ResetContext();
240     void ResetProgram();
241     void PushBack( __MemoryControl__<float> * mem);
242     void PushBack( __MemoryControl__<double> * mem);
243     void PushBack( __IndexControl__ * mem);
244
245     //error function:
246     void WriteError(cl_int err);
247 };
248
249 #endif

```

10.3.2 OpenCLManager.cpp

```

1 #include "preprocessor.h"
2 #include "OpenCLManager.h"
3 #include <iostream>
4 #include <fstream>
5 #include "mex.h"
6
7 //Definition of singleton:
8 OpenCLManager * __OpenCLManager__;
9
10 void OpenCLManager::PushBack( __MemoryControl__<float> * mem)
11 {
12     numMemoryF++;
13     if (numMemoryF == capMemoryF)
14     {
15         capMemoryF *= 2;
16         __MemoryControl__<float> ** temp = ( __MemoryControl__<
17             float> **) mxMalloc( sizeof( __MemoryControl__<float>*) *
18             capMemoryF );
19         mexMakeMemoryPersistent( temp );
20         for (unsigned int i = 0; i < numMemoryF-1; i += 1)
21         {
22             temp[ i ] = vectorMemoryF[ i ];
23         }
24         mxFree( vectorMemoryF );
25         vectorMemoryF = temp;

```

```

24     }
25     vectorMemoryF [ numMemoryF-1 ] = mem;
26 }
27 void OpenCLManager :: PushBack( __MemoryControl__<double> * mem)
28 {
29     numMemoryD++;
30     if ( numMemoryD == capMemoryD )
31     {
32         capMemoryD *= 2;
33         __MemoryControl__<double> ** temp = ( __MemoryControl__<
34             double> ** ) mxMalloc ( sizeof ( __MemoryControl__<double>* ) *
35             capMemoryD );
36         mexMakeMemoryPersistent ( temp );
37         for ( unsigned int i = 0; i < numMemoryD-1; i += 1 )
38         {
39             temp [ i ] = vectorMemoryD [ i ];
40         }
41         mxFree ( vectorMemoryD );
42         vectorMemoryD = temp;
43     }
44     vectorMemoryD [ numMemoryD-1 ] = mem;
45 }
46 void OpenCLManager :: PushBack( __IndexControl__ * mem)
47 {
48     numIndex++;
49     if ( numIndex == capIndex )
50     {
51         capIndex *= 2;
52         __IndexControl__ ** temp = ( __IndexControl__ ** ) mxMalloc (
53             sizeof ( __IndexControl__* ) * capIndex );
54         mexMakeMemoryPersistent ( temp );
55         for ( unsigned int i = 0; i < numIndex-1; i += 1 )
56         {
57             temp [ i ] = vectorIndex [ i ];
58         }
59         mxFree ( vectorIndex );
60         vectorIndex = temp;
61     }
62 // 
63 // Print Commands:
64 // 
65 void OpenCLManager :: PrintGPUs()
66 {
67     unsigned int tempCounter = 0;
68     char name[64];
69     char ext[4096];

```

```

70     size_t size = 64;
71     size_t ext_size = 4096;
72     char platformname[64];
73     for (unsigned int i = 0; i < numPlatforms; i += 1)
74     {
75         clGetPlatformInfo(vectorPlatforms[i], CL_PLATFORM_NAME,
76                           size, platformname, NULL);
77         mexPrintf("Platform %u: %s\n", i, platformname);
78         for (unsigned int j = 0; j < numDevices[i]; j += 1)
79         {
80             clGetDeviceInfo(vectorDevices[i][j], CL_DEVICE_NAME,
81                             size, name, NULL);
82             clGetDeviceInfo(vectorDevices[i][j],
83                             CL_DEVICE_EXTENSIONS, ext_size, ext, NULL);
83             mexPrintf("GPU %u: %s supports: %s\n", tempCounter, name,
84                       ext);
85             tempCounter++;
86         }
87     }
88 /**
89 //GPU buffer commands:
90 /**
91 void OpenCLManager::SwapGPUBufferData(const cl_mem & buffer,
92                                     void * ptr, unsigned int size, size_t sizeType)
93 {
94     cl_int err;
95     err = clEnqueueReadBuffer(queue, buffer, CL_TRUE, 0, size *
96                               sizeType, ptr, NULL, NULL, NULL);
97     #ifdef __DEBUG__
98     if (err < 0)
99     {
100         std::cout << "Failed to swap GPU buffer Data, code: " <<
101             err << std::endl;
102     }
103 #endif
104 }
105 void OpenCLManager::WriteGPUBufferData(const cl_mem & buffer,
106                                         void * ptr, unsigned int size, size_t sizeType)
107 {
108     cl_int err;
109     err = clEnqueueWriteBuffer(queue, buffer, CL_TRUE, 0, size *
110                               sizeType, ptr, NULL, NULL, NULL);
111     #ifdef __DEBUG__
112     if (err < 0)
113     {

```

```

110         std::cout << "Failed to swap GPU buffer Data, code: " <<
111             err << std::endl;
112     }
113 }
114
115 void OpenCLManager::ResizeGPUBuffer( __MemoryControl__<float>
116     * control, unsigned int size)
116 {
117     cl_int err;
118     clReleaseMemObject( control->buffer );
119     control->buffer = clCreateBuffer( context, CL_MEM_READ_WRITE
120         | CL_MEM_COPY_HOST_PTR, size * sizeof( float ), &( control->data[0] ), &err );
120 #ifdef __DEBUG__
121     if ( err < 0 )
122     {
123         std::cout << "Failed to resize GPU buffer, code: " <<
124             err << std::endl;
124     }
125 #endif
126     control->BuffersMatch = true;
127 }
128 void OpenCLManager::ResizeGPUBuffer( __MemoryControl__<double>
129     * control, unsigned int size)
129 {
130     cl_int err;
131     clReleaseMemObject( control->buffer );
132     control->buffer = clCreateBuffer( context, CL_MEM_READ_WRITE
133         | CL_MEM_COPY_HOST_PTR, size * sizeof( double ), &( control->data[0] ), &err );
133 #ifdef __DEBUG__
134     if ( err < 0 )
135     {
136         std::cout << "Failed to resize GPU buffer, code: " <<
137             err << std::endl;
137     }
138 #endif
139     control->BuffersMatch = true;
140 }
141 void OpenCLManager::ResizeGPUBuffer( __IndexControl__ *
142     control, unsigned int size)
142 {
143     cl_int err;
144     clReleaseMemObject( control->buffer );
145     control->buffer = clCreateBuffer( context, CL_MEM_READ_WRITE
146         | CL_MEM_COPY_HOST_PTR, size * sizeof( unsigned int ), &(control->data[0]), &err );
146 #ifdef __DEBUG__

```



```

189 __MemoryControl__<float> * OpenCLManager:: AllocateMemory(
190     float * data, unsigned int size)
191 {
192     //openCL error var:
193     cl_int err;
194
195     //declare new memorycontroller:
196     __MemoryControl__<float> * temp = (__MemoryControl__<float>
197         *)mxMalloc( sizeof( __MemoryControl__<float>) );
198     //temp = new(temp) __MemoryControl__<float>();
199     mexMakeMemoryPersistent( temp );
200     temp->size = size;
201     temp->data = ( float * )mxMalloc( sizeof( float ) * size );
202     mexMakeMemoryPersistent( temp->data );
203
204     if ( data != NULL )
205     {
206         temp->buffer = clCreateBuffer( context , CL_MEM_READ_WRITE |
207             CL_MEM_COPY_HOST_PTR, size* sizeof( float ), data , &err )
208             ;
209         #ifdef __DEBUG__
210             if ( err < 0 )
211             {
212                 std :: cout << "Error: Tried to create new memory object
213                     on GPU. Code: " << err << std :: endl;
214             }
215         #endif
216         temp->BuffersMatch = true;
217     }
218     else
219     {
220         temp->buffer = clCreateBuffer( context , CL_MEM_READ_WRITE,
221             size* sizeof( float ), NULL, &err );
222         #ifdef __DEBUG__
223             if ( err < 0 )
224             {
225                 std :: cout << "Error: Tried to create new memory object
226                     on GPU. Code: " << err << std :: endl;
227             }
228         #endif
229         temp->BuffersMatch = true;
230     }
231     temp->RefCount = 1;
232     //PushBack( temp );
233     return temp;
234 }
235 __MemoryControl__<double> * OpenCLManager:: AllocateMemory(
236     double * data, unsigned int size)

```

```

230 {
231     //openCL error var:
232     cl_int err;
233
234
235     //declare new memorycontroller:
236     __MemoryControl__<double> * temp = ( __MemoryControl__<double>
237         > *) mxMalloc( sizeof( __MemoryControl__<double> ) );
238     //temp = new( temp ) __MemoryControl__<float>();
239     mexMakeMemoryPersistent( temp );
240     temp->size = size;
241     temp->data = ( double * ) mxMalloc( sizeof( double ) * size );
242     mexMakeMemoryPersistent( temp->data );
243
244     if ( data != NULL )
245     {
246         temp->buffer = clCreateBuffer( context , CL_MEM_READ_WRITE |
247             CL_MEM_COPY_HOST_PTR, size* sizeof( double ), data , &err
248             );
249         #ifdef __DEBUG__
250             if ( err < 0 )
251             {
252                 std :: cout << "Error: Tried to create new memory object
253                     on GPU. Code: " << err << std :: endl;
254             }
255         #endif
256         temp->BuffersMatch = true;
257     }
258     else
259     {
260         temp->buffer = clCreateBuffer( context , CL_MEM_READ_WRITE,
261             size* sizeof( double ), NULL, &err );
262         #ifdef __DEBUG__
263             if ( err < 0 )
264             {
265                 std :: cout << "Error: Tried to create new memory object
266                     on GPU. Code: " << err << std :: endl;
267             }
268         #endif
269         temp->BuffersMatch = true;
270     }
271     temp->RefCount = 1;
272     //PushBack( temp );
273     return temp;
274 }
275
276 //Linear algebra components:
277 //

```

```

273
274
275 //Multiply by constant:
276 void OpenCLManager::VectorTimesConstantFF(cl_mem & vector,
277                                            cl_mem & output, float constant, unsigned int vectorSize,
278                                            cl_event * event)
277 {
278     cl_int err;
279     clSetKernelArg( kernelVectorTimesConstantFF, 0, sizeof(
280                     cl_mem ), &vector );
280     clSetKernelArg( kernelVectorTimesConstantFF, 1, sizeof(
281                     cl_mem ), &output );
281     clSetKernelArg( kernelVectorTimesConstantFF, 2, sizeof(
282                     float ), &constant );
282     clSetKernelArg( kernelVectorTimesConstantFF, 3, sizeof(
283                     unsigned int ), &vectorSize );
283
284     size_t global_size;
285     if (vectorSize % (VectorConstantThreadsPerGroup *
286                       VectorConstantRowsPerThread) == 0)
286     {
287         global_size = vectorSize/(VectorConstantRowsPerThread);
288     }
289     else
290     {
291         global_size = (vectorSize/(VectorConstantThreadsPerGroup *
292                           VectorConstantRowsPerThread)+1) *
293                           VectorConstantThreadsPerGroup;
294     }
295     size_t local_size = VectorConstantThreadsPerGroup;
296     err = clEnqueueNDRangeKernel(queue,
297                                 kernelVectorTimesConstantFF, 1, NULL, &global_size, &
298                                 local_size, 0, NULL, event);
299 #ifdef __DEBUG__
300     if (err < 0)
301     {
302         std::cout << "Failed to enqueue VectorTimesConstant, code: "
303                     << err << std::endl;
304     }
305 #endif
306 }
307
308 void OpenCLManager::VectorTimesConstantD(cl_kernel kernel,
309                                           cl_mem & vector, cl_mem & output, double constant,
310                                           unsigned int vectorSize, cl_event * event)
309 {
310     cl_int err;
311     clSetKernelArg( kernel, 0, sizeof( cl_mem ), &vector );
312     clSetKernelArg( kernel, 1, sizeof( cl_mem ), &output );
313     clSetKernelArg( kernel, 2, sizeof( double ), &constant );

```

