University of Tartu
Faculty of Mathematics and Computer Science
Institute of Computer Science

Kaupo Kuresson

Sparse Matrix Algorithms Using GPGPU

Bachelor's Thesis

Supervisor: Eero Vainikko

Author: ..... "....." May 2012
Supervisor: ..... "....." May 2012

Approved for defence:
Professor: .... "....." May 2012

Tartu 2012

Table of Contents

I Introduction

Matrix-vector multiplication is a part of a variety of applications in scientific and economic modeling, machine learning and signal processing. Quite common is the fact that the matrix used in these calculations is sparse. This means the matrix is primarily populated with zeros (See Figure 1). Sparse matrix-vector multiplication has been called one of the "Seven dwarfs" of high performance computing. The "dwarfs" are key algorithmic kernels in many scientific computing applications. These are dense linear algebra, sparse linear algebra, spectral methods, N-body methods, structured grids, unstructured grids and Monte Carlo [9].

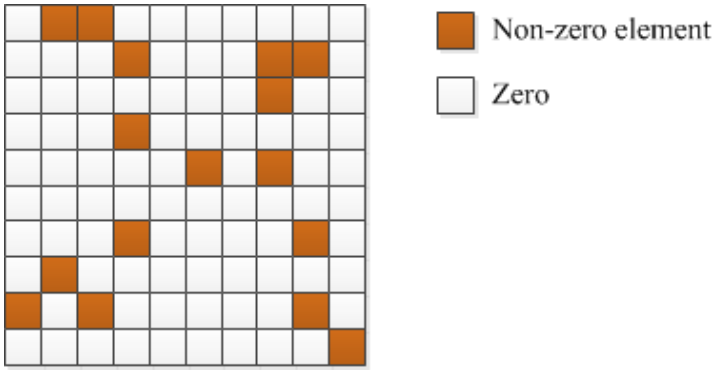

Figure 1 - An example of a sparse matrix

Since the zeros in a sparse matrix do not change the product when multiplying with a vector, different formats have been proposed to eliminate the need to store and process them. For example the coordinate (COO) format, where non-zero elements and their coordinates are stored. (See Figure 2) However, it is not always clear, which of these representations will yield the best result with a given matrix [10].

The purpose of this thesis is to benchmark and compare these different representations and algorithms for multiplication. Also, to see the performance differences of running the algorithms on CPUs (Central Processing Unit) and GPUs (Graphics Processing Unit) using GPGPU (General-Purpose Computing on Graphics Processing Units).

5	0	2
0	1	0
6	0	3

values	5	6	1	2	3
column index	0	0	1	2	2
row index	0	2	1	0	2

Figure 2 - A matrix (on the left) in the coordinate format (on the right).

Processors are mostly oriented towards latency - for reducing the time it takes to execute a single command. GPUs on the other hand, for throughput by being highly parallel using a

large amount of simpler cores. To efficiently use this computing potential, we need to divide the problem into smaller tasks. For example, in the case of multiplying a matrix and a vector, we can find each element of the resulting vector individually.

If A is a Matrix and b a vector, then the product of A and b:

$$c_i = \sum_{j=1}^{M} A_{ij} b_j$$

Each element of c can be computed independently.

OpenCL is an open, royalty-free standard for parallel programming of heterogeneous systems. Initially developed by Apple, the final technical specification was refined in conjunction with representatives from AMD, Intel, IBM and Nvidia before it was submitted to the Khronos Group [3]. The first public release of OpenCL was approved in December 2008.

OpenCL can make use of all available processing entities - CPUs, GPUs and other processors [2]. Work is divided into a N-dimensional computation domain and then a kernel is executed in parallel at each point in that domain. For example a 1024x1024 2D image could be divided so that a pixel is a single work-item in the computational domain. These work items can also be grouped into workgroups, which execute together and can synchronize with each other. Synchronization outside of a work group is not possible [4]. (See Figure 3)

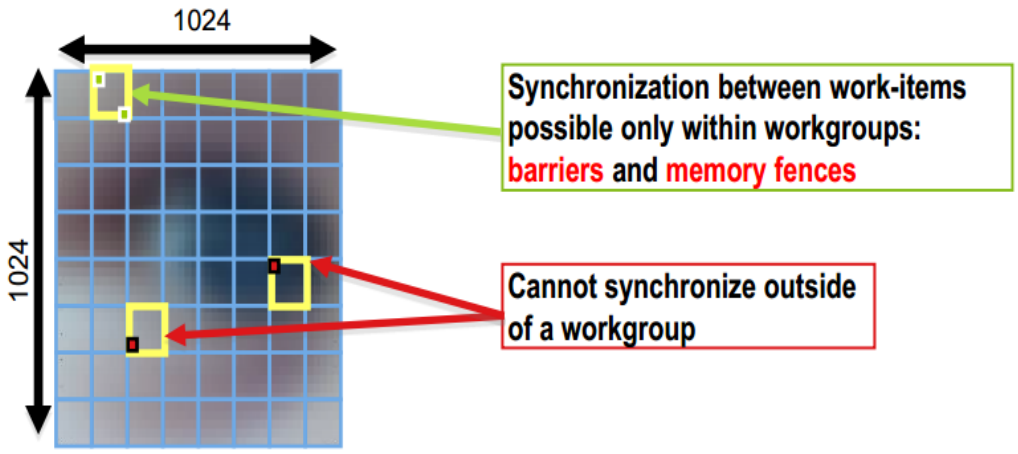

Figure 3 - Example of division into work-items and work-groups [4].

An OpenCL kernel is the code for a work item, which is basically a C function. The language for writing these kernels is derived from the C99 standard [4].

OpenCL is an alternative to using Nvidia's CUDA architecture, which can only be used with Nvidia GPUs [6]. OpenCL is still not as popular as CUDA and the lack of maturity has resulted in lower performance when compared with CUDA. However, a recent benchmark run by Kyle Spafford, from the Future Technology Group at Oak Ridge

National Lab (ORNL), shows that OpenCL can match CUDA performance on most of the basic math kernels [5] (See Figure 4).

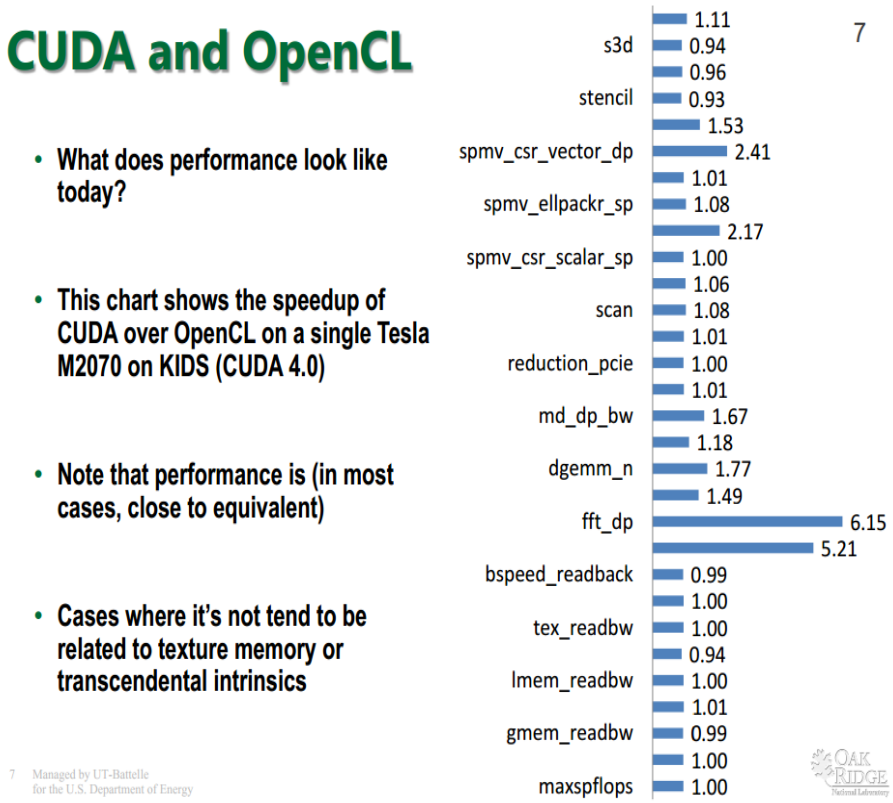

Figure 4 - Comparison of performance between OpenCL and CUDA [7].

AMD, who is also the most vocal booster of OpenCL technology, has posted a case study on diagonal sparse matrix vector multiplication. It offers OpenCL kernel examples and ideas like optimizing the algorithm with memory access patterns or representing the vector as an image. Using OpenCL images can provide additional memory bandwidth by using the texture caches on GPUs and the texture sampling hardware for reading from them [8].

