Universiteit Leiden
Opleiding Informatica

Implementation of A Parallel

Back Substitution Solver on GPUs

Name: Joost Nibbeling

Date: 29/08/2016

1st supervisor: K.F.D. Rietveld

2nd supervisor: H.A.G. Wijshoff

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)

Leiden University

Niels Bohrweg 1

2333 CA Leiden

The Netherlands

Abstract

Level scheduling is a commonly used technique to parallelize sparse matrix computations and groups the rows of a matrix that are independent of each other in blocks. Back substitution can then be performed to solve a matrix vector equation with this matrix, but instead of iterating over the rows, it is possible to iterate over the blocks and compute the results of all the rows in a single block in parallel. In this thesis we describe the implementation of a solver that finds these blocks and performs the back substitution on a GPU using CUDA. The goal is for the solver to be faster than solutions for generic matrices. To evaluate this we benchmark our solver and compare it with the performance of the cuSPARSE library. These results show that our solver is faster for big matrices with few blocks and small rows or when the amount of equations with the same matrix but with different vectors remains low.

Contents

Chapter 1

Introduction

Sparse matrices are matrices of which most elements are zeros. Over the years much research has been done to optimize equations evolving sparse matrices [GMSW84]. Even now research is done in optimizing these for today's multi-core systems [BPC+16]. This thesis focuses on a solver for equations in the form of $Ax = b$. Here A is a sparse matrix, b is a known vector and x is an unknown vector that has to be found.

We concentrate on upper triangular sparse matrices with single diagonal blocks on the diagonal. These matrices arise when performing an ILU factorization on matrices which are reordered using level scheduling [Saa00], which is a commonly used technique to parallelize sparse matrix computations. Such matrices occur as U factor used as preconditioner for iterative methods in which many triangular solves are performed. In this case the solver can use a more specific form of backward substitution, where instead of one row at a time, multiple rows are processed in one step. This is done by taking advantage of rows that do not depend on each other, but only on a set of previously processed rows. The question to answer is: Can this solver outperform generic solvers that have to work for arbitrary matrices?

The solver has been implemented with NVIDIA CUDA. CUDA is a parallel programming platform and API for General Purpose GPU computing. It allows programmers to harness the compute power of modern day GPUs. These GPUs have evolved to highly parallel multiprocessors with more raw compute power than CPUs. They are especially suited for solving problems that can easily be expressed as data-parallel computations. The algorithm used by the solver is exactly one such problem.

In Chapter 2 we will describe the algorithm used by the solver and the matrices it operates on in greater detail. Then in Chapter 3 we will describe our implementation on the solver using CUDA. Following that Chapter 4 describes the benchmarks that have been performed to measure the performance of our solver and of the cuSPARSE library. Chapter 5 contains a discussion of the results of these benchmarks. Chapter 6 ends with a summary, conclusions and suggestions for future work.

Chapter 2

Back Substitution for GPUs

The algorithm used by the solver is a form of back substitution. First we will give a short description of back substitution in general. Then we will describe the more specific type of back substitution used by the solver and the matrices in operates on. Finally, we will discuss the analysis of the structure of the matrix that the solver needs to work.

2.1 Back Substitution

Back substitution can only be performed on matrices that are upper triangular. Upper triangular matrices are square matrices of which all elements below the diagonal are zero. Figure 2.1 shows an example equation of the form $Ax = b$ with A being a $3x3$ upper triangular matrix and b and x being vectors of size 3. The last row of an upper triangular matrix has only a single element. The equation resulting from the last row has therefore only a single unknown and can thus be solved easily. The result is the last element of the vector x. The second row has at most two elements: one on the diagonal and one off the diagonal. In the equation resulting from this row, the off diagonal element is multiplied with the last element of the vector x. This has already been found in the previous step. Substituting this again leaves only a single unknown, which means the second to last element can be found easily. This process can be continued until the entirety of x has been found. Therefore x in Figure 2.1 can be found in three steps. See Algorithm 1 for the full pseudocode for back substitution