```

308     clSetKernelArg( kernel , 3, sizeof( unsigned int ) , &
309                     vectorSize );
310
311     size_t global_size ;
312     if (vectorSize % (VectorConstantThreadsPerGroup *
313                         VectorConstantRowsPerThread) == 0)
314     {
315     }
316     else
317     {
318         global_size = (vectorSize / (VectorConstantThreadsPerGroup *
319                             VectorConstantRowsPerThread) + 1) *
320                             VectorConstantThreadsPerGroup ;
321     }
322     size_t local_size = VectorConstantThreadsPerGroup ;
323     err = clEnqueueNDRangeKernel(queue , kernel , 1, NULL, &
324                                 global_size , &local_size , 0, NULL, event );
325 #ifdef __DEBUG__
326     if (err < 0)
327     {
328         std::cout << "Failed to enqueue VectorTimesConstant , code:
329                     " << err << std::endl ;
330     }
331 #endif
332 }
333 void OpenCLManager::VectorTimesConstantDD(cl_mem & vector ,
334                                            cl_mem & output , double constant , unsigned int vectorSize ,
335                                            cl_event * event )
336 {
337     VectorTimesConstantD(kernelVectorTimesConstantDD , vector ,
338                           output , constant , vectorSize , event );
339 }
340 void OpenCLManager::VectorTimesConstantFD(cl_mem & vector ,
341                                            cl_mem & output , double constant , unsigned int vectorSize ,
342                                            cl_event * event )
343 {
344     VectorTimesConstantD(kernelVectorTimesConstantFD , vector ,
345                           output , constant , vectorSize , event );
346 }
347 // Vector Minus Vector:
348 // Vector Operator Vector:
349 //
350 void OpenCLManager::VectorOperatorVector(cl_kernel kernel ,
351                                         cl_mem & vector1 , cl_mem & vector2 , cl_mem & output ,
352                                         unsigned int length , cl_event * event )
353 {
354     cl_int err ;

```

```

343     clSetKernelArg( kernel , 0, sizeof( cl_mem ) , &vector1 );
344     clSetKernelArg( kernel , 1, sizeof( cl_mem ) , &vector2 );
345     clSetKernelArg( kernel , 2, sizeof( cl_mem ) , &output );
346     clSetKernelArg( kernel , 3, sizeof( unsigned int ) , &length );
347
348     size_t global_size;
349     if ( length % (VectorAndVectorThreadsPerGroup *
350                   VectorAndVectorRowsPerThread) == 0)
351     {
352         global_size = length/(VectorAndVectorRowsPerThread);
353     }
354     else
355     {
356         global_size = (length/(VectorAndVectorThreadsPerGroup *
357                               VectorAndVectorRowsPerThread)+1) *
358                               VectorAndVectorThreadsPerGroup;
359     }
360     size_t local_size = VectorAndVectorThreadsPerGroup;
361
362     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
363                                 global_size, &local_size, 0, NULL, event);
364 #ifdef __DEBUG__
365     if (err < 0)
366     {
367         std::cout << "Failed to enqueue VectorAndVector, code: "
368             << err << std::endl;
369     }
370 #endif
371 }
372 void OpenCLManager::VectorMinusVectorFF(cl_mem & vector1 ,
373                                         cl_mem & vector2 , cl_mem & output , unsigned int length ,
374                                         cl_event * event)
375 {
376     VectorOperatorVector(kernelVectorMinusVectorFF , vector1 ,
377                           vector2 , output , length , event );
378 }
379 void OpenCLManager::VectorMinusVectorFD(cl_mem & vector1 ,
380                                         cl_mem & vector2 , cl_mem & output , unsigned int length ,
381                                         cl_event * event)
382 {
383     VectorOperatorVector(kernelVectorMinusVectorFD , vector1 ,
384                           vector2 , output , length , event );
385 }
386 void OpenCLManager::VectorMinusVectorDF(cl_mem & vector1 ,
387                                         cl_mem & vector2 , cl_mem & output , unsigned int length ,
388                                         cl_event * event)
389 {
390     VectorOperatorVector(kernelVectorMinusVectorDF , vector1 ,
391                           vector2 , output , length , event );

```

```

378 }
379 void OpenCLManager::VectorMinusVectorDD(cl_mem & vector1 ,
   cl_mem & vector2 , cl_mem & output , unsigned int length ,
   cl_event * event)
380 {
381   VectorOperatorVector(kernelVectorMinusVectorDD , vector1 ,
   vector2 , output , length , event);
382 }
383 void OpenCLManager::VectorPlusVectorFF(cl_mem & vector1 ,
   cl_mem & vector2 , cl_mem & output , unsigned int length ,
   cl_event * event)
384 {
385   VectorOperatorVector(kernelVectorPlusVectorFF , vector1 ,
   vector2 , output , length , event);
386 }
387 void OpenCLManager::VectorPlusVectorFD(cl_mem & vector1 ,
   cl_mem & vector2 , cl_mem & output , unsigned int length ,
   cl_event * event)
388 {
389   VectorOperatorVector(kernelVectorPlusVectorFD , vector1 ,
   vector2 , output , length , event);
390 }
391 void OpenCLManager::VectorPlusVectorDF(cl_mem & vector1 ,
   cl_mem & vector2 , cl_mem & output , unsigned int length ,
   cl_event * event)
392 {
393   VectorOperatorVector(kernelVectorPlusVectorDF , vector1 ,
   vector2 , output , length , event);
394 }
395 void OpenCLManager::VectorPlusVectorDD(cl_mem & vector1 ,
   cl_mem & vector2 , cl_mem & output , unsigned int length ,
   cl_event * event)
396 {
397   VectorOperatorVector(kernelVectorPlusVectorDD , vector1 ,
   vector2 , output , length , event );
398 }
399
400
401 //Vector times vector constant
402 void OpenCLManager::VectorOperatorVectorConstant( cl_kernel
   kernel , cl_mem & vector1 , cl_mem & vector2 , cl_mem &
   output , double con , unsigned int length , cl_event * event)
403 {
404   cl_int err;
405   clSetKernelArg( kernel , 0, sizeof( cl_mem ) , &vector1 );
406   clSetKernelArg( kernel , 1, sizeof( cl_mem ) , &vector2 );
407   clSetKernelArg( kernel , 2, sizeof( cl_mem ) , &output );
408   clSetKernelArg( kernel , 3, sizeof( double ) , &con );
409   clSetKernelArg( kernel , 4, sizeof( unsigned int ) , &length );

```

```

410
411     size_t global_size;
412     if (length % (VectorAndVectorThreadsPerGroup *
413                 VectorAndVectorRowsPerThread) == 0)
414     {
415         global_size = length / (VectorAndVectorRowsPerThread);
416     }
417     else
418     {
419         global_size = (length / (VectorAndVectorThreadsPerGroup *
420                               VectorAndVectorRowsPerThread) + 1) *
421                               VectorAndVectorThreadsPerGroup;
422     }
423     size_t local_size = VectorAndVectorThreadsPerGroup;
424
425     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
426                                   global_size, &local_size, 0, NULL, event);
427 #ifdef __DEBUG__
428     if (err < 0)
429     {
430         std::cout << "Failed to enqueue VectorAndVectorConstant,
431                         code: " << err << std::endl;
432     }
433 #endif
434 }
435
436 void OpenCLManager::VectorMinusVectorConstantFF(cl_mem &
437                                                 vector1, cl_mem &vector2, cl_mem &output, float con,
438                                                 unsigned int length, cl_event * event)
439 {
440     cl_int err;
441     clSetKernelArg( kernelVectorMinusVectorConstantFF, 0, sizeof
442                     ( cl_mem ), &vector1);
443     clSetKernelArg( kernelVectorMinusVectorConstantFF, 1, sizeof
444                     ( cl_mem ), &vector2);
445     clSetKernelArg( kernelVectorMinusVectorConstantFF, 2, sizeof
446                     ( cl_mem ), &output);
447     clSetKernelArg( kernelVectorMinusVectorConstantFF, 3, sizeof
448                     ( float ), &con);
449     clSetKernelArg( kernelVectorMinusVectorConstantFF, 4, sizeof
450                     ( unsigned int ), &length);
451
452     size_t global_size;
453     if (length % (VectorAndVectorThreadsPerGroup *
454                   VectorAndVectorRowsPerThread) == 0)
455     {
456         global_size = length / (VectorAndVectorRowsPerThread);
457     }
458     else

```

```

446    {
447        global_size = (length/(VectorAndVectorThreadsPerGroup *
448                      VectorAndVectorRowsPerThread)+1) *
449                      VectorAndVectorThreadsPerGroup;
450    }
451    size_t local_size = VectorAndVectorThreadsPerGroup;
452
453    err = clEnqueueNDRangeKernel(queue,
454                                  kernelVectorMinusVectorConstantFF , 1, NULL, &global_size
455                                  , &local_size , 0, NULL, event);
456 #ifdef __DEBUG__
457     if (err < 0)
458     {
459         std::cout << "Failed to enqueue VectorAndVectorConstant ,
460                         code: " << err << std::endl;
461     }
462 #endif
463 }
464
465 void OpenCLManager::VectorMinusVectorConstantFD(cl_mem &
466                                                 vector1 , cl_mem & vector2 , cl_mem & output , double con ,
467                                                 unsigned int length , cl_event * event)
468 {
469     VectorOperatorVectorConstant(
470         kernelVectorMinusVectorConstantFD , vector1 , vector2 ,
471         output , con , length , event);
472 }
473
474 void OpenCLManager::VectorMinusVectorConstantDF(cl_mem &
475                                                 vector1 , cl_mem & vector2 , cl_mem & output , double con ,
476                                                 unsigned int length , cl_event * event)
477 {
478     VectorOperatorVectorConstant(
479         kernelVectorMinusVectorConstantDF , vector1 , vector2 ,
480         output , con , length , event);
481 }
482
483 // Norm:
484 // Norm:
485
486 void OpenCLManager::Norm(cl_kernel kernel , cl_mem & input ,
487                          cl_mem & output , unsigned int threadsize , unsigned int

```

```

        problemsize , cl_event * event)
477 {
478     cl_int err;
479     clSetKernelArg( kernel , 0, sizeof( cl_mem ) , &input );
480     clSetKernelArg( kernel , 1, sizeof( cl_mem ) , &output );
481     clSetKernelArg( kernel , 2, sizeof( unsigned int ) , &
482                     threadsize );
483     clSetKernelArg( kernel , 3, sizeof( unsigned int ) , &
484                     problemsize );
485     unsigned numWorkItems;
486     if ( problemsize % ( threadsize*threadsize ) == 0 )
487     {
488         numWorkItems = problemsize/( threadsize*threadsize );
489     }
490     else
491     {
492         numWorkItems = problemsize/( threadsize*threadsize )+1;
493     }
494     const size_t global_size = numWorkItems*threadsize ;
495     const size_t local_size = threadsize ;
496
497     err = clEnqueueNDRangeKernel( queue , kernel , 1, NULL, &
498                                  global_size , &local_size , 0, NULL, event );
499 #ifdef __DEBUG__
500     if ( err < 0 )
501     {
502         std::cout << "Failed to enqueue Norm, code: " << err <<
503         std::endl;
504     }
505 #endif
506 }
507 void OpenCLManager::ParallelSumReductionF( cl_mem & input ,
508                                            cl_mem & output , unsigned int threadsize , unsigned int
509                                            problemsize , cl_event * event )
510 {
511     Norm( kernelReductionF , input , output , threadsize ,
512           problemsize , event );
513 }
514 void OpenCLManager::ParallelSumReductionD( cl_mem & input ,
515                                            cl_mem & output , unsigned int threadsize , unsigned int
516                                            problemsize , cl_event * event )
517 {
518     Norm( kernelReductionD , input , output , threadsize ,
519           problemsize , event );
520 }
521 void OpenCLManager::Norm2F( cl_mem & input , cl_mem & output ,
522                           unsigned int threadsize , unsigned int problemsize ,
523                           cl_event * event )

```