The next chapter will give an overview of the different storage formats, kernels written as a part of this thesis, their performance, characteristics and the hardware used for the tests.

II Hardware and Testing

This chapter will give an overview of the hardware and methods used for performance measurements. A guide on how to set-up an OpenCL project can be found in Appendix 2.

1 Personal Computer

First, the tests will be run on my desktop computer. Its hardware specifications can be seen in the following tables (Tables 1-3).

Hardware platform	1 AMD Phenom II X4 CPU 1 ATI HD5870 GPU
CPU cores and speed	4 @ 3.5 GHz
Main memory (DRAM)	4 GB (+1 GB video)

Table 1 - Hardware of PC used in testing

Information about the platform and devices can also be requested through OpenCL API (See Tables 2 and 3). This can be useful for finding suitable devices for a given task.

A compute unit is a part of the hardware that executes work-groups, which is a collection of work-items. Each compute unit can have more than 1 streaming processors. For example, the ATI Radeon HD5870 has 20 compute units, but a total of 1600 stream processors. Max clock frequency is the maximum configured clock frequency of the device in MHz.

Device name	Cypress
Device vendor	Advanced Micro Devices, Inc.
Device version	OpenCL 1.1 AMD-APP (851.4)
Driver version	CAL 1.4.1664 (VM)
Device max compute units	20
Device max clock frequency	850
Device global memory size	1073741824

Table 2 - Information gathered via OpenCL API about the GPU

Device name	AMD Phenom(tm) II X4 20 Processor
Device vendor	AuthenticAMD
Device version	OpenCL 1.1 AMD-APP (851.4)
Driver version	2.0
Device max compute units	4
Device max clock frequency	3515
Device global memory size	4153450496

Table 3 - Information gathered via OpenCL API about the CPU

2 EENet

Tests will also be run on EENet (the Estonian Education and Research Network) worknodes. These include nVidia Tesla S1070 Computing Systems, each with four Tesla T10 computing processors, which connect to host systems via PCI Express cables. They are targeted as a high-performance computing (HPC) solution [14]. In EENet, each worknode is connected to two nVidia Tesla T10 GPUs (See Table 4). Some tests will also be run to evaluate the benefit of using multiple GPUs simultaneously.

EENet resources can be accessed through Grid computing services. The XRSL (Extended Resource Specification Language) file used for describing and submitting the jobs is included in Appendix 1. (Item 18 – KK_0.xrsl)

Device name	Tesla T10 Processor
Device vendor	NVIDIA Corporation
Device version	OpenCL 1.0 CUDA
Driver version	285.05.23
Device max compute units	30
Device max clock frequency	1440
Device global memory size	4294770688

Table 4 - Information gathered via OpenCL API about the GPU (EENet)

3 Testing

To test the performance of the different storage formats and algorithms I will be using matrices from The University of Florida Sparse Matrix Collection. See Table 5 for a list of all matrixes used. This collection is a set of sparse matrices that arise in real applications. For example „psmigr_1“ describes migrations between countries and „viscoplastic1“ a finite-element method discretization of a viscoplastic collision problem. This is useful, because performance results with artificially-generated matrices can be misleading [11]. Vectors used in the tests are dense vectors generated with MatLab (See Appendix 1 – item 17).

Algorithm execution times will be measured. Each algorithm will be run 1000 times and the total time is then divided by 1000. This will result in a more reliable average. In the measurements, single precision floating-point numbers are used.

To better evaluate the storage methods, they will be given a rating based on how many non-zero elements per second they can process.

The initialization of OpenCL will not be included in these timings. In a real world situation, the environment can be set up once and then reused for a large number of calculations.

#	Name	Size (rows x columns)	Number of non-zero elements	Original problem
1	heart1	3,557 x 3,557	1,387,773	2D/3D
2	Trefethen_20000b	19,999 x 19,999	287,217	Combinatorial
3	TSOPF_RS_b162_c3	15,374 x 15,374	610,299	Power network
4	HB/psmigr_1	3,140 x 3,140	543,162	Economic
5	Oberwolfach/spiral	1,434 x 1,434	9,831	Model reduction
6	viscoplastic1	4,326 x 4,326	61,166	Materials
7	Meszaros/deter4	3,235 x 9,133	19,231	Linear programming
8	NYPA/Maragal_2	555 x 350	4,357	Least squares
9	MKS/fp	7,548 x 7,548	834,222	Electromagnetics

Table 5 - Matrices used in the tests. From The University of Florida Sparse Matrix Collection [11].

III Storage Formats

This chapter will describe the different matrix storage formats used in the tests, how they affect the calculations and benchmark results.

4 Full Matrix Storage Format

4.1 Description

Matrices used for the tests are first imported from the Matrix Market coordinate format (See Figure 4) as a full representation and then converted to other storage formats [12].

All of the conversion methods written for this thesis can be found in Appendix 1 – item 2 – „matrix_operations.cpp/.hpp“. There are already libraries available for doing this, but the programming helped to understand the structure of each format.

%% Header	
% Comments	
%	M - Number of rows in the matrix
M N L	N - Number of columns in the matrix
I1 J1 A(I1,J1)	L - Number of non-zero elements
I2 J2 A(I2,J2)	I,J indices
I3 J3 A(I3,J3)	A - value at that index
…	
IL JL A(IL,JL)	

Figure 4 – Matrix market format example

In the full representation form all of the elements are stored, regardless if they are non-zeros or zeros. This is very inefficient. The sample matrix used in describing this and other storage formats can be seen on Figure 5.

The storage requirement for a N x M matrix would be:

$$SR = N \times M \times ValueSize$$

(The number of elements in the matrix times the size of the values stored in it.)

0	0	3	1	0
2	0	0	4	0
0	0	2	0	0
0	3	0	1	0
4	0	2	0	3

Figure 5 - A sample 5x5 sparse matrix that will be used with the examples.

We store the matrix in a 1-Dimensional array, because we do not want the overhead of creating an exact structural representation (See Figure 6). The index of an element can easily be calculated by knowing the number of rows and columns in the original matrix.

M = | 0 | 0 | 3 | 1 | 0 | 2 | 0 | 0 | 4 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 3 | 0 | 1 | 0 | 4 | 0 | 2 | 0 | 3 |

numRows = 5

numCols = 5

Figure 6 - A full representation of a sparse matrix.

4.2 Calculation

The calculation for such a representation is almost a 1:1 copy of the expression:

$$c_i = \sum_{j=1}^{M} A_{ij} b_j$$

This means we look through all of the elements in the original matrix. In C++ this can be achieved by using loops, example code is shown next.

The following pseudocode function 'findProduct' (See Figure 7) finds the product of a matrix and vector and returns the resulting vector. Table 6 describes the variables used in this example.

Variable	Description
matrix	The representation of the matrix as an 1-D array
v	vector
rowStart	Index in the array, where the current row starts
rows	Number of rows in the original matrix
cols	Number of columns in the original matrix
answer	The product of matrix and v

Table 6 – Variables used in the C++ example.

```
float * function findProduct
for (i = 0; i < rows ; i++) {
        int rowStart = i*cols;
        for (j = 0; j < cols  ; j++) {
                answer[i] = answer[i] + matrix[rowStart+j] * v[j];
        }
 }
 return answer;
```

Figure 7 – C++ code example.

The same operation as an OpenCL kernel can be seen on Figure 8:

```
__kernel
void basic_opencl(__global float *matrix, __global float *v, __const int rows,
__const int cols, __global float *answer) {
  int row = get_global_id(0);
  float accumulator = 0;
  int rowStart = row * cols;

  for(int col = 0; col < cols; col++) {
     accumulator += matrix[rowStart + col] * v[col];
  }
  answer[row] = accumulator;
}
```

Figure 8 - Example of an OpenCL kernel.

For the rest of the kernels and host code see Appendix 1.

4.3 Performance

Table 7 shows the advantage of simply using OpenCL to parallelize matrix vector multiplication when using the full storage format. The sequential C++ code uses just 1 core of the CPU. OpenCL can help us utilize all available computing power by using multiple threads. Figure 9 helps to illustrate the speedup of using OpenCL.