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Figure 2.1: Equation of the form Ax = b with an upper diagonal matrix

input : Upper triangular matrix A of size n x n, Vector b of size n
output: Vector x of size n satisfying Ax = b

```
1 x[n] ← b[n] / A[n, n];
2 foreach row i of A from n − 1 to 1 do
3 |   s ← DotProduct(A[i, i +1 .. n], x[i+1 .. n]);
4 |   x[i] ← b[i] − (s / A[i, i]);
5 end
```

Algorithm 1: Back Substitution pseudocode

It is clear that for every element of the vector x, all the following elements need to be found first. This means it is impossible to compute elements of x fully in parallel. As a GPU wants to do many independent calculations in parallel, this algorithm is not suited for GPUs

2.2 Parallel Back Substitution

To be able to compute multiple elements of the vector x in parallel, a special form of upper triangular matrices is needed. This form has square blocks on the diagonal of which every element is zero, except the element on the diagonal itself.

$$\begin{bmatrix} a_{11} & 0 & a_{13} & a_{14} & a_{15} & a_{16} \\ 0 & a_{22} & a_{23} & a_{24} & a_{25} & a_{16} \\ 0 & 0 & a_{33} & 0 & a_{35} & a_{36} \\ 0 & 0 & 0 & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & 0 & a_{55} & 0 \\ 0 & 0 & 0 & 0 & 0 & a_{66} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \end{bmatrix}$$

Figure 2.2: Equation of the form Ax = b with upper diagonal matrix with zero blocks on diagonal

See Figure 2.2 for an example. Here there are three blocks on the diagonal of size $2x2$. These are indicated by the boxes. The elements of x that correspond to the rows that belong to the same block can be solved in parallel. In the example x_5 and x_6 can solved found immediately, as there are no other unknown variables involved in the equations resulting from the rows of the lowest block:

$$x_5 = b_6 / a_{55}$$

$$x_6 = b_6 / a_{66}$$

Then these variables can be substituted in the equations resulting from the rows in the middle box. This results in a matrix vector multiplication of the already known elements of x with the elements of the rows

that come after the block on the diagonal:

$$\begin{bmatrix} a_{35} & a_{36} \\ a_{45} & a_{46} \end{bmatrix} \cdot \begin{bmatrix} x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} s_3 \\ s_4 \end{bmatrix}$$

These results can then be used to compute x_3 and x_4:

$$x_3 = (b_3 - s_3)/a_{33}$$

$$x_6 = (b_4 - s_4)/a_{44}$$

This can be repeated for the next blocks until the entirety of x has been found. Thus the equation from Figure 2.2 can be solved in three steps. See Algorithm 2 for the full pseudocode.

input : Upper triangular matrix A of size n x n, Vector b of size n, blocks in A
output: Vector x of size n satisfying Ax = b

```
1 foreach row i in the first block do
2 |  x[i] ← b[i] / A[i, i];
3 end
4 foreach block after the first do
5 |  s ← MatrixVectorMultiplication(A[blockStart..blockEnd, blockEnd..n], x[blockEnd..n]);
6 |  foreach row i in current block do
7 |  |  x[i] ← b[i] − (s[i] / A[i, i]);
8 |  end
9 end
```

Algorithm 2: Block Back Substitution pseudocode

The loops from row 1-3 and row 6-8 and the matrix vector multiplication in row 5 compute the elements of x in a block fully independent of each other. Therefore these computations can be run in parallel and are thus suited to be run on a GPU. Note that when all blocks on the diagonal are of size 1 and thus only consist of the element on the diagonal itself, there is no parallelism and the algorithm becomes a normal back substitution.

2.3 Finding the blocks

To use the algorithm described in the previous section, the matrix needs to be analyzed first to find the blocks. The approach used is based on the level scheduling technique described in [Saaoo]. This technique works for all upper diagonal matrices and assigns each row a level. This level indicates on which other rows the row depends. Before the corresponding element x_i of row i can be found, the elements of x corresponding with rows with a lower level need to be found first. The level of a row is determined by its off-diagonal elements. The column index of each off-diagonal element is the row this element depends on, because it needs to be multiplied by the element that is found by solving the equation from this row. The level is then determined

by which of the rows has the highest level. Level 1 consists of the rows that have only a non-zero element on the diagonal and thus depend on no other rows. Algorithm 3 shows the pseudocode to assign each row its level.