```

513 {
514     Norm(kernelNorm2F, input, output, threadsize, problemsize,
515           event);
515 }
516 void OpenCLManager::Norm2D(cl_mem & input, cl_mem & output,
517                           unsigned int threadsize, unsigned int problemsize,
518                           cl_event * event)
517 {
518     Norm(kernelNorm2D, input, output, threadsize, problemsize,
519           event);
519 }
520 void OpenCLManager::NormInfF(cl_mem & input, cl_mem & output,
521                           unsigned int threadsize, unsigned int problemsize,
522                           cl_event * event)
521 {
522     Norm(kernelNormInfF, input, output, threadsize, problemsize,
523           event);
523 }
524 void OpenCLManager::NormInfD(cl_mem & input, cl_mem & output,
525                           unsigned int threadsize, unsigned int problemsize,
526                           cl_event * event)
525 {
526     Norm(kernelNormInfD, input, output, threadsize, problemsize,
527           event);
527 }
528
529 /**
530 //Matrix vector:
531 /**
532 void OpenCLManager::SparseMatrixVector(cl_kernel kernel,
533                                         cl_mem & matData, cl_mem & matCol, cl_mem & matRow, cl_mem
534                                         & vecData, cl_mem & returnData, unsigned int height,
535                                         unsigned int width, unsigned int numIndexes, unsigned int
536                                         rowVectorLength, cl_event * event)
533 {
534     cl_int err;
535     clSetKernelArg( kernel, 0, sizeof( cl_mem ), &matData );
536     clSetKernelArg( kernel, 1, sizeof( cl_mem ), &matCol );
537     clSetKernelArg( kernel, 2, sizeof( cl_mem ), &matRow );
538     clSetKernelArg( kernel, 3, sizeof( cl_mem ), &vecData );
539     clSetKernelArg( kernel, 4, sizeof( cl_mem ), &returnData );
540     clSetKernelArg( kernel, 5, sizeof( unsigned int ), &
541                     rowVectorLength );
541
542     size_t global_size;
543     if (rowVectorLength % (SparseMatrixVectorThreadsPerGroup *
544                           SparseMatrixVectorRowsPerThread) == 0)
544     {

```



```

        unsigned int numIndexes, unsigned int rowVectorLength,
        cl_event * event)
574 {
575     SparseMatrixVector(kernelSparseMatrixVectorDF, matData,
576                         matCol, matRow, vecData, returnData, height, width,
577                         numIndexes, rowVectorLength, event);
578 }
579 void OpenCLManager::SparseMatrixVectorDD(cl_mem & matData,
580                                         cl_mem & matCol, cl_mem & matRow, cl_mem & vecData, cl_mem
581                                         & returnData, unsigned int height, unsigned int width,
582                                         unsigned int numIndexes, unsigned int rowVectorLength,
583                                         cl_event * event)
584 {
585     cl_int err;
586     clSetKernelArg( kernel, 0, sizeof( cl_mem ), &matData );
587     clSetKernelArg( kernel, 1, sizeof( cl_mem ), &vecData );
588     clSetKernelArg( kernel, 2, sizeof( cl_mem ), &returnData );
589     clSetKernelArg( kernel, 3, sizeof( unsigned int ), &height );
590     clSetKernelArg( kernel, 4, sizeof( unsigned int ), &
591                     bandwidth );
592     clSetKernelArg( kernel, 5, sizeof( unsigned int ), &height );
593     size_t global_size;
594     if (height % (BandMatrixVectorThreadsPerGroup *
595                   BandMatrixVectorRowsPerThread) == 0)
596     {
597         global_size = height / (BandMatrixVectorRowsPerThread);
598     }
599     else
600     {
601         global_size = (height / (BandMatrixVectorRowsPerThread *
602                               BandMatrixVectorThreadsPerGroup) + 1) *
603                               BandMatrixVectorThreadsPerGroup;
604     }
605     size_t local_size = BandMatrixVectorThreadsPerGroup;
606     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
607                                 global_size, &local_size, 0, NULL, event);
608 #ifdef __DEBUG__

```

```

605     if (err < 0)
606     {
607         std::cout << "Failed to enqueue BandMatrixVector, code: "
608             << err << std::endl;
609     }
610 #endif
611 void OpenCLManager::BandMatrixVectorFF(cl_mem & matData,
612                                         cl_mem & vecData, cl_mem & returnData, unsigned int height
613                                         , unsigned int bandwidth, unsigned int length, cl_event * event)
614 {
615     BandMatrixVector(kernelBandMatrixVectorFF, matData, vecData,
616                      returnData, height, bandwidth, length, event);
617 }
618 void OpenCLManager::BandMatrixVectorDD(cl_mem & matData,
619                                         cl_mem & vecData, cl_mem & returnData, unsigned int height
620                                         , unsigned int bandwidth, unsigned int length, cl_event * event)
621 {
622     BandMatrixVector(kernelBandMatrixVectorDD, matData, vecData,
623                      returnData, height, bandwidth, length, event);
624 }
625 void OpenCLManager::BandMatrixVectorFD(cl_mem & matData,
626                                         cl_mem & vecData, cl_mem & returnData, unsigned int height
627                                         , unsigned int bandwidth, unsigned int length, cl_event * event)
628 {
629 //Refinement methods:
630
631 void OpenCLManager::FineToCoarse(cl_kernel kernel, cl_mem &
632                                     fineData, cl_mem & corData, unsigned int corWidth,
633                                     unsigned int corHeight, cl_event * event)
634 {
635     cl_int err;
636     unsigned int fineWidth = 2*corWidth-1;

```

```

635     clSetKernelArg( kernel , 0, sizeof( cl_mem ) , &fineData );
636     clSetKernelArg( kernel , 1, sizeof( cl_mem ) , &corData );
637     clSetKernelArg( kernel , 2, sizeof( unsigned int ) , &
638                     fineWidth );
638     clSetKernelArg( kernel , 3, sizeof( unsigned int ) , &corWidth
639                     );
639
640     size_t global_size ;
641     if (corWidth*corHeight % (FTCThreadsPerGroup *
642                               FTCRowsPerThread) == 0)
642     {
643         global_size = corWidth*corHeight /(FTCRowsPerThread) ;
644     }
645     else
646     {
647         global_size = (corWidth*corHeight /(FTCThreadsPerGroup *
648                         FTCRowsPerThread)+1) * FTCThreadsPerGroup ;
648     }
649     size_t local_size = FTCThreadsPerGroup ;
650     err = clEnqueueNDRangeKernel(queue, kernel , 1, NULL, &
651                                   global_size , &local_size , 0, NULL, event );
651 #ifdef __DEBUG__
652     if (err < 0)
653     {
654         std::cout << "Failed to enqueue FineToCoarse , code: " <<
655                     err << std::endl ;
655     }
656 #endif
657 }
658 void OpenCLManager::FineToCoarseFF(cl_mem & fineData , cl_mem &
659                                     corData , unsigned int corWidth , unsigned int corHeight ,
660                                     cl_event * event )
661 {
660     FineToCoarse(kernelRefineFTCFF , fineData , corData , corWidth ,
661                   corHeight , event );
661 }
662 void OpenCLManager::FineToCoarseDF(cl_mem & fineData , cl_mem &
663                                     corData , unsigned int corWidth , unsigned int corHeight ,
664                                     cl_event * event )
663 {
664     FineToCoarse(kernelRefineFTCDF , fineData , corData , corWidth ,
665                   corHeight , event );
665 }
666 void OpenCLManager::FineToCoarseDD(cl_mem & fineData , cl_mem &
667                                     corData , unsigned int corWidth , unsigned int corHeight ,
668                                     cl_event * event )
667 {
668     FineToCoarse(kernelRefineFTCDD , fineData , corData , corWidth ,
669                   corHeight , event );

```

```

669 }
670 void OpenCLManager::CoarseToFine(cl_kernel kernel, cl_mem &
671     fineData, cl_mem & corData, unsigned int fineWidth,
672     unsigned int fineHeight, cl_event * event)
673 {
674     cl_int err;
675     unsigned int corWidth = fineWidth/2+1;
676     clSetKernelArg( kernel, 0, sizeof( cl_mem ), &fineData );
677     clSetKernelArg( kernel, 1, sizeof( cl_mem ), &corData );
678     clSetKernelArg( kernel, 2, sizeof( unsigned int ), &corWidth
679                     );
680     clSetKernelArg( kernel, 3, sizeof( unsigned int ), &
681                     fineWidth );
682     size_t global_size;
683     if (fineWidth*fineHeight % (CTFThreadsPerGroup *
684         CTFRowsPerThread) == 0)
685     {
686         global_size = fineWidth*fineHeight/(CTFRowsPerThread);
687     }
688     else
689     {
690         global_size = (fineWidth*fineHeight/(CTFThreadsPerGroup *
691                         CTFRowsPerThread)+1) * CTFThreadsPerGroup;
692     }
693     size_t local_size = CTFThreadsPerGroup;
694     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
695         global_size, &local_size, 0, NULL, event);
696 #ifdef __DEBUG__
697     if (err < 0)
698     {
699         std::cout << "Failed to enqueue CoarseToFine, code: " <<
700             err << std::endl;
701     }
702 #endif
703 }
704 void OpenCLManager::CoarseToFineFF(cl_mem & fineData, cl_mem &
705     corData, unsigned int fineWidth, unsigned int fineHeight,
706     cl_event * event)
707 {
708     CoarseToFine(kernelRefineCTFFF, fineData, corData, fineWidth
709                 , fineHeight, event);
710 }
711 void OpenCLManager::CoarseToFineFD(cl_mem & fineData, cl_mem &
712     corData, unsigned int fineWidth, unsigned int fineHeight,
713     cl_event * event)
714 {
715     CoarseToFine(kernelRefineCTFFD, fineData, corData, fineWidth
716                 , fineHeight, event);

```

```

704 }
705 void OpenCLManager::CoarseToFineDD(cl_mem & fineData , cl_mem &
706                                     corData , unsigned int fineWidth , unsigned int fineHeight ,
707                                     cl_event * event)
708 {
709     CoarseToFine(kernelRefineCTFDD , fineData , corData , fineWidth
710                 , fineHeight , event);
711 }
712 //Jacobi methods:
713 //
714 void OpenCLManager::JacobiF(cl_mem & output , cl_mem & input ,
715                             cl_mem & rightData , unsigned int width , unsigned int
716                             height , float spacing , unsigned int gridSize , cl_event *
717                             event)
718 {
719     cl_int err;
720     clSetKernelArg( kernelJacobiF , 0 , sizeof( cl_mem ) , &output )
721             ;
722     clSetKernelArg( kernelJacobiF , 1 , sizeof( cl_mem ) , &input );
723     clSetKernelArg( kernelJacobiF , 2 , sizeof( cl_mem ) , &
724                     rightData );
725     clSetKernelArg( kernelJacobiF , 3 , sizeof( unsigned int ) , &
726                     width );
727     clSetKernelArg( kernelJacobiF , 4 , sizeof( unsigned int ) , &
728                     height );
729     clSetKernelArg( kernelJacobiF , 5 , sizeof( float ) , &spacing )
730             ;
731     size_t global_size;
732     if (width*height % (JacobiThreadsPerGroup *
733                         JacobiRowsPerThread) == 0)
734     {
735         global_size = width*height/(JacobiRowsPerThread);
736     }
737     else
738     {
739         global_size = (width*height/(JacobiThreadsPerGroup *
740                         JacobiRowsPerThread)+1) * JacobiThreadsPerGroup;
741     }
742     size_t local_size = JacobiThreadsPerGroup;
743     err = clEnqueueNDRangeKernel(queue , kernelJacobiF , 1 , NULL,
744                                 &global_size , &local_size , 0 , NULL, event);
745 #ifdef __DEBUG__
746     if (err < 0)
747     {

```

```

738     std::cout << "Failed to enqueue kernel JacobiF, code: " <<
739         err << std::endl;
740     }
741 #endif
742 }
743 void OpenCLManager::JacobiD(cl_mem & output, cl_mem & input,
744                             cl_mem & rightData, unsigned int width, unsigned int
745                             height, double spacing, unsigned int gridSize, cl_event *
746                             event)
747 {
748     cl_int err;
749     clSetKernelArg( kernelJacobiD, 0, sizeof( cl_mem ), &output )
750         ;
751     clSetKernelArg( kernelJacobiD, 1, sizeof( cl_mem ), &input );
752     clSetKernelArg( kernelJacobiD, 2, sizeof( cl_mem ), &
753                     rightData );
754     clSetKernelArg( kernelJacobiD, 3, sizeof( unsigned int ), &
755                     width );
756     clSetKernelArg( kernelJacobiD, 4, sizeof( unsigned int ), &
757                     height );
758     clSetKernelArg( kernelJacobiD, 5, sizeof( double ), &spacing
759                     );
760
761     size_t global_size;
762     if (width*height % (JacobiThreadsPerGroup *
763                         JacobiRowsPerThread) == 0)
764     {
765         global_size = width*height/(JacobiRowsPerThread);
766     }
767     else
768     {
769         global_size = (width*height/(JacobiThreadsPerGroup *
770                         JacobiRowsPerThread)+1) * JacobiThreadsPerGroup;
771     }
772     size_t local_size = JacobiThreadsPerGroup;
773     err = clEnqueueNDRangeKernel(queue, kernelJacobiD, 1, NULL,
774                                 &global_size, &local_size, 0, NULL, event);
775 #ifdef __DEBUG__
776     if (err < 0)
777     {
778         std::cout << "Failed to enqueue kernel JacobiF, code: " <<
779             err << std::endl;
780     }
781 #endif
782 }
783
784 void OpenCLManager::JacobiMethodF(cl_kernel kernel, cl_mem &
785                                   leftData, cl_mem & rightData, unsigned int width, unsigned

```