Matrix	**Execution time (ms) - Sequential, C++ loop**	**Execution time (ms) - OpenCL, CPU**	**Execution time (ms) - OpenCL, GPU**	**Speedup of OpenCL vs sequential (CPU, Phenom II x4)**	**Speedup of OpenCL vs sequential (GPU, HD5870)**
heart1	69	38	20	1,81x	3,45x
Trefethen_20000b	2218	1360	604	1,63x	3,67x
TSOPF_RS_b162_c3	1333	739	359	1,80x	3,71x
psmigr_1	53	30	15	1,77x	3,53x
spiral	11	7	3	1,57x	3,67x
viscoplastic1	102	55	29	1,86x	3,52x
deter4	161	82	45	1,96x	3,58x
Maragal_2	1,1	0,7	0,3	1,64x	3,55x
fp	310	145	87	2,14x	3,56x
		Average speedup:		**1.80x**	**3.58x**

Table 7 - performance using the full storage format

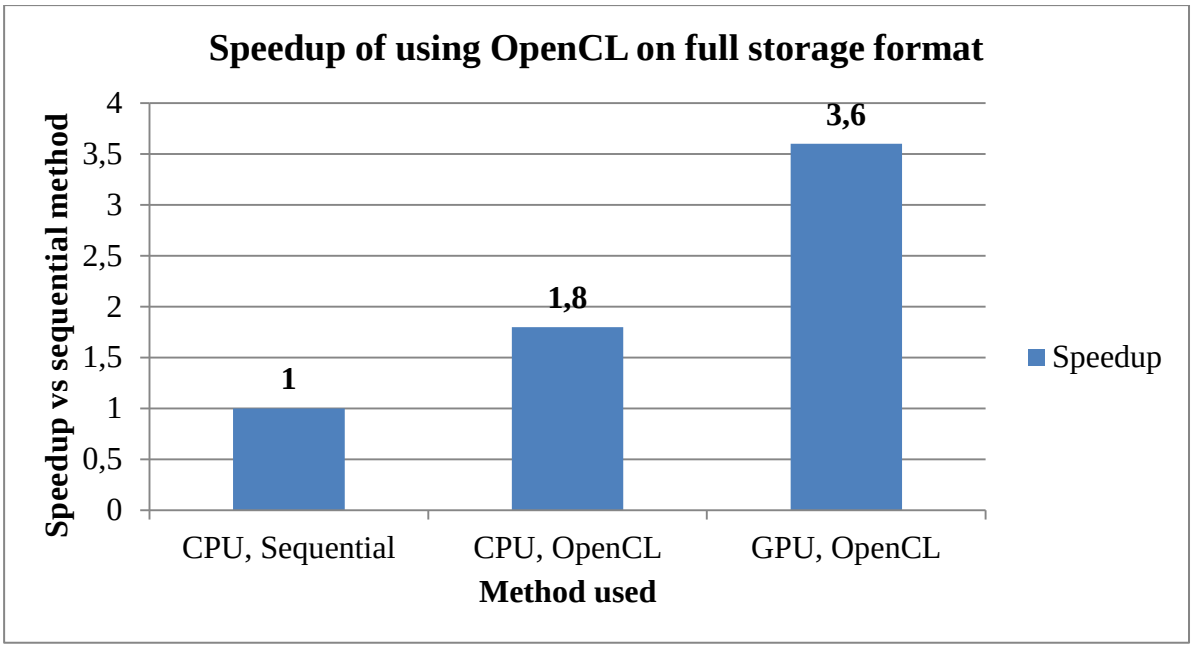

Figure 9 - Speedup of OpenCL over sequential C++ version, using full representation.

5 Coordinate Storage Format (COO)

5.1 Description

The coordinate storage format is made by storing the non-zero values in an array (Values) and the row and column indices in two other arrays (RowIdx, ColIdx). (See Figure 10)

0	0	3	1	0
2	0	0	4	0
0	0	2	0	0
0	3	0	1	0
4	0	2	0	3

Values =	3	1	2	4	2	3	1	4	2	3
RowIdx =	0	0	1	1	2	3	3	4	4	4
ColIdx =	2	3	0	3	2	1	3	0	2	4

Figure 10 - Original matrix and the COO representation.

Storage requirement for the coordinate storage format is the following:

SR = (2 x IdxSize + ValueSize) x NonZeros

(IdxSize - size of the indices,

ValueSize - size of the individual values stored in the matrix,

NonZeros - number of non-zero elements in the matrix.)

5.2 Calculation

For this storage format, we can create 1 work-item per non-zero element in the matrix. So each kernel execution multiplies a matrix and vector element and then adds it to a global result vector. This is done by using an atomic operation in OpenCL. Atomics are a way for different work-items to access a common part of the memory. This is not fast when compared with non-atomic memory operations, because of different cores having to wait for access to the area of memory [13].

5.3 Performance

Table 8 shows the performance of the COO format on the HD5870 and Tesla T10 GPUs. Here, the HD5870 is on average 30 times faster than the T10.

Matrix	**Millions of non-zero elements per second, OpenCL (HD5870)**	**Millions of non-zero elements per second, OpenCL (Tesla T10)**
heart1	6,67	0,08
Trefethen_20000b	15,44	0,75
TSOPF_RS_b162_c3	10,49	0,22
psmigr_1	6,18	0,10
spiral	6,78	0,41
viscoplastic1	9,27	0,61
deter4	12,02	0,94
Maragal_2	7,26	0,47
fp	11,31	0,42

Table 8 - Performance of COO format

This was an unexpected result, because in other tests, the performance of these GPUs was fairly similar.

Contrary to other OpenCL kernels, this one uses atomic operations. A difference of the GPUs is the version of OpenCL that they support – 1.0 for the T10, 1.1 for the HD5870.

To use atomics in OpenCL 1.0, they need to be included by a directive:

#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable

In OpenCL 1.1, the operations are already built-in.

Based on these facts, I decided to make a new kernel to measure the speed of atomics with both of the GPUs. (See Appendix 1 – items 15 & 16) The kernels add floating point numbers to a common memory location by using an atomic operation. The results showed that these operations were 15-20 times slower on the nVidia platform. (See Table 9) As of writing this thesis, I have not yet found an explanation for this discrepancy.

Number of work-items	HD5870	Tesla T10	
50000	9,09s	144,67s	15,9x slower
50	0,18s	3,66s	20x slower

Table 9 – Results of the atomic operations test.

6 ELLPACK (ELL)

6.1 Description

A matrix is converted to the ELL format by first determining the maximum number of non-zero elements in a row. This is done, because when the non-zero elements are stored in an array (Values), all of the shorter rows are padded with zeros to be the same length. Another array (ColIdx) contains the column index for the corresponding element in Values. (See Figure 11) The index for the padded elements is P, which could be set to a distinct negative number, for example *-1*. Since the padded value is 0, it does not actually make a difference in the calculation.

0	0	3	1	0
2	0	0	4	0
0	0	2	0	0
0	3	0	1	0
4	0	2	0	3

Values = | 3 | 1 | 0 | 2 | 4 | 0 | 2 | 0 | 0 | 3 | 1 | 0 | 4 | 2 | 3 |

ColIdx = | 2 | 3 | P | 0 | 3 | P | 2 | P | P | 1 | 3 | P | 0 | 2 | 4 |

MaxNonZeros = 3

Figure 11 - Original matrix and the ELL representation.

Storage requirement for the ELL storage format is the following:

$$SR = (IdxSize + ValueSize) \times N \times MNZ$$

(IdxSize - size of the indices,

ValueSize - size of the individual values stored in the matrix,

N - number of rows,

MNZ- maximum number of non-zero elements in a row.)

The storage efficiency of ELL depends on the variation of non-zero elements per rows. A lot of space can be wasted on padding shorter rows if they vary greatly in length.

6.2 Calculation

When using the ELL format, one work-item is created per row. Since we know that all the rows were padded to the length of the maximum number of non-zero elements in a row (MaxNonZeros), it is easy to calculate where each row starts and ends.

By using the function *get_global_id(0)*, a work item can find out its position in the work-group. For this algorithm it is the row that the work-item is multiplying.

By looking at the column indices corresponding to the values, we can select the appropriate vector element for the multiplication (Or skip it if the value is padded). Each work-item accumulates these results into a specific row in the answer.

6.3 Performance

For the kernel and host code used for the following measurements, see Appendix 1 – items 11 and 12.

Table 10 and Figure 9 show the performance of the ELL format on the HD5870 and Tesla T10 GPUs. The ELL format shows a big improvement over the COO format, being on average 7 times faster on the HD5870. The T10 is only ahead on one of the matrices, being 10-50% slower than the HD5870 on others.

The T10 has more compute units, but on the HD5870 the compute-units contain more stream processors. (240 on the T10, 1600 on the HD5870.) The theoretical processing power (single precision) on the HD5870 is 2,72 TeraFLOPS, 1036 GigaFLOPS on a T10 GPU. The processors on the HD5870 are simpler and do not provide as good latency, but

multiplying a matrix and vector element is a simple operation. The larger number of stream processors gives the ATI GPU an advantage in these tests [17,18,19].