input : Upper triangular matrix A of size n x n
output: Level for each row in A

```
1 foreach row i of A from n to 1 do
2 |   if The numbers of elements after the diagonal is o then
3 |   |   level[i] = 1;
4 |   end
5 |   else
6 |   |   level[i] = 1+max(level(j) for all A[i, j] such that j > i and A[i, j] > 0 );
7 |   end
8 end
```

Algorithm 3: Level assignment pseudocode

Because our solver goes through the matrix from bottom to top, we can start a new block every time a level is higher than the level of the previous row. In the ideal case, all blocks are as large as possible. This means that all rows with the same level are adjacent so that each block corresponds to a single level and that the blocks are sorted by this level in reverse. When this is not the case, the amount of blocks can quickly increase, thereby decreasing the parallelism and the performance of the solver. We are therefore mostly interested in matrices in which the blocks are optimal.

Finally this algorithm can be simplified because we look for blocks from bottom to top assigning each row to a block as we go. The algorithm checks all off-diagonal elements after the diagonal, but only the first is actually relevant for our purposes. If the first element depends on a previous row that has been assigned to the block we are currently assigning rows to, a new block has to be started regardless of the other elements. If the first element does depend only on a previous block, all subsequent elements will as well. These depend on lower rows than the first off-diagonal element, because their column index is higher. These rows are already assigned to the same block or another previous one.

Chapter 3

Implementation

The implementation of the solver uses NVIDIA CUDA. CUDA is a parallel programming platform created by NVIDIA that allows the use of their graphics processing units (GPUs) for general purpose computing. It is suited for data parallel computations: the same instructions are executed in parallel on different pieces of data. CUDA is available for different languages. The solver is implemented using C++. This section starts with an introduction of CUDA. This is followed by a description of the format of the sparse matrices the solver uses. This followed by the implementation details of the solver.

3.1 CUDA

In the context of CUDA the GPU is called a device. This device acts as a coprocessor on the system that executes a CUDA program. This system is called the host. The GPU has its own separate memory, so this has to be copied to and from the system. This memory is called the global memory.

The host launches special functions on the device. These functions are called kernels. When a kernel launches, the amount of threads that execute the kernel has to be specified. This is done by specifying a grid of thread blocks. This grid can have up to three dimensions. These thread blocks can also have up to three dimensions and they consist of threads. Every blocks gets a unique ID specifying its position in the grid and every thread gets a unique ID specifying its position in a block. The amount of threads and blocks is usually determined by the size of the data the kernel operates on and the IDs can be used as an index into the data.

Threads are scheduled in groups of 32. This group is called a warp. When a thread block has a size that is not a multiple of 32, the difference in threads between the amount of threads in the block and the nearest multiple of 32 still needs to be scheduled. These threads are then left idle. Thread blocks should therefore always contain a multiple of 32 threads.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        C[i] = A[i] + B[i];
}

 int main()
{
        ...
        int threads = 32;
        int blocks = N/32;