```

    int height , float spacing , unsigned int gridSize ,
    cl_event * event)
773 {
774     cl_int err;
775     clSetKernelArg( kernel , 0, sizeof( cl_mem ) , &leftData );
776     clSetKernelArg( kernel , 1, sizeof( cl_mem ) , &rightData );
777     clSetKernelArg( kernel , 2, sizeof( unsigned int ) , &width );
778     clSetKernelArg( kernel , 3, sizeof( unsigned int ) , &height );
779     clSetKernelArg( kernel , 4, sizeof( float ) , &spacing );
780
781     size_t global_size;
782     if (width*height % (RBGSThreadsPerGroup * RBGSRowsPerThread)
783         == 0)
784     {
785         global_size = width*height/(RBGSRowsPerThread);
786     }
787     else
788     {
789         global_size = (width*height/(RBGSThreadsPerGroup *
790                         RBGSRowsPerThread)+1) * RBGSThreadsPerGroup;
791     }
792     size_t local_size = RBGSThreadsPerGroup;
793     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
794         global_size, &local_size, 0, NULL, event);
795     #ifdef __DEBUG__
796     if (err < 0)
797     {
798         std::cout << "Failed to enqueue kernel JackkobiMethodOdd ,
799             code: " << err << std::endl;
800     }
801     #endif
802 }
803 void OpenCLManager::JacobiMethodD( cl_kernel kernel , cl_mem &
804     leftData , cl_mem & rightData , unsigned int width , unsigned
805     int height , double spacing , unsigned int gridSize ,
806     cl_event * event)
807 {
808     cl_int err;
809     clSetKernelArg( kernel , 0, sizeof( cl_mem ) , &leftData );
810     clSetKernelArg( kernel , 1, sizeof( cl_mem ) , &rightData );
811     clSetKernelArg( kernel , 2, sizeof( unsigned int ) , &width );
812     clSetKernelArg( kernel , 3, sizeof( unsigned int ) , &height );
813     clSetKernelArg( kernel , 4, sizeof( double ) , &spacing );
814
815     size_t global_size;
816     if (width*height % (RBGSThreadsPerGroup * RBGSRowsPerThread)
817         == 0)
818     {
819         global_size = width*height/(RBGSRowsPerThread);
820     }
821 }
```

```

812     }
813     else
814     {
815         global_size = (width*height/(RBGSThreadsPerGroup *
816                         RBGSRowsPerThread)+1) * RBGSThreadsPerGroup;
817     size_t local_size = RBGSThreadsPerGroup;
818
819     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &
820                                   global_size, &local_size, 0, NULL, event);
821 #ifdef __DEBUG__
822     if (err < 0)
823     {
824         std::cout << "Failed to enqueue kernel JacobiMethodOdd,
825                     code: " << err << std::endl;
826     }
827 #endif
828 }
829 void OpenCLManager::JacobiMethodOddD(cl_mem & leftData, cl_mem
830 & rightData, unsigned int width, unsigned int height,
831 double spacing, unsigned int gridSize, cl_event * event)
832 {
833     JacobiMethodD(kernelJacobiMethodOddD, leftData, rightData,
834                 width, height, spacing, gridSize, event);
835 }
836 void OpenCLManager::JacobiMethodEvenD(cl_mem & leftData,
837 cl_mem & rightData, unsigned int width, unsigned int
838 height, double spacing, unsigned int gridSize, cl_event * event)
839 {
840     JacobiMethodD(kernelJacobiMethodEvenD, leftData, rightData,
841                 width, height, spacing, gridSize, event);
842 }
843 void OpenCLManager::JacobiMethodOddF(cl_mem & leftData,
844 cl_mem & rightData, unsigned int width, unsigned int
845 height, float spacing, unsigned int gridSize, cl_event * event)
846 {
847     JacobiMethodF(kernelJacobiMethodOddF, leftData, rightData,
848                 width, height, spacing, gridSize, event);
849 }
850 void OpenCLManager::JacobiMethodEvenF(cl_mem & leftData,
851 cl_mem & rightData, unsigned int width, unsigned int
852 height, float spacing, unsigned int gridSize, cl_event * event)
853 {
854     JacobiMethodF(kernelJacobiMethodEvenF, leftData, rightData,
855                 width, height, spacing, gridSize, event);
856 }
857 
```

```

844
845 void OpenCLManager::JacobiDefectF(cl_mem & leftData , cl_mem &
846   rightData , cl_mem & defectData , unsigned int width ,
847   unsigned int height , float spacing , cl_event * event)
848 {
849   cl_int err ;
850   clSetKernelArg(kernelJacobiDefectF , 0 , sizeof( cl_mem ) , &
851   leftData );
852   clSetKernelArg(kernelJacobiDefectF , 1 , sizeof( cl_mem ) , &
853   rightData );
854   clSetKernelArg(kernelJacobiDefectF , 2 , sizeof( cl_mem ) , &
855   defectData );
856   clSetKernelArg(kernelJacobiDefectF , 3 , sizeof( unsigned int )
857   , &width );
858   clSetKernelArg(kernelJacobiDefectF , 4 , sizeof( unsigned int )
859   , &height );
860   clSetKernelArg(kernelJacobiDefectF , 5 , sizeof( float ) , &
861   spacing );
862
863   size_t global_size ;
864   if (width*height % (DefectThreadsPerGroup *
865     DefectRowsPerThread) == 0)
866   {
867     global_size = width*height/(DefectRowsPerThread) ;
868   }
869   else
870   {
871     global_size = (width*height/(DefectThreadsPerGroup *
872     DefectRowsPerThread)+1) * DefectThreadsPerGroup ;
873
874   size_t local_size = DefectThreadsPerGroup ;
875   err = clEnqueueNDRangeKernel(queue , kernelJacobiDefectF , 1 ,
876     NULL , &global_size , &local_size , 0 , NULL , event );
877
878 #ifdef __DEBUG__
879   if (err < 0)
880   {
881     std::cout << "Failed to enqueue kernel JacobiDefect" <<
882       std::endl ;
883   }
884 #endif
885 }
886
887 void OpenCLManager::JacobiDefectD(cl_mem & leftData , cl_mem &
888   rightData , cl_mem & defectData , unsigned int width ,
889   unsigned int height , double spacing , cl_event * event)
890 {
891   cl_int err ;
892   clSetKernelArg(kernelJacobiDefectD , 0 , sizeof( cl_mem ) , &
893   leftData );

```

```

877     clSetKernelArg(kernelJacobiDefectD , 1, sizeof( cl_mem ) , &
878                     rightData);
879     clSetKernelArg(kernelJacobiDefectD , 2, sizeof( cl_mem ) , &
880                     defectData);
881     clSetKernelArg(kernelJacobiDefectD , 3, sizeof( unsigned int
882                     ) , &width);
883     clSetKernelArg(kernelJacobiDefectD , 4, sizeof( unsigned int
884                     ) , &height);
885     clSetKernelArg(kernelJacobiDefectD , 5, sizeof( double ) , &
886                     spacing);
887
888     size_t global_size;
889     if (width*height % (DefectThreadsPerGroup *
890                         DefectRowsPerThread) == 0)
891     {
892         global_size = width*height/(DefectRowsPerThread);
893     }
894     else
895     {
896         global_size = (width*height/(DefectThreadsPerGroup *
897                         DefectRowsPerThread)+1) * DefectThreadsPerGroup;
898     }
899     size_t local_size = DefectThreadsPerGroup;
900
901     err = clEnqueueNDRangeKernel(queue, kernelJacobiDefectD , 1,
902                                   NULL, &global_size , &local_size , 0, NULL, event);
903 #ifdef __DEBUG__
904     if (err < 0)
905     {
906         std::cout << "Failed to enqueue kernel JacobiDefect" <<
907             std::endl;
908     }
909 #endif
910 }
911
912 // Methods:
913
914 void OpenCLManager::AddSource( char * name )
915 {
916     numSourceFiles++;
917     char ** temp = (char **)mxMalloc( sizeof(char *) *
918                                     numSourceFiles);
919     mexMakeMemoryPersistent(temp);
920     for (unsigned int i = 0; i < numSourceFiles-1; i += 1)
921     {
922         temp[ i ] = vectorSourceFiles[ i ];
923     }
924     temp[ numSourceFiles -1] = name;

```

```
916     if (vectorSourceFiles != NULL)
917         mxFree( vectorSourceFiles );
918     vectorSourceFiles = temp;
919 }
920
921 /**
922 //Memory deletion :
923 /**
924 void OpenCLManager::DeleteMemory(__MemoryControl__<float> *
925     mem)
925 {
926     clReleaseMemObject(mem->buffer);
927     mxFree(mem->data);
928     mxFree(mem);
929 }
930 void OpenCLManager::DeleteMemory(__MemoryControl__<double> *
931     mem)
931 {
932     clReleaseMemObject(mem->buffer);
933     mxFree(mem->data);
934     mxFree(mem);
935 }
936 void OpenCLManager::DeleteIndex(__IndexControl__ * mem)
937 {
938     clReleaseMemObject(mem->buffer);
939     mxFree(mem->data);
940     mxFree(mem);
941 }
942 /**
943 //OpenCL control flow :
944 /**
945 void OpenCLManager::SetActiveGPU(unsigned int index)
946 {
947     unsigned int DeviceIndex = 0;
948     unsigned int PlatformIndex = 0;
949     while (true)
950     {
951         if (PlatformIndex >= numPlatforms)
952         {
953             mexPrintf("Failed to set GPU: bad index\n");
954             return;
955         }
956         if (index >= numDevices[PlatformIndex])
957         {
958             index -= numDevices[PlatformIndex];
959             PlatformIndex++;
960         }
961     else
962     {
```

```

963     DeviceIndex = index ;
964     break ;
965   }
966 }
967
968 //find maker of card:
969 char platformname[64];
970 size_t platformnamesize = 64;
971 clGetPlatformInfo(vectorPlatforms[PlatformIndex] ,
972     CL_PLATFORM_NAME, platformnamesize , platformname , NULL) ;
973 if ( (platformname[0] == 'A') && (platformname[1] == 'M') &&
974     (platformname[2] == 'D') )
975 {
976     NVIDIA = false ;
977 }
978
979 //Save current settings:
980 cl_int err ;
981 platform = vectorPlatforms[PlatformIndex] ;
982 device = vectorDevices[PlatformIndex][DeviceIndex] ;
983
984 //create context:
985 context = clCreateContext(NULL, 1 , &device , NULL, NULL, &err
986 );
987 #ifdef __DEBUG__
988 if (err < 0)
989 {
990     std::cout << "Error creating program, code: " << err <<
991     std::endl ;
992 }
993 #endif
994
995 //Load all kernels:
996 program_strings = (char **)mxMalloc( sizeof(char *) *
997     numSourceFiles) ;
998 mexMakeMemoryPersistent(program_strings) ;
999 program_sizes = (size_t *)mxMalloc( sizeof(size_t) *
1000     numSourceFiles) ;
1001 mexMakeMemoryPersistent(program_sizes) ;
1002 FILE * program_handle ;
1003
1004 for ( unsigned int i = 0; i < numSourceFiles; i += 1)
1005 {
1006     program_handle = fopen(vectorSourceFiles[i] , "r") ;
1007     fseek(program_handle , 0 , SEEK_END) ;
1008     program_sizes[i] = ftell(program_handle) ;
1009     rewind(program_handle) ;
1010     program_strings[i] = (char *)mxMalloc( sizeof(char) * (
1011         program_sizes[i]+1)) ;

```