Matrix	Millions of non-zero elements per second OpenCL (HD5870)	Millions of non-zero elements per second OpenCL (Tesla T10)
heart1	126,16	70,81
Trefethen_20000b	143,61	179,51
TSOPF_RS_b162_c3	122,06	108,98
psmigr_1	28,07	13,25
spiral	32,77	16,39
viscoplastic1	87,38	61,17
deter4	42,74	32,05
Maragal_2	21,79	7,26
fp	43,29	24,53

Table 10 - Performance of ELL format

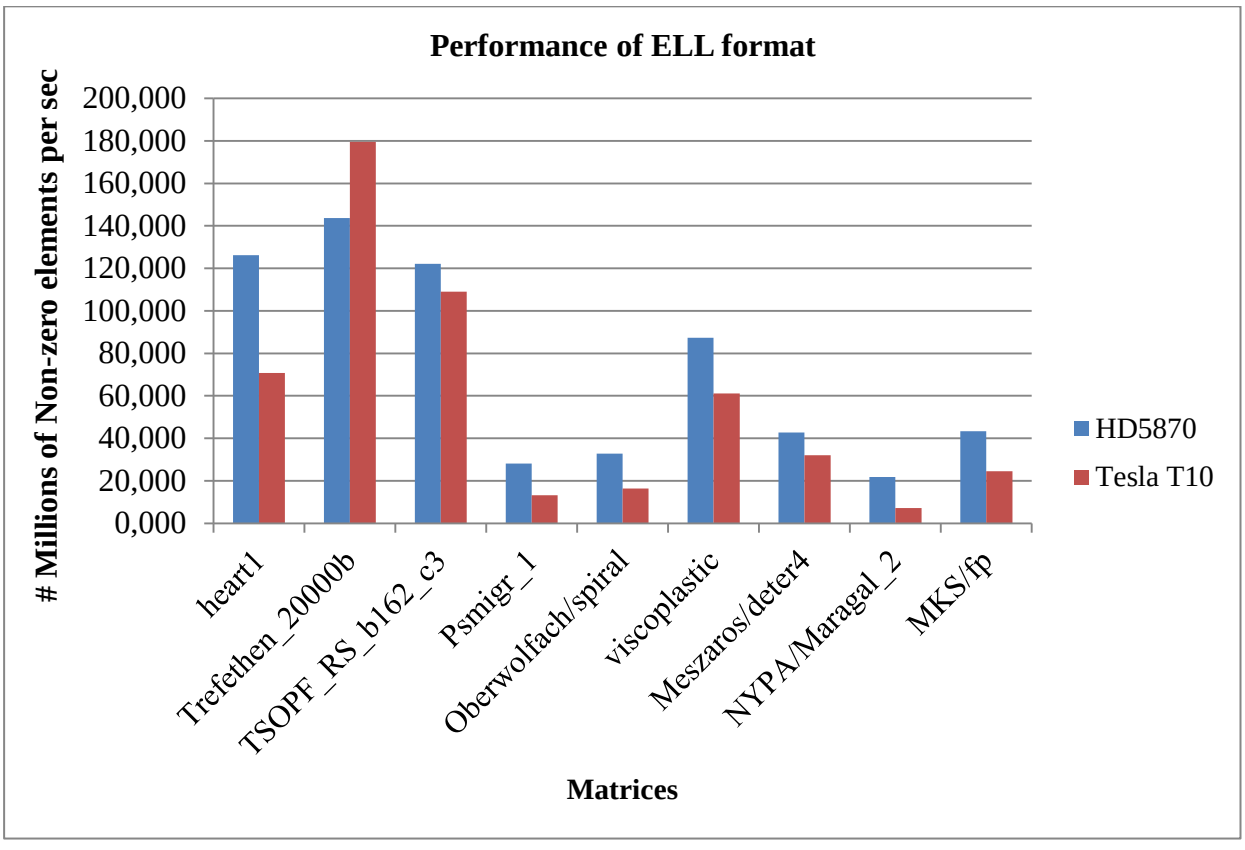

Figure 9 - Performance of ELL format

6.4 Performance Using OpenCL Images

A feature of OpenCL is the ability to represent numerical data as images. Using images can provide additional memory bandwidth by using the texture caches on GPUs and the texture sampling hardware for reading from them. To test this, I created a new kernel (See Appendix 1, item 12 - ellpack_opencl_img.cl), still using the ELL format, but this time the vector was given as a 2D-image. Results can be seen in Table 11.

Matrix	HD5870, Vector as array.	HD5870, Vector as image.	Tesla T10, Vector as array.	Tesla T10, Vector as image.
heart1	126,16	36,14	70,81	81,16
Trefethen_20000b	143,61	71,80	179,51	179,51
TSOPF_RS_b162_c3	122,06	46,24	108,98	117,37
psmigr_1	28,07	7,83	13,25	16,87
spiral	32,77	12,29	16,39	16,39
viscoplastic1	87,38	15,29	61,17	61,17
deter4	42,74	13,74	32,05	36,54
Maragal_2	21,79	10,89	7,26	10,89
fp	43,29	12,78	24,53	29,36

Table 11 – Performance in millions of non-zero elements per second.

When using the HD5870, converting the vector into an image dropped the performance by over 50%. (See Figure 10) On the other hand, the nVidia T10 GPU had an average 17% performance increase. (See Figure 11).

The speed up (or slow down) of using images is determined by the data usage patterns and texture hardware. When the vector element being requested is not in the texture cache (a cache-miss), it will trigger a new global memory fetch. When the percentage of elements that generate a cache-miss is high, using an image can be slower than normal memory [15].

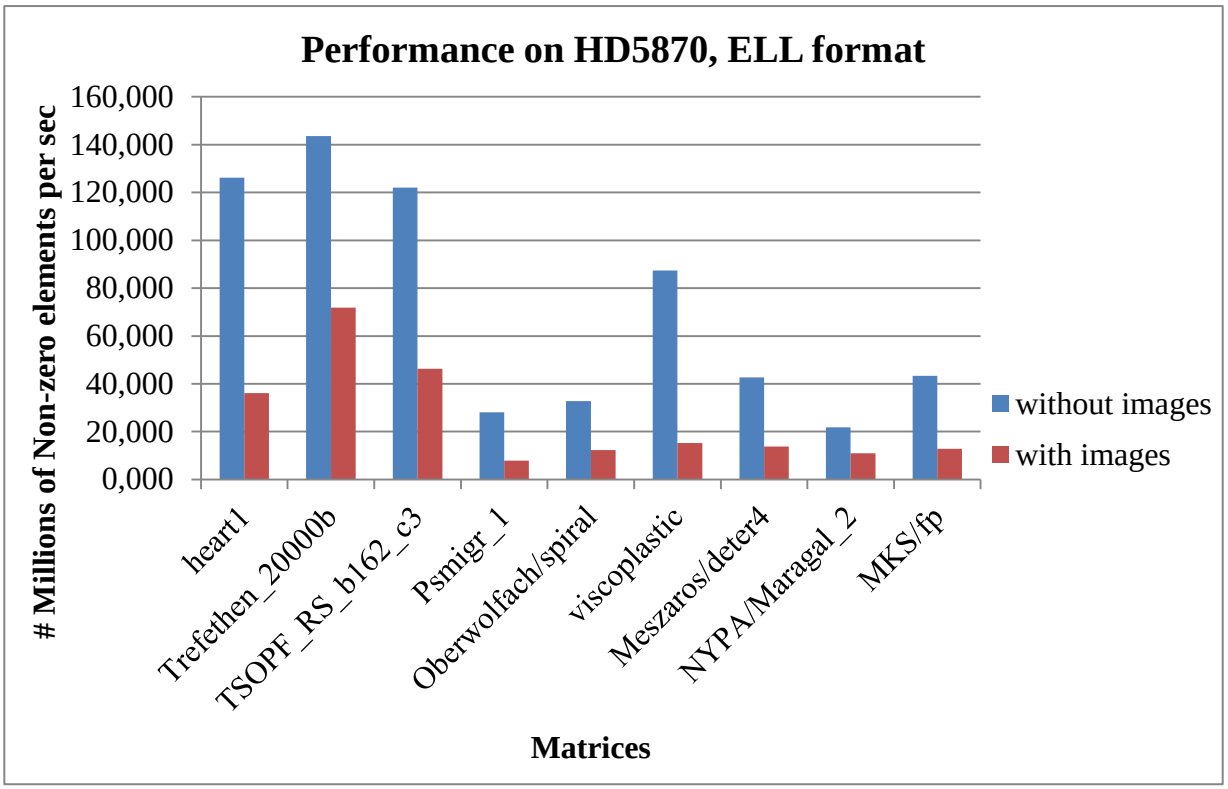

Figure 10 – performance on the HD5870, ELL format, vector as image.