```

1005     mexMakeMemoryPersistent(program_strings[i]);
1006     program_strings[i][program_sizes[i]] = '\0';
1007     fread(program_strings[i], sizeof(char), program_sizes[i],
1008           program_handle);
1009 }
1010 //create program:
1011 program = clCreateProgramWithSource(context, numSourceFiles,
1012                                     (const char **)program_strings, program_sizes, &err);
1013 if (err < 0)
1014     mexPrintf("Fail...\n", err);
1015 #ifdef __DEBUG__
1016     if (err < 0)
1017     {
1018         std::cout << "Error creating program, code: " << err <<
1019                     std::endl;
1020     }
1021 #endif
1022 //build program:
1023 char * options;
1024 if (EnableDouble && NVIDIA)
1025 {
1026     options = "-D__DOUBLE_ALLOWED__ -IKernels -D__NVIDIA__";
1027 }
1028 else if (EnableDouble)
1029 {
1030     options = "-D__DOUBLE_ALLOWED__ -IKernels"; //
1031 }
1032 else
1033 {
1034     options = "-IKernels";
1035 }
1036 const char * optionsConst = options;
1037 err = clBuildProgram(program, 1, &device, optionsConst, NULL,
1038                      NULL);
1039 #ifdef __DEBUG__
1040     if (err < 0)
1041     {
1042         std::cout << "Error building program, code: ";
1043         WriteError(err);
1044     }
1045 #endif
1046 //Program build info:
1047 #ifdef __PROGRAM_BUILD_INFO__
1048     size_t program_log_size;

```

```

1049     clGetProgramBuildInfo(program, device,
1050         CL_PROGRAM_BUILD_LOG, 0, NULL, &program_log_size);
1051     char * program_log = (char *)mxMalloc(sizeof(char)*
1052         program_log_size);
1053     clGetProgramBuildInfo(program, device,
1054         CL_PROGRAM_BUILD_LOG, program_log_size, program_log,
1055         NULL);
1056     std::cout << program_log << std::endl;
1057 #endif
1058 //create kernels:
1059 kernelReductionF = CreateKernel((char*)"VectorReductionF");
1060 kernelVectorTimesConstantFF = CreateKernel((char*)""
1061     VectorTimesConstantFF");
1062 kernelJacobiF = CreateKernel((char*)"JacobiMethodF");
1063 kernelJacobiMethodOddF = CreateKernel((char*)""
1064     JacobiMethodOddF");
1065 kernelJacobiMethodEvenF = CreateKernel((char*)""
1066     JacobiMethodEvenF");
1067 kernelJacobiDefectF = CreateKernel((char*)"JacobiCalcDefectF"
1068     ");
1069 kernelRefineFTCFF = CreateKernel((char*)"RefineFTCFF");
1070 kernelRefineCTFFF = CreateKernel((char*)"RefineCTFFF");
1071 kernelVectorMinusVectorFF = CreateKernel((char*)""
1072     VectorMinusVectorFF");
1073 kernelVectorMinusVectorConstantFF = CreateKernel((char*)""
1074     VectorMinusVectorConstantFF");
1075 kernelNorm2F = CreateKernel((char*)"Norm2F");
1076 kernelNormInff = CreateKernel((char*)"NormInff");
1077 kernelVectorPlusVectorFF = CreateKernel((char*)""
1078     VectorPlusVectorFF");
1079 kernelReductionD = CreateKernel((char*)"VectorReductionD");
1080 kernelSparseMatrixVectorFF = CreateKernel((char*)""
1081     SparseMatrixVectorFF");
1082 kernelBandMatrixVectorFF = CreateKernel((char*)""
1083     BandMatrixVectorFF");
1084 if (EnableDouble)
1085 {
1086     kernelVectorTimesConstantFD = CreateKernel((char*)""
1087     VectorTimesConstantFD");
1088     kernelVectorTimesConstantDD = CreateKernel((char*)""
1089     VectorTimesConstantDD");
1090     kernelJacobiD = CreateKernel((char*)"JacobiMethodD");
1091     kernelJacobiMethodOddD = CreateKernel((char*)""
1092     JacobiMethodOddD");
1093     kernelJacobiMethodEvenD = CreateKernel((char*)""
1094     JacobiMethodEvenD");
1095     kernelJacobiDefectD = CreateKernel((char*)""
1096     JacobiCalcDefectD");

```

```

1080     kernelRefineFTCDF = CreateKernel((char*)"RefineFTCDF");
1081     kernelRefineFTCDD = CreateKernel((char*)"RefineFTCDD");
1082     kernelRefineCTFFD = CreateKernel((char*)"RefineCTFFD");
1083     kernelRefineCTFDD = CreateKernel((char*)"RefineCTFDD");
1084     kernelVectorMinusVectorFD = CreateKernel((char*)""
1085         "VectorMinusVectorFD");
1086     kernelVectorMinusVectorDF = CreateKernel((char*)""
1087         "VectorMinusVectorDF");
1088     kernelVectorMinusVectorDD = CreateKernel((char*)""
1089         "VectorMinusVectorDD");
1090     kernelVectorMinusVectorConstantFD = CreateKernel((char*)""
1091         "VectorMinusVectorConstantFD");
1092     kernelVectorMinusVectorConstantDF = CreateKernel((char*)""
1093         "VectorMinusVectorConstantDF");
1094     kernelVectorMinusVectorConstantDD = CreateKernel((char*)""
1095         "VectorMinusVectorConstantDD");
1096     kernelNorm2D = CreateKernel((char*)"Norm2D");
1097     kernelNormInfD = CreateKernel((char*)"NormInfD");
1098     kernelVectorPlusVectorFD = CreateKernel((char*)""
1099         "VectorPlusVectorFF");
1100     kernelVectorPlusVectorDF = CreateKernel((char*)""
1101         "VectorPlusVectorFF");
1102     kernelVectorPlusVectorDD = CreateKernel((char*)""
1103         "VectorPlusVectorFF");
1104     kernelSparseMatrixVectorDF = CreateKernel((char*)""
1105         "SparseMatrixVectorDF");
1106     kernelSparseMatrixVectorDD = CreateKernel((char*)""
1107         "SparseMatrixVectorDD");
1108     kernelSparseMatrixVectorFD = CreateKernel((char*)""
1109         "SparseMatrixVectorFD");
1110     kernelBandMatrixVectorFD = CreateKernel((char*)""
1111         "BandMatrixVectorFD");
1112     kernelBandMatrixVectorDF = CreateKernel((char*)""
1113         "BandMatrixVectorDF");
1114     kernelBandMatrixVectorDD = CreateKernel((char*)""
1115         "BandMatrixVectorDD");
1116 }
1117
1118 queue = clCreateCommandQueue(context, device,
1119     CL_QUEUE_PROFILING_ENABLE, &err);
1120 #ifdef __DEBUG__
1121     if (err < 0)
1122     {
1123         std::cout << "Error creating queue, code: " << err <<
1124             std::endl;
1125     }
1126 #endif
1127 }
```

```
1|112 //  
1|113 // Allocators : void  
1|114 //  
1|115 //  
1|116 OpenCLManager :: OpenCLManager ()  
1|117 {  
1|118     cl_int err;  
1|119     NVIDIA = true; //assume nvidia card.  
1|120     //Init variables:  
1|121     numSourceFiles = 0;  
1|122     numIndex = 0;  
1|123     numMemoryF = 0;  
1|124     numMemoryD = 0;  
1|125     capMemoryF = 1024;  
1|126     capMemoryD = 1024;  
1|127     capIndex = 1024;  
1|128     vectorSourceFiles = NULL;  
1|129     vectorIndex = (_IndexControl **)mxMalloc( sizeof( _IndexControl ) * capIndex );  
1|130     vectorMemoryD = (_MemoryControl <double> **)mxMalloc( sizeof( _MemoryControl <double> ) * capMemoryD );  
1|131     vectorMemoryF = (_MemoryControl <float> **)mxMalloc( sizeof( _MemoryControl <float> ) * capMemoryF );  
1|132     mexMakeMemoryPersistent( vectorIndex );  
1|133     mexMakeMemoryPersistent( vectorMemoryD );  
1|134     mexMakeMemoryPersistent( vectorMemoryF );  
1|135     EnableDouble = false;  
1|136  
1|137     //Tuning stuff:  
1|138     SparseMatrixVectorRowsPerThread = 64;  
1|139     SparseMatrixVectorThreadsPerGroup = 64;  
1|140     BandMatrixVectorRowsPerThread = 64;  
1|141     BandMatrixVectorThreadsPerGroup = 64;  
1|142     NormRowsPerThread = 64;  
1|143     NormThreadsPerGroup = 64;  
1|144     VectorAndVectorRowsPerThread = 64;  
1|145     VectorAndVectorThreadsPerGroup = 64;  
1|146     VectorConstantRowsPerThread = 64;  
1|147     VectorConstantThreadsPerGroup = 64;  
1|148     JacobiRowsPerThread = 64;  
1|149     JacobiThreadsPerGroup = 64;  
1|150     RBGSRowsPerThread = 64;  
1|151     RBGSThreadsPerGroup = 64;  
1|152     DefectRowsPerThread = 64;  
1|153     DefectThreadsPerGroup = 64;  
1|154     FTCRowsPerThread = 64;  
1|155     FTCThreadsPerGroup = 64;  
1|156     CTFRowsPerThread = 64;  
1|157     CTFThreadsPerGroup = 64;
```

```

1158 //Init OpenCL Stuff:
1159
1160 err = clGetPlatformIDs(0, NULL, &numPlatforms);
1161 vectorPlatforms = (cl_platform_id *)mxMalloc( sizeof(
1162     cl_platform_id) * numPlatforms);
1163 mexMakeMemoryPersistent(vectorPlatforms);
1164 err = clGetPlatformIDs(numPlatforms, vectorPlatforms, NULL);
1165 vectorDevices = (cl_device_id **)mxMalloc( sizeof(
1166     cl_device_id) * numPlatforms);
1167 mexMakeMemoryPersistent(vectorDevices);
1168 numDevices = (unsigned int *)mxMalloc( sizeof(unsigned int)
1169     * numPlatforms);
1170 mexMakeMemoryPersistent(numDevices);
1171
1172 for (unsigned int i = 0; i < numPlatforms; i += 1)
1173 {
1174     cl_uint testvar = 0;
1175     clGetDeviceIDs(vectorPlatforms[i], CL_DEVICE_TYPE_GPU, 0,
1176                     NULL, &testvar);
1177     numDevices[i] = testvar;
1178     mexPrintf("number of devices: %u\n", numDevices[i]);
1179     vectorDevices[i] = (cl_device_id *)mxMalloc( sizeof(
1180         cl_device_id) * numDevices[i]);
1181     mexMakeMemoryPersistent(vectorDevices[i]);
1182     clGetDeviceIDs(vectorPlatforms[i], CL_DEVICE_TYPE_GPU,
1183                     numDevices[i], vectorDevices[i], NULL);
1184 }
1185 AddSource((char *)"Kernels/VectorReduction.cl");
1186 AddSource((char *)"Kernels/BandMatrixVector.cl");
1187 AddSource((char *)"Kernels/SparseMatrixVector.cl");
1188 AddSource((char *)"Kernels/JacobiMethod.cl");
1189 AddSource((char *)"Kernels/JacobiMethodEven.cl");
1190 AddSource((char *)"Kernels/JacobiMethodOdd.cl");
1191 AddSource((char *)"Kernels/JacobiCalcDefect.cl");
1192 AddSource((char *)"Kernels/Norm2Smart.cl");
1193 AddSource((char *)"Kernels/NormInfSmart.cl");
1194 AddSource((char *)"Kernels/RefineFTC.cl");
1195 AddSource((char *)"Kernels/RefineCTF.cl");
1196 AddSource((char *)"Kernels/VectorTimesConstant.cl");
1197 AddSource((char *)"Kernels/VectorPlusVector.cl");
1198 AddSource((char *)"Kernels/VectorMinusVector.cl");
1199 AddSource((char *)"Kernels/VectorMinusVectorConstant.cl");
1200
1201 //Deallocators:

```

```
1201 //  
1202 OpenCLManager :: ~OpenCLManager ()  
1203 {  
1204     ReleaseOpenCL ();  
1205 }  
1206 void OpenCLManager :: ReleaseOpenCL ()  
1207 {  
1208     #ifdef __DEBUG__  
1209         unsigned int count = 0;  
1210         for (unsigned int i = 0; i < numMemoryD; i += 1)  
1211         {  
1212             if (vectorMemoryD [ i ] != NULL)  
1213                 count++;  
1214         }  
1215         for (unsigned int i = 0; i < numMemoryF; i += 1)  
1216         {  
1217             if (vectorMemoryF [ i ] != NULL)  
1218                 count++;  
1219         }  
1220         mexPrintf("Number of memory spots assigned: %u, number of  
1221             memory leaks: %u\n", numMemoryD+numMemoryF, count);  
1222 #endif  
1223 //Internal ordering:  
1224 mxFree (vectorMemoryD);  
1225 mxFree (vectorMemoryF);  
1226 mxFree (vectorIndex);  
1227 mxFree (vectorSourceFiles);  
1228 //kernels:  
1229 clReleaseKernel (kernelReductionF);  
1230 clReleaseKernel (kernelJacobiMethodOddF);  
1231 clReleaseKernel (kernelJacobiMethodEvenF);  
1232 clReleaseKernel (kernelJacobiDefectF);  
1233 clReleaseKernel (kernelRefineCTFFF);  
1234 clReleaseKernel (kernelRefineFTCFF);  
1235 clReleaseKernel (kernelVectorTimesConstantFF);  
1236 clReleaseKernel (kernelVectorMinusVectorFF);  
1237 clReleaseKernel (kernelVectorMinusVectorConstantFF);  
1238 clReleaseKernel (kernelVectorPlusVectorFF);  
1239 clReleaseKernel (kernelNormInfF);  
1240 clReleaseKernel (kernelNorm2F);  
1241 clReleaseKernel (kernelSparseMatrixVectorFF);  
1242 if (EnableDouble)  
1243 {  
1244     clReleaseKernel (kernelReductionD);  
1245     clReleaseKernel (kernelJacobiMethodOddD);  
1246     clReleaseKernel (kernelJacobiMethodEvenD);  
1247     clReleaseKernel (kernelJacobiDefectD);  
1248     clReleaseKernel (kernelRefineCTFFD);
```

```

1249     clReleaseKernel(kernelRefineCTFDD);
1250     clReleaseKernel(kernelRefineFTCDF);
1251     clReleaseKernel(kernelRefineFTCDD);
1252     clReleaseKernel(kernelVectorTimesConstantFD);
1253     clReleaseKernel(kernelVectorTimesConstantDD);
1254     clReleaseKernel(kernelVectorMinusVectorFD);
1255     clReleaseKernel(kernelVectorMinusVectorDF);
1256     clReleaseKernel(kernelVectorMinusVectorDD);
1257     clReleaseKernel(kernelVectorMinusVectorConstantFD);
1258     clReleaseKernel(kernelVectorMinusVectorConstantDF);
1259     clReleaseKernel(kernelVectorMinusVectorConstantDD);
1260     clReleaseKernel(kernelVectorPlusVectorFD);
1261     clReleaseKernel(kernelVectorPlusVectorDF);
1262     clReleaseKernel(kernelVectorPlusVectorDD);
1263     clReleaseKernel(kernelNormInfD);
1264     clReleaseKernel(kernelNorm2D);
1265     clReleaseKernel(kernelSparseMatrixVectorDD);
1266     clReleaseKernel(kernelSparseMatrixVectorFD);
1267     clReleaseKernel(kernelSparseMatrixVectorDF);
1268 }
1269 //clean up:
1270 clReleaseCommandQueue(queue);
1271 clReleaseProgram(program);
1272 clReleaseContext(context);
1273
1274 //clean up rest:
1275 mxFree(vectorPlatforms);
1276 for (unsigned int i = 0; i < numPlatforms; i++)
1277 {
1278     mxFree(vectorDevices[i]);
1279 }
1280 mxFree(vectorDevices);
1281 mxFree(numDevices);
1282 mxFree(program_strings);
1283 mxFree(program_sizes);
1284 }
1285
1286 //Shortcuts:
1287 cl_kernel OpenCLManager::CreateKernel(char * name)
1288 {
1289     cl_int err;
1290     cl_kernel temp = clCreateKernel(program, name, &err);
1291 #ifdef __DEBUG__
1292     if (err < 0)
1293     {
1294         mexPrintf("Failed to create kernel %s\n", name);
1295     }
1296 #endif
1297     return temp;

```

```
1298 }
1299
1300 void OpenCLManager::AllowDouble()
1301 {
1302     EnableDouble = true;
1303 }
1304
1305 void OpenCLManager::WaitForCPU()
1306 {
1307     clFinish(queue);
1308 }
1309
1310 float OpenCLManager::GetExecutionTime(cl_event * event)
1311 {
1312     cl_ulong start, end;
1313     clGetEventProfilingInfo(*event, CL_PROFILING_COMMAND_END,
1314                             sizeof(cl_ulong), &end, NULL);
1314     clGetEventProfilingInfo(*event, CL_PROFILING_COMMAND_START,
1315                             sizeof(cl_ulong), &start, NULL);
1315
1316     return (end - start) * 1.0e-3f;
1317 }
1318
1319 //Autotuning functions:
1320 void OpenCLManager::SetSparseMatrixVectorRowsPerThread(size_t
1321             newvalue)
1321 {
1322     SparseMatrixVectorRowsPerThread = newvalue;
1323 }
1324 void OpenCLManager::SetSparseMatrixVectorThreadsPerGroup(
1325             size_t newvalue)
1325 {
1326     SparseMatrixVectorThreadsPerGroup = newvalue;
1327 }
1328 void OpenCLManager::SetBandMatrixVectorRowsPerThread(size_t
1329             newvalue)
1329 {
1330     BandMatrixVectorRowsPerThread = newvalue;
1331 }
1332 void OpenCLManager::SetBandMatrixVectorThreadsPerGroup(size_t
1333             newvalue)
1333 {
1334     BandMatrixVectorThreadsPerGroup = newvalue;
1335 }
1336 void OpenCLManager::SetNormRowsPerThread(size_t newvalue)
1337 {
1338     NormRowsPerThread = newvalue;
1339 }
1340 void OpenCLManager::SetNormThreadsPerGroup(size_t newvalue)
```

```
1341 {
1342     NormThreadsPerGroup = newvalue;
1343 }
1344 void OpenCLManager::SetVectorAndVectorRowsPerThread( size_t
1345     newvalue)
1346 {
1347     VectorAndVectorRowsPerThread = newvalue;
1348 }
1349 void OpenCLManager::SetVectorAndVectorThreadsPerGroup( size_t
1350     newvalue)
1351 {
1352     VectorAndVectorThreadsPerGroup = newvalue;
1353 }
1354 void OpenCLManager::SetVectorConstantRowsPerThread( size_t
1355     newvalue)
1356 {
1357     VectorConstantRowsPerThread = newvalue;
1358 }
1359 void OpenCLManager::SetVectorConstantThreadsPerGroup( size_t
1360     newvalue)
1361 {
1362     VectorConstantThreadsPerGroup = newvalue;
1363 }
1364 void OpenCLManager::SetJacobiRowsPerThread( size_t newvalue)
1365 {
1366     JacobiRowsPerThread = newvalue;
1367 }
1368 void OpenCLManager::SetJacobiThreadsPerGroup( size_t newvalue)
1369 {
1370     JacobiThreadsPerGroup = newvalue;
1371 }
1372 void OpenCLManager::SetRBGSRowsPerThread( size_t newvalue)
1373 {
1374     RBGSRowsPerThread = newvalue;
1375 }
1376 void OpenCLManager::SetRBGSThreadsPerGroup( size_t newvalue)
1377 {
1378     RBGSThreadsPerGroup = newvalue;
1379 }
1380 void OpenCLManager::SetDefectRowsPerThread( size_t newvalue)
1381 {
1382     DefectRowsPerThread = newvalue;
1383 }
1384 void OpenCLManager::SetDefectThreadsPerGroup( size_t newvalue)
1385 {
```

```
1386     FTCRowsPerThread = newvalue;
1387 }
1388 void OpenCLManager::SetFTCThreadsPerGroup( size_t newvalue )
1389 {
1390     FTCThreadsPerGroup = newvalue;
1391 }
1392 void OpenCLManager::SetCTFRowsPerThread( size_t newvalue )
1393 {
1394     CTFRowsPerThread = newvalue;
1395 }
1396 void OpenCLManager::SetCTFThreadsPerGroup( size_t newvalue )
1397 {
1398     CTFThreadsPerGroup = newvalue;
1399 }
1400
1401 void OpenCLManager::WriteError( cl_int err )
1402 {
1403     switch (err)
1404     {
1405         case CL_INVALID_COMMAND_QUEUE:
1406             mexPrintf(" invalid command queue.\n");
1407             break;
1408         case CL_INVALID_KERNEL:
1409             mexPrintf(" invalid kernel.\n");
1410             break;
1411         case CL_INVALID_CONTEXT:
1412             mexPrintf(" invalid context.\n");
1413             break;
1414         case CL_INVALID_KERNEL_ARGS:
1415             mexPrintf(" invalid kernel arguments.\n");
1416             break;
1417         case CL_INVALID_WORK_DIMENSION:
1418             mexPrintf(" invalid work dimension.\n");
1419             break;
1420         case CL_INVALID_WORK_GROUP_SIZE:
1421             mexPrintf(" invalid work group size.\n");
1422             break;
1423         case CL_INVALID_WORK_ITEM_SIZE:
1424             mexPrintf(" invalid work item size.\n");
1425             break;
1426         case CL_INVALID_GLOBAL_OFFSET:
1427             mexPrintf(" invalid global offset.\n");
1428             break;
1429         case CL_OUT_OF_RESOURCES:
1430             mexPrintf(" out of resources.\n");
1431             break;
1432         case CL_MEM_OBJECT_ALLOCATION_FAILURE:
1433             mexPrintf(" failed to allocate memory object.\n");
1434             break;
```

```

1435     case CL_INVALID_EVENT_WAIT_LIST:
1436         mexPrintf(" invalid event wait list.\n");
1437         break;
1438     case CL_OUT_OF_HOST_MEMORY:
1439         mexPrintf(" out of host memory.\n");
1440         break;
1441     }
1442 }
```

10.3.3 OpenCLHigh.h

```

1 #include "preprocessor.h"
2 #include "OpenCLManager.h"
3 #include "MathVector.h"
4 #include <cmath>
5 #include "mex.h"
6
7 #ifndef __OPENCLHIGH_H__
8 #define __OPENCLHIGH_H__
9
10 //Helper functions to simulate templates for c functionality:
11 void HelpNormInf(OpenCLManager * __OpenCLManager__, MathVector<float> * input, MathVector<float> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);
12 void HelpNormInf(OpenCLManager * __OpenCLManager__, MathVector<double> * input, MathVector<double> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);
13 void HelpSum(OpenCLManager * __OpenCLManager__, MathVector<float> * input, MathVector<float> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);
14 void HelpSum(OpenCLManager * __OpenCLManager__, MathVector<double> * input, MathVector<double> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);
15 void HelpNorm2(OpenCLManager * __OpenCLManager__, MathVector<float> * input, MathVector<float> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);
16 void HelpNorm2(OpenCLManager * __OpenCLManager__, MathVector<double> * input, MathVector<double> * output, unsigned int threadsize, unsigned int remLength, cl_event * event);
17
18 //Helper functions for C++ api:
19 template <class T> T NormInf(MathVector<T> & input, unsigned int threadsize, float * time);
20 template <class T> T NormInf(MathVector<T> & input, float * time);
```