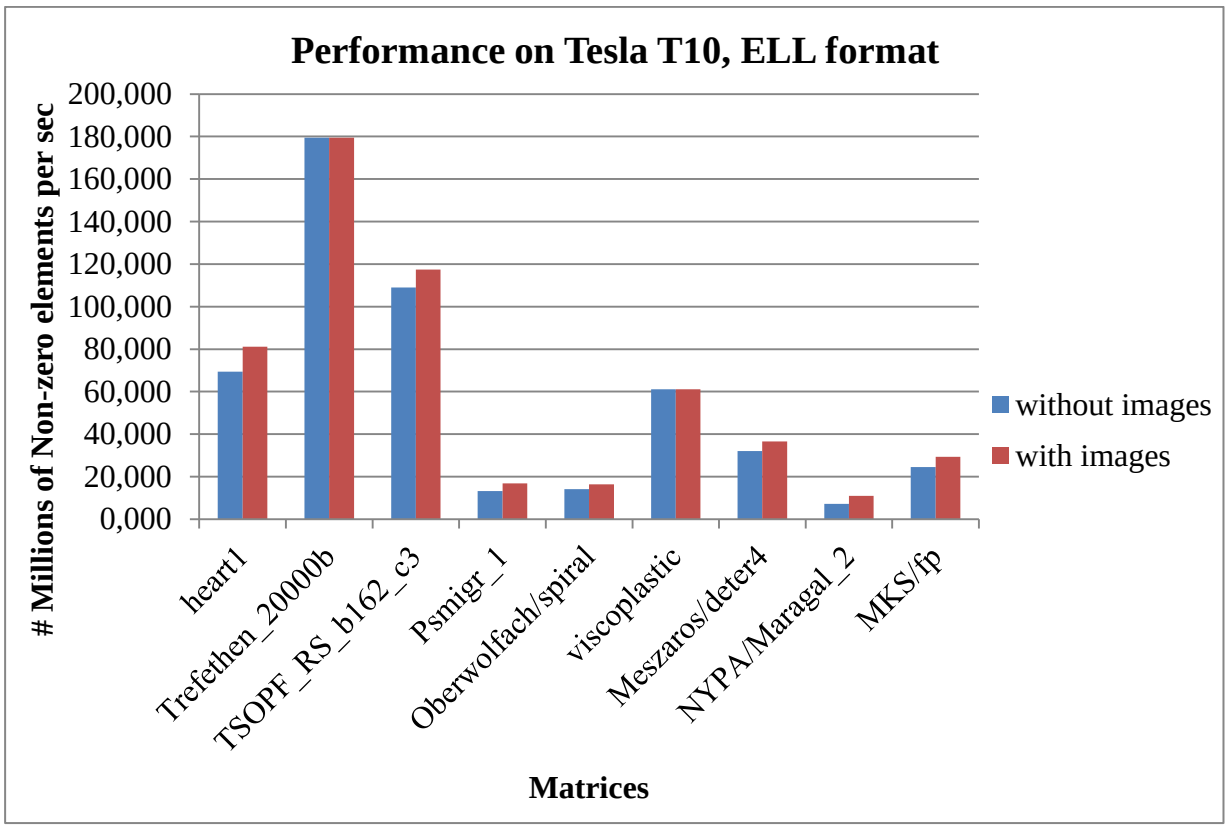

Figure 11 – performance on the T10, ELL format, vector as image.

7 Compressed Sparse Row (CSR)

7.1 Description

In the compressed sparse row format, the non-zero elements are stored in one array (Values) and the column indices for each of those elements in another (ColIdx). A third array (RowPtr) contains the indices at which each row starts in the Values array (See Figure 12).

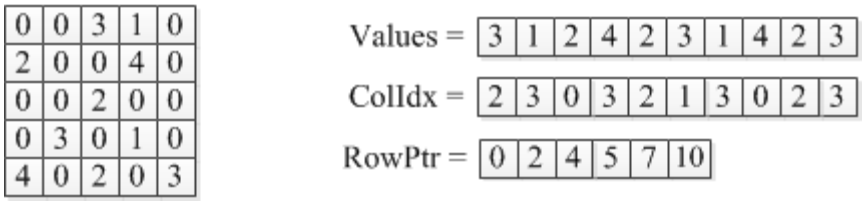

Figure 12 - Original matrix and the CSR representation.

Storage requirement for the CSR storage format is the following:

SR = (IdxSize + ValueSize) x NonZeros + IdxSize x N

(IdxSize - size of the indices,

ValueSize - size of the individual values stored in the matrix,

N - number of rows,

NonZeros - number of non-zero elements in the matrix.)

7.2 Calculation

Like the ELL format, one work-item is created per row. Again, by using the function *get_global_id(0)*, a work item can find out its position in the work-group. For example, if this position is "*pos*", then the kernel knows that the row it has to multiply starts at *RowPtr[pos]* and ends at *RowPtr[pos+1]-1*.

Column indices also work much the same way as in the ELL format - by looking at the column indices corresponding to the values, we can select the appropriate vector element for the multiplication. Each work-item accumulates these results into a specific row in the answer.

7.3 Performance

For the kernel and host code used for the following measurements, see Appendix 1 – items 9 and 10.

The performance difference between CSR and ELL in some cases is more than double in favor of the former. (See Tables 11 and 12, Figures 10, 11 and 13) An explanation can be found by looking at the storage requirements of the two formats. I will use the matrix „heart1“ as an example, as it is one where this big difference in performance is present. The following formulas are described in more detail in chapters 6.1 and 7.1.

The storage requirement for CSR format is:

SR = (IdxSize + ValueSize) x NonZeros + IdxSize x N = (4 + 4) x 1387773 + 4 x 3557 = 11,116,412 bytes

(The IdxSize using integer type is 4 bytes, ValuesSize using float type is also 4 bytes.)

For ELL:

SR = (IdxSize + ValueSize) x N x MNZ = (4 + 4) x 3557 x 1120 = 31,870,720 bytes

(The maximum number of non-zero elements in a row was found during the conversion process to ELL format.)

The ELL format of this matrix requires close to 3 times as much memory as the CSR variant. This means that more time is used to copy the data to a device and the bandwidth is not used as effectively.

Matrix	**Millions of Non-zero elements per second** **OpenCL (HD5870)**	**Millions of Non-zero elements per second** **OpenCL (Tesla T10)**
heart1	346,94	157,70
Trefethen_20000b	143,61	179,51
TSOPF_RS_b162_c3	254,29	203,43
psmigr_1	319,51	47,65
spiral	98,31	49,16
viscoplastic1	174,76	152,92
deter4	96,16	24,04
Maragal_2	87,14	21,79
fp	326,37	192,85