```

21 template <class T> T Norm2(MathVector<T> & input , unsigned int
22   threadsize , float * time);
23 template <class T> T Norm2(MathVector<T> & input , float * time
24   );
23 template <class T> T sum(MathVector<T> & input , unsigned int
24   threadsize , float * time);
24 template <class T> T sum(MathVector<T> & input , float * time);
25
26
27 //Norms and sum:
28 template <class T> T NormInf(OpenCLManager * __OpenCLManager__,
29   MathVector<T> & input , unsigned int threadsize , float *
30   time);
29 template <class T> T Norm2(OpenCLManager * __OpenCLManager__,
30   MathVector<T> & input , unsigned int threadsize , float *
31   time);
30 template <class T> T sum(OpenCLManager * __OpenCLManager__,
31   MathVector<T> & input , unsigned int threadsize , float *
32   time);
32
33
34
35
36 //
37 //HELPER FUNCTIONS:
38 //
39 void HelpNormInf(OpenCLManager * __OpenCLManager__, MathVector
40   <float> * input , MathVector<float> * output , unsigned int
41   threadsize , unsigned int remLength , cl_event * event)
42 {
43   __OpenCLManager__->NormInfF(input->GetDataBuffer() , output->
44     GetDataBuffer() , threadsize , remLength , event);
45 }
43 void HelpNormInf(OpenCLManager * __OpenCLManager__ , MathVector
44   <double> * input , MathVector<double> * output , unsigned
45   int threadsize , unsigned int remLength , cl_event * event)
46 {
47   __OpenCLManager__->NormInfD(input->GetDataBuffer() , output->
48     GetDataBuffer() , threadsize , remLength , event);
49 }
47 void HelpSum(OpenCLManager * __OpenCLManager__ , MathVector<
50   float> * input , MathVector<float> * output , unsigned int
50   threadsize , unsigned int remLength , cl_event * event)
51 {
52   __OpenCLManager__->ParallelSumReductionF(input->
53     GetDataBuffer() , output->GetDataBuffer() , threadsize ,
54     remLength , event);
55 }
```

```

51 void HelpSum(OpenCLManager * __OpenCLManager__, MathVector<
52     double> * input, MathVector<double> * output, unsigned int
53     threadsize, unsigned int remLength, cl_event * event)
54 {
55     __OpenCLManager__->ParallelSumReductionD(input->
56         GetDataBuffer(), output->GetDataBuffer(), threadsize,
57         remLength, event);
58 }
59 void HelpNorm2(OpenCLManager * __OpenCLManager__, MathVector<
60     float> * input, MathVector<float> * output, unsigned int
61     threadsize, unsigned int remLength, cl_event * event)
62 {
63     __OpenCLManager__->Norm2F(input->GetDataBuffer(), output->
64         GetDataBuffer(), threadsize, remLength, event);
65 }
66 //C++ API:
67 //
68 //
69 //NORMS:
70 //
71 //
72 template <class T> T NormInf(OpenCLManager * __OpenCLManager__
73     , MathVector<T> & input, unsigned int threadsize, float *
74     time)
75 {
76     *time = 0.0f;
77     cl_event event;
78     //Declare variables:
79     T result = 0.0;
80     unsigned int numIter = 0;
81     unsigned int remWork = input.GetLength();
82     //Find the number of iterations required:
83     while (remWork > 1)
84     {
85         if (remWork % (threadsize*threadsize) == 0)
86             remWork /= (threadsize*threadsize);
87         else
88             remWork = remWork/(threadsize*threadsize)+1;
89         numIter++;

```

```

88     }
89     MathVector<T> * pInput, * pOut;
90     pInput = &input;
91
92     remWork = input.GetLength();
93     unsigned int remLength = remWork;
94     for (unsigned int i = 0; i < numIter; i += 1)
95     {
96         //Calculate the remaining work:
97         if (remWork % (threadsize*threadsize) == 0)
98             remWork /= (threadsize*threadsize);
99         else
100            remWork = remWork/(threadsize*threadsize)+1;
101        //if at last iteration, create array for storage,
102        //otherwise keep on GPU:
103        if (remWork == 1)
104            pOut = new MathVector<T>(__OpenCLManager__, remWork, (T*)
105                                      NULL);
106        else
107            pOut = new MathVector<T>(__OpenCLManager__, remWork);
108
109        //Call the parallel sum reduction routine:
110        HelpNormInf(__OpenCLManager__, pInput, pOut, threadsize,
111                    remLength, &event);
112        remLength = remWork;
113        *time += __OpenCLManager__->GetExecutionTime(&event);
114
115        //Reset pointers:
116        if (pInput != &input)
117        {
118            delete pInput;
119        }
120        pInput = pOut;
121
122        //If at last iteration, swap buffers and delete temp
123        //buffer:
124        if (remWork == 1)
125        {
126            __OpenCLManager__->SwapGPUBufferData(pOut->GetDataBuffer()
127                                                    (), pOut->GetData(), remWork, sizeof(T));
128            result = *(pOut->GetData());
129            delete pOut;
130        }
131    }
132
133    return result;
134 }
```

```

131 template <class T> T Norm2(OpenCLManager * __OpenCLManager__,
132                           MathVector<T> & input , unsigned int threadsize , float *
133                           time)
134 {
135   //Declare variables:
136   *time = 0.0f;
137   cl_event event;
138   T result = 0.0;
139   unsigned int numIter = 0;
140   unsigned int remWork = input .GetLength () ;
141   //Find the number of iterations required:
142   while (remWork > 1)
143   {
144     if (remWork % (threadsize*threadsize) == 0)
145       remWork /= (threadsize*threadsize);
146     else
147       remWork = remWork/(threadsize*threadsize)+1;
148     numIter++;
149   }
150   MathVector<T> * pInput , * pOut;
151   pInput = &input ;
152   remWork = input .GetLength ();
153   unsigned int remLength = remWork;
154   for (unsigned int i = 0; i < numIter; i += 1)
155   {
156     //Calculate the remaining work:
157     if (remWork % (threadsize*threadsize) == 0)
158       remWork /= (threadsize*threadsize);
159     else
160       remWork = remWork/(threadsize*threadsize)+1;
161     //if at last iteration , create array for storage ,
162     //otherwise keep on GPU:
163     if (remWork == 1)
164       pOut = new MathVector<T>(__OpenCLManager__ , remWork , (T
165                               *)NULL);
166     else
167       pOut = new MathVector<T>(__OpenCLManager__ , remWork);
168     //Call the parallel sum reduction routine:
169     if (i ==0)
170       HelpNorm2(__OpenCLManager__ , pInput , pOut , threadsize ,
171                  remLength , &event);
172     else
173       HelpSum(__OpenCLManager__ , pInput , pOut , threadsize ,
174                  remLength , &event);
175   remLength = remWork;

```

```

174     *time += __OpenCLManager__->GetExecutionTime(&event);
175
176     //Reset pointers:
177     if (pInput != &input)
178     {
179         delete pInput;
180     }
181     pInput = pOut;
182
183     //If at last iteration , swap buffers and delete temp
184     //buffer:
185     if (remWork == 1)
186     {
187         __OpenCLManager__->SwapGPUBufferData(pOut->GetDataBuffer()
188             () , pOut->GetData() , remWork , sizeof(T));
189         result = *(pOut->GetData());
190         delete pOut;
191     }
192     return sqrt(result);
193 }
194 template <class T> T sum(OpenCLManager * __OpenCLManager__,
195     MathVector<T> & input , unsigned int threadsize , float *
196     time)
197 {
198     *time = 0.0f;
199     cl_event event;
200     //Declare variables:
201     T result = 0.0;
202     unsigned int numIter = 0;
203     unsigned int remWork = input.GetLength();
204     //Find the number of iterations required:
205     while (remWork > 1)
206     {
207         if (remWork % (threadsize*threadsize) == 0)
208             remWork /= (threadsize*threadsize);
209         else
210             remWork = remWork/(threadsize*threadsize)+1;
211         numIter++;
212     }
213     MathVector<T> * pInput , * pOut;
214     pInput = &input;
215     remWork = input.GetLength();
216     unsigned int remLength = remWork;
217     for (unsigned int i = 0; i < numIter ; i += 1)
218     {
219         //Calculate the remaining work:
220         if (remWork % (threadsize*threadsize) == 0)

```

```

219     remWork /= (threadsize*threadsize);
220 else
221     remWork = remWork/(threadsize*threadsize)+1;
222
223 //if at last iteration , create array for storage ,
224 //otherwise keep on GPU:
224 if (remWork == 1)
225     pOut = new MathVector<T>(__OpenCLManager__, remWork, (T
226     *)NULL);
227 else
228     pOut = new MathVector<T>(__OpenCLManager__, remWork);
229
230 //Call the parallel sum reduction routine:
231 HelpSum(__OpenCLManager__, (MathVector<T> *)pInput, (
232     MathVector<T> *)pOut, threadsize, remLength, &event);
233 remLength = remWork;
234 *time += __OpenCLManager__->GetExecutionTime(&event);
235
236 //Reset pointers:
237 if (pInput != &input)
238 {
239     delete pInput;
240 }
241 pInput = pOut;
242
243 //If at last iteration , swap buffers and delete temp
244 //buffer:
245 if (remWork == 1)
246 {
247     __OpenCLManager__->SwapGPUBufferData(pOut->GetDataBuffer
248     () , pOut->GetData() , remWork, sizeof(T));
249     result = *(pOut->GetData());
250     delete pOut;
251 }
252
253
254 #endif

```

10.3.4 MemoryControl.h

```

1 #include "preprocessor.h"
2 #include <vector>

```

```
3  
4 #ifndef __MEMORYCONTROL_H__  
5 #define __MEMORYCONTROL_H__  
6  
7 template <class T>  
8 struct __MemoryControl__  
9 {  
10 public:  
11     T * data;  
12     cl_mem buffer;  
13     unsigned int size;  
14     int RefCount;  
15     bool BuffersMatch;  
16 };  
17  
18 struct __IndexControl__  
19 {  
20 public:  
21     unsigned int * data;  
22     cl_mem buffer;  
23     unsigned int size;  
24     int RefCount;  
25     bool BuffersMatch;  
26 };  
27  
28 #endif /* __MEMORYCONTROL_H__ */
```

10.3.5 MathVector.h

```
1 #ifndef __MATHVECTOR_H__  
2 #define __MATHVECTOR_H__  
3  
4 #include "MemoryControl.h"  
5 #include "preprocessor.h"  
6  
7 class ClassController  
8 {  
9     public:  
10     int Type;  
11 };  
12  
13 template <class T>  
14 class MathVector : public ClassController  
15 {  
16     public:  
17     MathVector ();
```

```

18     MathVector (OpenCLManager *, unsigned int);
19     MathVector (OpenCLManager *, unsigned int, T * real);
20     void Init(OpenCLManager *, unsigned int);
21     void Init(OpenCLManager *, unsigned int, T * real);
22     ~MathVector ();
23     cl_mem & GetDataBuffer();
24     unsigned int GetLength();
25     void SyncBuffers();
26     T * GetData();
27     __MemoryControl__<T> * GetMemoryControl();
28     void SetMemoryControl(__MemoryControl__<T> *);
29 //operators:
30     MathVector * operator*(T element);
31     MathVector * operator+(MathVector<T> & other);
32 private:
33     OpenCLManager * __OpenCLManager__;
34     unsigned int length;
35     __MemoryControl__<T> * memory;
36 };
37
38
39 #include "OpenCLHigh.h"
40 #include "OpenCLManager.h"
41
42
43
44 /**
45 //Constructors:
46 /**
47 template <class T>
48 MathVector<T>::MathVector ()
49 {
50 }
51 template <class T>
52 MathVector<T>::MathVector (OpenCLManager * __OpenCLManager__,
53                           unsigned int len)
54 {
55     Init(__OpenCLManager__, len);
56 }
57 template <class T>
58 MathVector<T>::MathVector (OpenCLManager * __OpenCLManager__,
59                           unsigned int len, T * real)
60 {
61     Init(__OpenCLManager__, len, real);
62 }
63 void MathVector<T>::Init (OpenCLManager * __OpenCLManager__,
64                           unsigned int len)
65 {

```

```

64     length = len ;
65     memory = __OpenCLManager__->AllocateMemory((T*)NULL, len) ;
66     this->__OpenCLManager__ = __OpenCLManager__ ;
67 }
68 template <class T>
69 void MathVector<T>::Init (OpenCLManager * __OpenCLManager__ ,
70                           unsigned int len , T * real)
71 {
72     length = len ;
73     memory = __OpenCLManager__->AllocateMemory( real , len ) ;
74     this->__OpenCLManager__ = __OpenCLManager__ ;
75 }
76 /**
77 //Destructors :
78 /**
79 template <class T>
80 MathVector<T>::~MathVector ()
81 {
82     __OpenCLManager__->DeleteMemory( memory ) ;
83 }
84 /**
85 /**
86 //Methods :
87 /**
88 template <class T>
89 cl_mem & MathVector<T>::GetDataBuffer()
90 {
91     return memory->buffer ;
92 }
93 template <class T>
94 unsigned int MathVector<T>::GetLength()
95 {
96     return length ;
97 }
98 template <class T>
99 T * MathVector<T>::GetData()
100 {
101     return memory->data ;
102 }
103 /**
104 //Buffer sync'ing :
105 /**
106 /**
107 template <class T>
108 void MathVector<T>::SyncBuffers()
109 {
110     __OpenCLManager__->SwapGPUBufferData( memory->buffer , memory
111                                         ->data , length , sizeof(T)) ; /*
```