Table 12 - Performance of CSR format

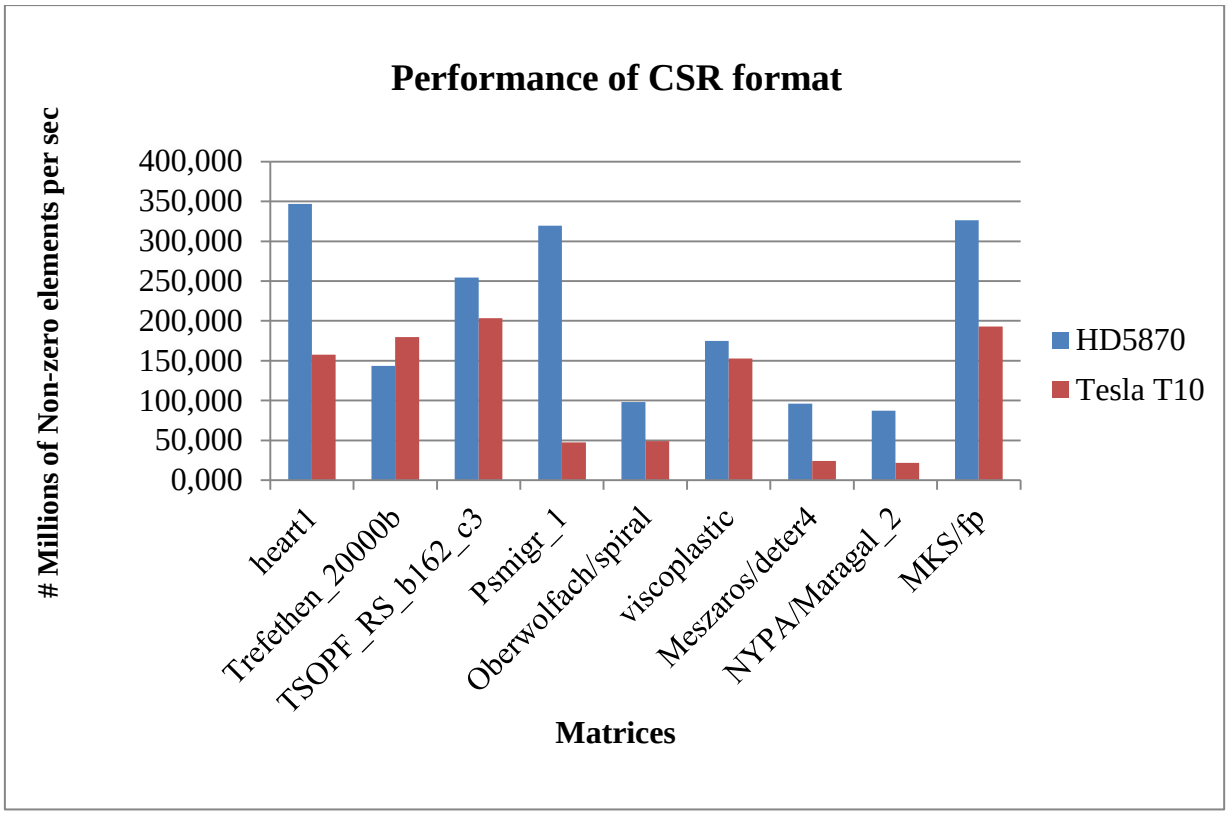

Figure 13 - Performance of CSR format

8 Compressed Diagonal Storage Format (CDS)

8.1 Description

In the CDS format, diagonals that contain non-zero elements are stored. To make the addressing of the elements uniform, all of the diagonals are padded to the length of the longest diagonal (See Figure 14).

This format is really only suitable for sparse matrices where the non-zero elements are placed densely along diagonals. On other types of matrices, too much space and processing time is wasted on elements that were padded or were zeros to begin with.

1	0	3	1	0
0	4	0	2	4
0	0	2	0	0
3	0	0	1	0
0	2	0	0	3

Values = | 0 | 0 | 0 | 1 | 4 | 0 | 0 | 3 | 2 | 0 | 1 | 4 | 2 | 1 | 3 | 3 | 2 | 0 | 0 | 0 |

Offset = | 3 | 2 | 0 | -3 |

Figure 14 - A diagonal sparse matrix and the CDS representation

Storage requirement for the CDS storage format is the following:

SR = ValueSize x NonZeroDiags x LongestDiag + OffsetSize x NonZeroDiags

(ValueSize - size of the individual values stored in the matrix,

NonZeroDiags - number of diagonals with non-zero elements in the matrix,

LongestDiag – Number of elements in the longest diagonal,

OffsetSize - size of the offset indices.)

8.2 Calculation

For the CDS format, one work-item is created per row. By using the function *get_global_id(0)*, a work item finds out its position in the work-group. The work-item then loops through all the diagonals, multiplying the elements in this row with the corresponding ones from the vector. The column of the matrix elements is calculated by using the offsets array.

8.3 Performance

For the kernel and host code used for the following measurements, see Appendix 1 – items 13 and 14.

As discussed earlier, the only suitable type of matrix for this type of storage is one, where the non-zero elements are located densely on the diagonals. Out of the matrices chosen for the tests, only one fits this criteria - Trefethen_20000b. The structure of this matrix can be seen on Figure 15. White areas contain zeros, blue areas non-zero elements.

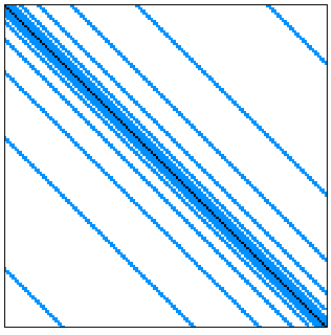

Figure 15 – Structure of matrix „Trefethen_20000b“ [12].

The results are indicative of this – matrices besides „Trefethen_20000b“ achieved very poor performance using the CDS storage format.

Matrix	Millions of Non-zero elements per second OpenCL (HD5870)	Millions of Non-zero elements per second OpenCL (Tesla T10)
heart1	5,61	17,39
Trefethen_20000b	143,61	159,57
TSOPF_RS_b162_c3	0,54	1,67
psmigr_1	2,44	7,94
spiral	2,89	2,89
viscoplastic1	0,19	0,57
deter4	0,10	0,22
Maragal_2	1,71	1,98
fp	0,47	2,25

Table 13 - Performance of CDS format.

9 Performance Using Two Devices Simultaneously

This chapter describes the results of distributing the calculation between two Tesla T10 GPUs. For the kernels and host code used for the following measurements, see Appendix 1 – items 19 through 22.

A drawback of the OpenCL version 1.0, which is the highest for nVidia devices at this time, is the lack of support for setting a global work offset [16]. This is a feature added in version 1.1 and it allows to set the global position, where kernels start executing. For example, an offset of *'5'* would mean that the *get_global_id(0)* call in the first kernel would return *'5'*. This way, the rows of the matrix could be grouped and distributed for calculation in multiple kernels and devices. A solution is to pass the offset as an argument to the kernel ourselves.

A better approach is to also divide the data into segments. In the case of ELL format, each device could handle the calculation of a certain amount of rows in the original matrix. In the tests described in this chapter, the data was divided in half. If *n* is the total number of rows then one device would calculate rows *1* to *n / 2*, the other *n / 2* to *n*.

The speed up of this approach depends highly on the structure of the matrix. If all the non-zero elements are in the upper half or vice-versa, there will be no performance gain at all. If the non-zero elements are evenly distributed, the speed up can ideally be close to 100%.

Results of the tests can be seen in Table 14 and Figure 16. With the smaller matrices, there was actually a slowdown when using two devices. This can be explained by the too small amount of input data when compared to the additional overhead of launching another kernel and command queue. With bigger matrices, speed-ups of up to 60% were gained.

Matrix	Millions of Non-zero elements per second, ELL, OpenCL (Tesla T10)	Millions of Non-zero elements per second, ELL, OpenCL (2 x Tesla T10)	Millions of Non-zero elements per second, CSR, OpenCL (Tesla T10)	Millions of Non-zero elements per second, CSR, OpenCL (2 x Tesla T10)
heart1	81,16	144,56	157,70	255,11
Trefethen_20000b	179,51	205,16	179,51	241,36
TSOPF_RS_b162_c3	117,37	195,61	203,43	216,42
psmigr_1	16,87	32,64	47,65	81,31
spiral	16,39	13,65	49,16	15,36
viscoplastic1	61,17	61,17	152,92	83,79
deter4	27,47	27,47	24,04	25,64
Maragal_2	10,89	6,22	21,79	9,68
fp	29,36	56,80	192,85	238,36

Table 14 – Performance on one Tesla T10 GPU compared to two at the same time.

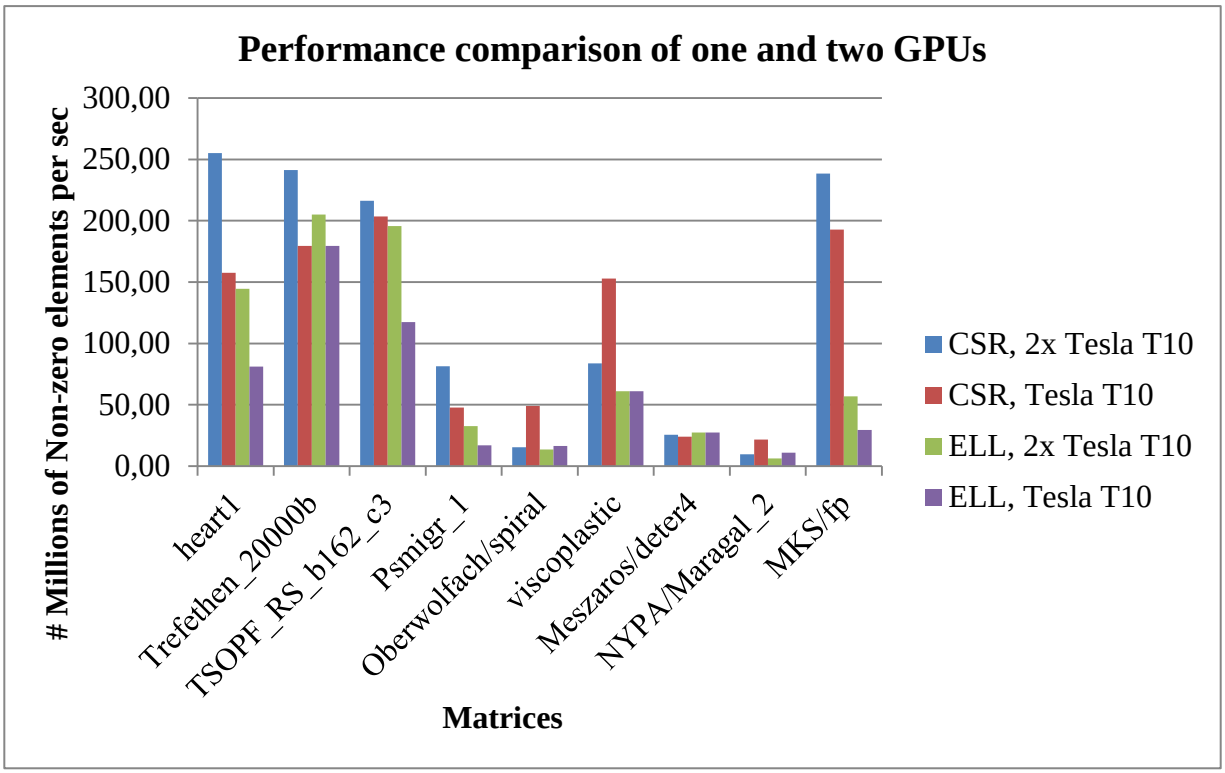

Figure 16 – Performance on one Tesla T10 GPU compared to two at the same time.