```

111     if (memory->BuffersMatch == false) //obsolete code from c
112     {
113         memory->BuffersMatch = true;
114     } */
116 }
117
118 template <class T>
119 __MemoryControl__<T> * MathVector<T>::GetMemoryControl()
120 {
121     return memory;
122 }
123
124 template <class T>
125 void MathVector<T>::SetMemoryControl(__MemoryControl__<T> *
126 newMem)
127 {
128     memory = newMem;
129 }
130 #endif /* __MATHVECTOR_H__ */
```

10.3.6 Matrix.h

```

1 #include "MemoryControl.h"
2 #include "preprocessor.h"
3 #include "MathVector.h"
4
5 #ifndef __MATRIX_H__
6 #define __MATRIX_H__
7
8
9 template <class T>
10 class Matrix : public MathVector<T>
11 {
12     public:
13         Matrix(OpenCLManager *, unsigned int, unsigned int);
14         Matrix(OpenCLManager *, unsigned int, unsigned int, T *
15                 real);
15         void InitMatrix(OpenCLManager *, unsigned int, unsigned
16                         int);
16         void InitMatrix(OpenCLManager *, unsigned int, unsigned
17                         int, T * real);
17         ~Matrix();
18         unsigned int GetWidth();
```

```

19     unsigned int GetHeight();
20 private:
21     unsigned int width, height;
22 };
23
24
25 /**
26 // Constructors:
27 /**
28 template <class T>
29 Matrix<T>::Matrix (OpenCLManager * __OpenCLManager__, unsigned
30     int width, unsigned int height)
31 {
32     InitMatrix(__OpenCLManager__, width, height);
33 }
34 template <class T>
35 Matrix<T>::Matrix (OpenCLManager * __OpenCLManager__, unsigned
36     int width, unsigned int height, T * real)
37 {
38     InitMatrix(__OpenCLManager__, width, height);
39 void Matrix<T>::InitMatrix (OpenCLManager * __OpenCLManager__,
40     unsigned int width, unsigned int height)
41 {
42     this->width = width;
43     this->height = height;
44     Init(__OpenCLManager__, width*height,(T *)NULL);
45 }
46 template <class T>
47 void Matrix<T>::InitMatrix (OpenCLManager * __OpenCLManager__,
48     unsigned int width, unsigned int height, T * real)
49 {
50     this->width = width;
51     this->height = height;
52     Init(__OpenCLManager__, width*height, real);
53 /**
54 // Getters:
55 /**
56 template <class T>
57 unsigned int Matrix<T>::GetWidth()
58 {
59     return width;
60 }
61 template <class T>
62 unsigned int Matrix<T>::GetHeight()
63 {

```

```

64     return height;
65 }
66
67 /**
68 //Destructor:
69 /**
70 template <class T>
71 Matrix<T>::~Matrix()
72 {
73 }
74
75 #endif /* __MATRIX_H__ */

```

10.3.7 BandMatrix.h

```

1 #include "preprocessor.h"
2 #include "MemoryControl.h"
3
4 #ifndef BANDMATRIX_H
5 #define BANDMATRIX_H
6 template <class T>
7 class BandMatrix : public MathVector<T>
8 {
9     public:
10     BandMatrix(OpenCLManager * __OpenCLManager__, unsigned int
11                 height, unsigned int width, unsigned int bandwidth, T
12                 * data);
13     BandMatrix(OpenCLManager * __OpenCLManager__, unsigned int
14                 height, unsigned int width, unsigned int bandwidth);
15     ~BandMatrix();
16     void InitMatrix(OpenCLManager * __OpenCLManager__,
17                     unsigned int height, unsigned int width, unsigned int
18                     bandwidth, T * data);
19     unsigned int GetBandwidth();
20     unsigned int GetHeight();
21     unsigned int GetWidth();
22     private:
23     unsigned int height;
24     unsigned int width;
25     unsigned int bandwidth;
26     __MemoryControl__<T> * memory;
27     void SortVector( T * & data, unsigned int length, unsigned
28                     int bandwidth);
29 };
30
31 template <class T>

```

```

26 BandMatrix<T>::BandMatrix(OpenCLManager * __OpenCLManager__,
27     unsigned int height, unsigned int width, unsigned int
28     bandwidth, T * data)
29 {
30     InitMatrix(__OpenCLManager__, height, width, bandwidth, data
31 );
32 }
33
34 template <class T>
35 void BandMatrix<T>::InitMatrix(OpenCLManager *
36     __OpenCLManager__, unsigned int height, unsigned int width
37     , unsigned int bandwidth, T * data)
38 {
39     this->height = height;
40     this->width = width;
41     this->bandwidth = bandwidth;
42
43     //determine length:
44     SortVector(data, height, bandwidth);
45     Init(__OpenCLManager__, height*(1+2*bandwidth), data);
46 }
47
48 template <class T>
49 unsigned int BandMatrix<T>::GetWidth()
50 {
51     return width;
52 }
53
54 template <class T>
55 unsigned int BandMatrix<T>::GetHeight()
56 {
57     return height;
58 }
59 template <class T>
60 void BandMatrix<T>::SortVector( T * & data, unsigned int
61     length, unsigned int bandwidth)
62 {
63     T * newData = (T*)mxMalloc( sizeof(T)*length*(1 + 2*bandwidth
64     ));
65
66     unsigned int index = 0;
67     int low, high;
68     for (int i = 0; i < length; i += 1)
69     {

```

```

68     for (int j = 0; j < 1+2*bandwidth; j += 1)
69     {
70         newData[i*(1+2*bandwidth) + j] = data[i + j*length];
71     }
72 }
73 data = newData;
74 }
75 #endif

```

10.3.8 SparseMatrix.h

```

1 #include "preprocessor.h"
2 #include "MemoryControl.h"
3 #include "MathVector.h"
4
5 #ifndef SPARSEMATRIX_H
6 #define SPARSEMATRIX_H
7 template <class T>
8 class SparseMatrix : public MathVector<T>
9 {
10     public:
11         SparseMatrix(OpenCLManager * __OpenCLManager__, unsigned
12                     int height, unsigned int width, unsigned int length,
13                     size_t * rows, size_t * columns, T * data);
14         ~SparseMatrix ();
15         void InitMatrix(OpenCLManager * __OpenCLManager__,
16                         unsigned int height, unsigned int width, unsigned int
17                         length, size_t * rows, size_t * columns, T * data);
18         int * GetColumnIndexes();
19         int * GetRowIndexes();
20         unsigned int GetHeight();
21         unsigned int GetWidth();
22         unsigned int GetRowVectorLength();
23         cl_mem & GetRowIndexBuffer();
24         cl_mem & GetColumnIndexBuffer();
25         __IndexControl__ * GetRowControl();
26         __IndexControl__ * GetColumnControl();
27     private:
28         unsigned int height;
29         unsigned int width;
30         __IndexControl__ * column;
31         __IndexControl__ * row;
32         unsigned int SortVectors(unsigned int length, unsigned * &
33                                 rows, unsigned * & columns, T * & data);
34 };
35 template <class T>

```

```

31 SparseMatrix<T>::SparseMatrix(OpenCLManager *
32     __OpenCLManager__, unsigned int height, unsigned int width
33     , unsigned int length, size_t * rows, size_t * columns, T
34     * data)
35 {
36     InitMatrix(__OpenCLManager__, height, width, length, rows,
37     columns, data);
38 }
39 template <class T>
40 void SparseMatrix<T>::InitMatrix(OpenCLManager *
41     __OpenCLManager__, unsigned int height, unsigned int width
42     , unsigned int length, size_t * rows, size_t * columns, T
43     * data)
44 {
45     this->height = height;
46     this->width = width;
47     unsigned int * columnsNew = (unsigned int *)mxMalloc(sizeof(
48         unsigned int)*length);
49     unsigned int * rowsNew = (unsigned int *)mxMalloc(sizeof(
50         unsigned int)*length);
51     unsigned int currentCol = 0;
52     unsigned int index = 1;
53     unsigned int low = columns[0];
54     unsigned int high = columns[1];
55     while (true)
56     {
57         for (unsigned int i = low; i < high; i++)
58         {
59             columnsNew[i] = currentCol;
60             rowsNew[i] = rows[i];
61         }
62         if (high >= length)
63             break;
64         currentCol++;
65         low = high;
66         index++;
67         high = columns[index];
68     }
69     unsigned int rowLength = SortVectors(length,rowsNew,
70     columnsNew,data);
71     Init(__OpenCLManager__, length, data);
72     column = __OpenCLManager__->AllocateIndex(columnsNew, length
73         );
74     row = __OpenCLManager__->AllocateIndex(rowsNew, rowLength+1)
75         ;
76 }
77 template <class T>
78 unsigned int SparseMatrix<T>::GetWidth()
79 {

```

```
68     return width;
69 }
70 template <class T>
71 unsigned int SparseMatrix<T>::GetHeight()
72 {
73     return height;
74 }
75 template <class T>
76 int * SparseMatrix<T>::GetColumnIndexes()
77 {
78     return column->data;
79 }
80 template <class T>
81 int * SparseMatrix<T>::GetRowIndexes()
82 {
83     return row->data;
84 }
85 template <class T>
86 unsigned int SparseMatrix<T>::SortVectors(unsigned int length,
87                                         unsigned * & rows, unsigned * & columns, T * & data)
87 {
88     int i = 0;
89     int temp;
90     T tempData;
91     //Sort arrays:
92     while (i < length - 1)
93     {
94         if ((rows[i] > rows[i + 1]) || ((rows[i] == rows[i + 1]) && (
95             columns[i] > columns[i + 1])))
96         {
97             temp = rows[i + 1];
98             rows[i + 1] = rows[i];
99             rows[i] = temp;
100            temp = columns[i + 1];
101            columns[i + 1] = columns[i];
102            columns[i] = temp;
103            tempData = data[i + 1];
104            data[i + 1] = data[i];
105            data[i] = tempData;
106            if (i > 0)
107            {
108                i--;
109            }
110        }
111    }
112    i++;
113 }
114 }
```

```
115     unsigned int rowLength=height ;
116
117     //Declare new array:
118     unsigned int * newRows = ( unsigned int *)mxMalloc( sizeof(
119                               unsigned int)*(rowLength+1));
120
121     unsigned int currentcount = 0;
122     newRows[0] = 0;
123     for ( i = 0; i < rowLength; i++)
124     {
125         while( currentcount < length)
126         {
127             if ( rows[ currentcount]==i )
128             {
129                 currentcount++;
130             }
131             else
132             {
133                 newRows[ i+1] = currentcount ;
134                 break ;
135             }
136             if ( currentcount == length)
137             {
138                 newRows[ i+1] = length ;
139             }
140     }
141
142     rows = newRows;
143     return rowLength;
144 }
145 template <class T>
146 cl_mem & SparseMatrix<T>::GetColumnIndexBuffer()
147 {
148     return column->buffer ;
149 }
150 template <class T>
151 cl_mem & SparseMatrix<T>::GetRowIndexBuffer()
152 {
153     return row->buffer ;
154 }
155
156 template <class T>
157 __IndexControl__ * SparseMatrix<T>::GetRowControl()
158 {
159     return row ;
160 }
161
162 template <class T>
```

```
163     _IndexControl__ * SparseMatrix<T>::GetColumnControl()
164 {
165     return column;
166 }
167 template <class T>
168 unsigned int SparseMatrix<T>::GetRowVectorLength()
169 {
170     return row->size;
171 }
172
173 #endif
```

10.3.9 preprocessor.h

```
1 #ifndef PREPROCESSOR_H
2 #define PREPROCESSOR_H
3
4 #define __DEBUG__
5 #define __PROGRAM_BUILD_INFO__ "build_info.txt"
6 //#pragma OPENCL EXTENSION cl_khr_fp64 : enable
7 //#define USE_DOUBLE_PRECISION
8
9 //Define standard libraries used:
10 #include <string>
11 #include <iostream>
12 #include <vector>
13 #ifdef MAC
14 #include <OpenCL/cl.h>
15 #else
16 #include <CL/cl.h>
17 #endif
18
19
20 //Define commands for handling errors:
21 #ifdef __DEBUG__
22
23 #endif
24
25
26 #endif
```