III Conclusions and Further Development

The purpose of this thesis was to benchmark and compare different representations of sparse matrices and algorithms for multiplying them with a vector. Also, to see the performance differences of running the algorithms on a CPU and GPU(s). Four different storage formats were tested – full matrix storage, coordinate storage (COO), ELLPACK (ELL), compressed sparse row (CSR) and compressed diagonal storage (CDS).

Performance tests were run on a desktop computer and also on EENet (the Estonian Education and Research Network) worknodes. The EENet worknodes added the opportunity for dividing the workload between their two GPUs.

Using OpenCL gave a speedup of 3,6 times over pure C++ code when using the full storage method and basic algorithm. With more complex storage formats, the speed gain was even more distinct. A combined report of the results can be seen on Figure 17. The COO format on the Tesla T10 is not included, because of its almost non-existent performance. ELL format on the T10 is the improved version with the vector represented as a texture.

Converting the vectors into images did not give the expected speedup on most cases. Still, it performed slightly better on the nVidia hardware. An option would be to create both kinds on kernels in a situation like this – one with image support, another with normal memory access, and see which one performs better. The conversion into textures requires only slight modifications of the kernel and host-code.

The problem with using OpenCl is the need to effectively parallelize the original task and use as much of the available computing power as possible. As the results show, the performance is highly dependent on the type of matrix and hardware used. For an all-round choice the CSR seems to be the best, being the fastest in all tests. This may of course change with the selection of new matrices and further optimization of kernels.

The performance benefit when using multiple devices also depends on the type of matrices used – with smaller ones, the additional overhead of creating a new command queue and kernel execution can nullify the advantage of more processing power. With larger matrices, speed ups of up to 60% were noted.

As a continuation of this work, more storage formats could be tested. There are many more to choose from – ELLPACK-R, CSC (Compressed Sparse Column), JAD (Jagged Diagonal Storage), etc. A better algorithm could be developed to spread computational load more evenly to multiple devices. The CPU could also be utilized, by giving it a smaller task in the calculation. Also, reasons for the low performance of COO format on the Tesla T10 GPU could be investigated.

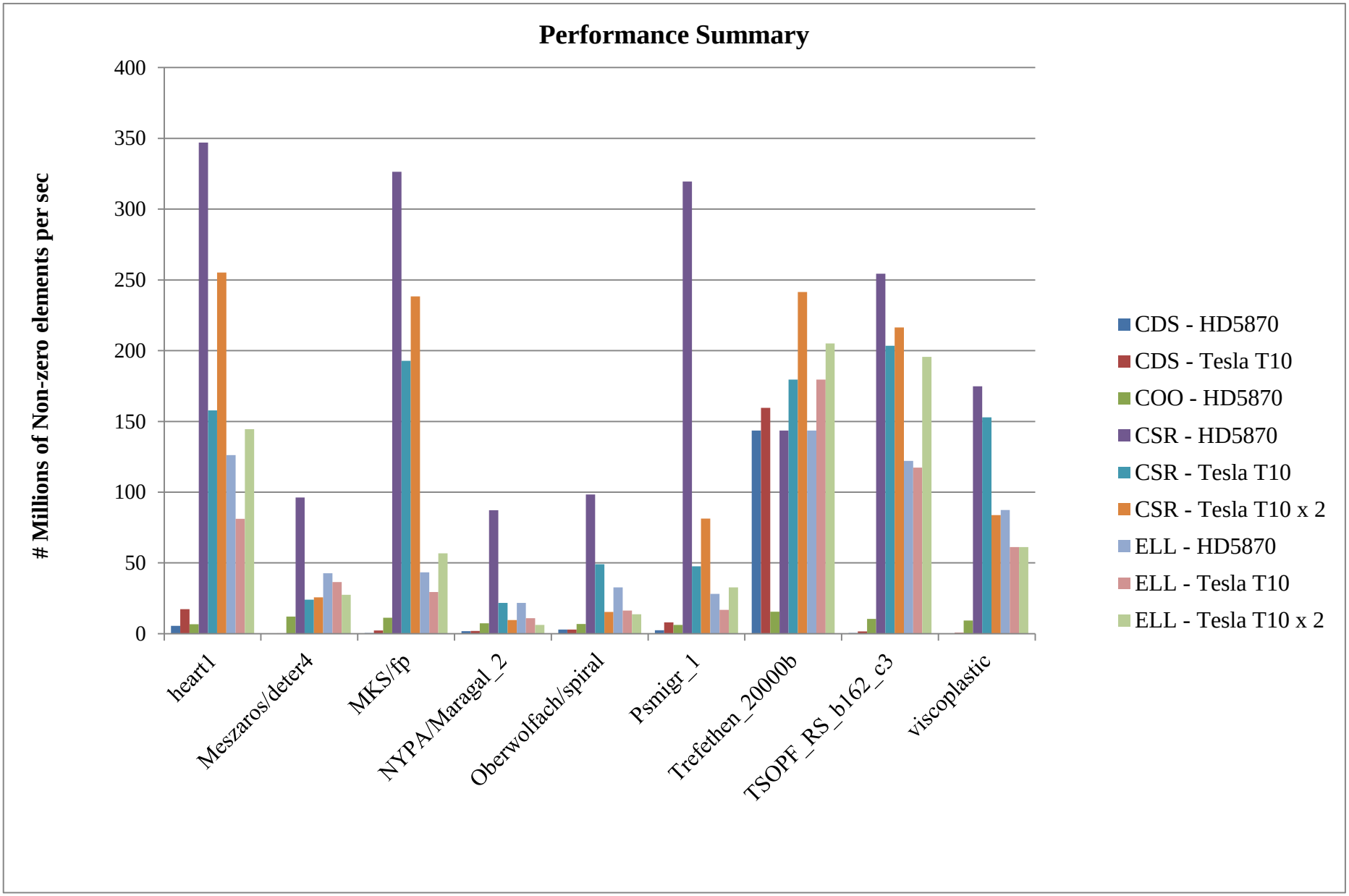

Figure 17 – Performance summary on a per matrix basis.

Hõreda maatriksi algoritmid kasutades GPGPU-d

Kaupo Kuresson

Bakalaureusetöö (6 EAP)

Resümee

Antud bakalauresetöö eesmärgiks oli lahendada võimalikult efektiivselt suuremahulisi arvutusi nõudvaid ülesandeid, kasutades selleks GPGPU'd ehk üldotstarbelist arvutamist graafikakaartidel. Konkreetse näitena vaadeldi hõreda maatriksi ning vektori korrutise leidmist. Maatriksi ja vektori korrutamine on aluseks paljudele algoritmidele – näiteks pilditöötlus ja masinõpe.

Hõre maatriks on maatriks, mille enamus elemente on nullid. Kuna nullid vektoriga korrutamisel lõpptulemust ei muuda, on eesmärgiks vältida ebavajalikku nullide korrutamist. Selle saavutamiseks saab muuta kasutatavat algoritmi ja viisi, kuidas maatriksit salvestatakse.

Lõputöö käigus testiti nelja erinevat hõreda maatriksi salvestamise formaati. Vaatluse all oli formaatide eripärasid arvestades loodud maatriksi ja vektori korrutamise algoritmide jõudlus ja mäluvajadus. Formaatideks olid „täielik", „koordinaadipõhine", „ELLPACK", „pakitud hõredad read" ja „pakitud diagonaalid". Eesmärgiks oli hinnata ka algoritmide jõudluserinevust protsessori ja graafikakaardi rakendamisel.

Algoritmide realiseerimiseks kasutati OpenCL'i. OpenCl on raamistik, mille abil saab kirjutada programme, mis võivad käskude täitmiseks kasutada nii protsessoreid kui ka graafikakaarte. Põhiliseks raskuseks on sealjuures ülesande jagamine väiksemateks osadeks, et neid saaks lahendada paralleelselt ja arvutusjõudlust efektiivsemalt ära kasutada.

Teste jooksutati autori lauaarvutil ja EENeti (Eesti Hariduse ja Teaduse Andmesidevõrk) arvutussõlmedel. EENeti kaudu avanes lisavõimalus proovida arvutusülesannete jagamist kahe graafikakaardi vahel.

„Täielikku" salvestusformaati kasutades oli OpenCL-i kasutamine tavalise C++ koodiga võrreldes 3,6 korda kiirem. Keerukamate formaatide puhul oli jõudluse kasv veelgi märgatavam.

Tulemustest ilmnes, et jõudlus sõltub suuresti maatriksite struktuurist ja kasutatud riistvarast. Näiteks sai „koordinaadipõhine" formaat nVidia graafikakaardil ATI omaga võrreldes ligi 30 korda halvemaid tulemusi.

ELLPACK formaadi puhul andis nVidia kaardile lisajõudlust vektori tekstuurina esitamine. ATI kaart sai aga võrreldes vektori tavalise esitusega poole võrra halvema tulemuse.

Testide põhjal tundus universaalse lahendusena parim „pakitud hõredad read" formaat, mis andis parima tulemuse kõigi maatriksite puhul. See võib aga uute maatriksite valikul muutuda.

Algoritmide kahe graafikakaardi vahel jagamine tagas suuremate elementide arvuga maatriksite puhul kiiruse kasvu kuni 60%. Teise seadme kasutamisel peab arvestama väljakutsete suurema arvu ja lisakulude kasvuga. See tähendab, et väiksemate maatriksite puhul, kus arvutamine võtab vähem aega, ei pruugi jõudlus kahe seadmega suureneda. Testitulemustest oligi näha, et väiksemate maatriksite puhul oli kahe graafikakaardiga saadud tulemus aeglasem, kui ühega.

Bibliography

[1] Khronos Group, OpenCL, http://www.khronos.org/opencl/

(Last visited 06.05.2012).

[2] nVidia, OpenCL Overview, http://gpgpu.org/wp/wp-content/uploads/2009/06/05-OpenCLIntroduction.pdf

(Last visited 10.05. 2012).

[3] Wikipedia, OpenCL, http://en.wikipedia.org/wiki/OpenCL

(Last visited 10.05. 2012).

[4] OpenCL Overview, Ofer Rosenberg, AMD, November 2011,

http://www.khronos.org/assets/uploads/developers/library/overview/opencl-overview.pdf (Last visited 06.05.2012).

[5] HPC Wire, OpenCL gains ground on CUDA, Michael Feldman,

http://www.hpcwire.com/hpcwire/2012-02-28/opencl_gains_ground_on_cuda.html

(Last visited 10.05. 2012).

[6] nVidia, CUDA, http://developer.nvidia.com/what-cuda

(Last visited 08.05. 2012).

[7] SHOC: Overview and Kernel Walkthrough, Kyle Spafford,

http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2012-02-20/13-shoc.pdf

(Last visited 08.05. 2012).

[8] OpenCL™ Optimization Case Study: Diagonal Sparse Matrix Vector Multiplication

http://developer.amd.com/documentation/articles/Pages/OpenCL-Optimization-Case-Study.aspx

(Last visited 08.05. 2012).

[9] Algorithms in Scientic Computing II, Michael Bader 2011

http://www5.in.tum.de/lehre/vorlesungen/algowiss2/WS11/intro.pdf

(Last visited 07.05. 2012).

[10] Generating and Automatically Tuning OpenCL Code for Sparse Linear Algebra, http://www.many-core.group.cam.ac.uk/ukgpucc2/talks/Grewe.pdf

(Last visited 07.05.2012).

[11] The University of Florida Sparse Matrix Collection,

http://www.cise.ufl.edu/research/sparse/matrices/ (Last visited 07.05.2012).

[12] Matrix Market formats, http://math.nist.gov/MatrixMarket/formats.html

(Last visited 07.05.2012).

[13] AMD OpenCL Forum, thread “A question about atomic_add”, http://devgurus.amd.com/thread/158548 (Last visited 07.05.2012).

[14] nVidia Tesla S1070 Overview, nVidia Corporation,

www.nvidia.com/docs/IO/43395/SP-04154-001_v02.pdf

(Last visited 07.05.2012).

[15] Stackoverflow forum, Thread „Disadvantages of using Texture Cache / Image2D for 2D Arrays?“, http://stackoverflow.com/questions/7258139/disadvantages-of-using-texture-cache-image2d-for-2d-arrays

(Last visited 06.05.2012).

[16] Khronos Group, OpenCL 1.0 API,

http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/clEnqueueNDRangeKernel.html

(Last visited 10.05.2012).

[17] Wikipedia, nVidia Tesla, http://en.wikipedia.org/wiki/Nvidia_Tesla

(Last visited 10.05.2012).

[18] AMD, ATI Radeon HD5870 specification,

http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5870/pages/ati-radeon-hd-5870-overview.aspx#2

(Last visited 10.05.2012).

[19] AMD, AMD Accelerated Parallel Processing OpenCL Programming Guide http://developer.amd.com/sdks/amdappsdk/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf

(Last visited 10.05.2012).

Appendix 1 - Source code

The source code for all the opencl kernels and the host program can be accessed on Google Code:

http://code.google.com/p/sparse-matrix-algorithms-opencl/source/browse/#svn/trunk/gpgpu/tt

#	Filename	Description
1	main.cpp	Entry point for the program.
2	matrix_operations.cpp , matrix_operations.hpp	For importing and converting matrices to different formats.
3	init_opencl.cpp , init_opencl.hpp	Initializes OpenCL context.
4	basic_cpp.cpp , basic_cpp.hpp	Uses C++ code to find the product of a matrix and vector.
5	basic_opencl.cpp , basic_opencl.hpp	Host code for OpenCL, full storage format.
6	basic_opencl.cl	OpenCL kernel, full storage format.
7	coo_opencl.cpp , coo_opencl.hpp	Host code for OpenCL, COO format.
8	coo_opencl.cl	OpenCL kernel, COO format.
9	csr_opencl.cpp , csr_opencl.hpp	Host code for OpenCL, CSR format.
10	csr_opencl.cl , csr_opencl_img.cl	OpenCL kernels, CSR format. With / without representing the vector as a 2D-image.
11	ellpack_opencl.cpp , ellpack_opencl.hpp	Host code for OpenCL, ELL format.

12	ellpack_opencl.cl , ellpack_opencl_img.cl	OpenCL kernels, ELL format. With / without representing the vector as a 2D-image.
13	cds_opencl.cpp , cds_opencl.hpp	Host code for OpenCL, CDS format.
14	cds_opencl.cl , cds_opencl_img.cl	OpenCL kernels, CDS format. With / without representing the vector as a 2D-image.
15	atomic_test.cpp , atomic_test.hpp	Host code for testing the performance of atomic operations.
16	atomic_test.cl	OpenCl kernel for testing the performance of atomic operations.
17	genMatrix.m	A MatLab script to generate a random matrix / vector.
18	KK_0.xrsl	Job description file used for accessing EENet resources.
19	csr_opencl_twodev.cl	Kernel for OpenCL, CSR format, using two devices.
20	csr_opencl_twodev.cpp, csr_opencl_twodev.hpp	Host code for OpenCL, CSR format, using two devices.
21	ellpack_opencl_twodev.cl	Kernel for OpenCL, ELL format, using two devices.
22	ellpack_opencl_twodev.cpp, ellpack_opencl_twodev.hpp	Host code for OpenCL, ELL format, using two devices.

Appendix 2 – A Guide for Setting Up the Environment

Requirements:

- OpenCL.lib in Windows / OpenCL.so in Linux
- OpenCL headers.

For ATI users, these are packaged with the AMD APP SDK:
http://developer.amd.com/sdks/amdappsdk/pages/default.aspx

For nVidia, the library is included with display drivers, headers with the Cuda Toolkit:
http://developer.nvidia.com/cuda-toolkit-30-downloads

To set up the project in MS Visual Studio:

- Project properties -> C/C++, General -> Additional Include Directories
  add "C:\Program Files (x86)\AMD APP\include"

- Project properties -> Linker, General -> Additional Library Directories
  add "C:\Program Files (x86)\AMD APP\lib\x86"

- Project properties -> Linker, Input -> Additional Dependencies
  add "opencl.lib"

The process is similar when using other SDKs – include the additional headers and link the OpenCL library.