        // Kernel invocation with N/32 blocks and 32 threads per block
        <<<blocks, threads>>>(A, B, C);
        ...
}
```

Listing 3.1: Add two vectors A and B of size N and store result in C

$$\begin{bmatrix} 1.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 3.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 7.0 & 8.0 \\ 0.0 & 0.0 & 9.0 & 0.0 & 6.0 \end{bmatrix}$$

$$\begin{bmatrix} 1.0 & 4.0 & 0.0 \\ 2.0 & 3.0 & 0.0 \\ 5.0 & 7.0 & 8.0 \\ 9.0 & 6.0 & 0.0 \end{bmatrix} \begin{bmatrix} 1 & 2 & -1 \\ 2 & 3 & -1 \\ 1 & 4 & 5 \\ 3 & 5 & -1 \end{bmatrix}$$

Figure 3.1: Matrix and its Ellpack-Itpack representation

A simple example adding two vectors is shown in listing 3.1. In main() the kernel is invoked with a one-dimensional grid of N/32 one-dimensional blocks of 32 threads each. It is assumed that N is a multiple of 32. In the kernel VecAdd the index i is first calculated using the block and thread id in combination with the grid size. Finally A and B are added together and the result is stored in C.

3.2 Sparse Matrix Format

Because most of the elements in a sparse matrix are zero, several ways to store sparse matrices in a compressed format have been developed. The solver uses the Ellpack-Itpack format that is described in [Ell]. This format was chosen for its suitability for vector processing. It splits the matrix up in two *mxk* arrays. The first array contains only the nonzero elements of the matrix, with all zeros in between removed. The second array contains the column indices of all the elements. See Figure 3.1 for an example.

Note that *k* is determined by the row with the most amount of elements. In Figure 3.1 this is the third row. Rows with less than *k* elements are padded with zeros in the data array and with -1 in the column index

$$\begin{bmatrix} 1.0 & 2.0 & 5.0 & 9.0 & 4.0 & 3.0 & 7.0 & 6.0 & 0.0 & 0.0 & 8.0 & 0.0 \end{bmatrix}$$

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

Figure 3.2: Representation of the matrix in Figure 3.1 in memory.

array.

Our solver expects both the arrays to be stored in memory in column-major order. This means that elements of each column are contiguous in memory. See Figure 3.2 for how the two arrays from Figure 3.1 would be stored in memory. The reason for this choice is explained in the next section.

3.3 Matrix Vector Multiplication

The most computationally expensive part of the solver is the matrix vector multiplication. Every element of a row has to be multiplied by the corresponding vector element and these results have to be summed. This is done by using a thread per row. Every thread then loops over the elements of the row and at the end writes back the result when finished.

The matrices are stored in global memory. This memory is relatively slow. However, when consecutive threads in a warp access consecutive memory locations, these can be combined into a single memory access. This is called coalescing. If the matrices were stored in row-major order the situation shown in Figure 3.3 would happen. The threads access elements that are far apart in memory. This means coalescing is impossible. When storing the matrices in column-major order, the access pattern changes to the one shown in Figure 3.4. Now the threads access elements that are contiguous in memory. These memory accesses can therefore be coalesced into a single access.

For every element of a row each thread performs two steps: multiply the matrix element with the corresponding vector element and add the results to the result so far. Because the solver operates on floating point numbers, rounding takes place after every step. This can result in a loss of precision. To lessen this, the solver uses fused multiply-add as described in [WFF]. This combines the addition and multiplication and performs only a single rounding step. This makes the results more accurate.

To process every row, a simple approach would be to take a set amount of threads per block and then use enough blocks to so that the amount of thread matches the amount of rows. This however means that matrices that exceed the maximum amount of threads cannot be used. Therefore the kernel uses a grid stride loop as explained in [Har]. Every thread gets its index assigned as shown in Section 3.1. This is the first row the thread processes. After the row has been computed, this index is incremented by the full amount of threads

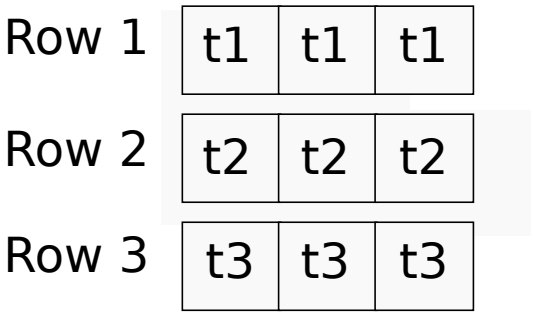

Figure 3.3: Matrix vector multiplication without coalescing

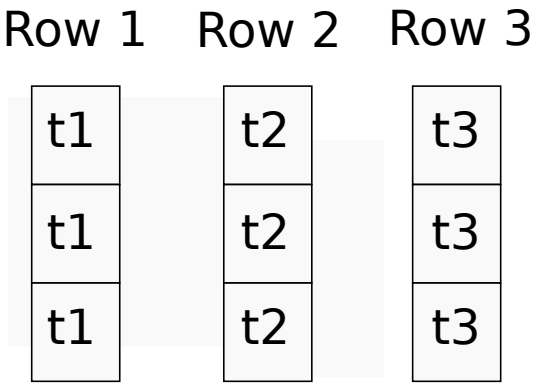

Figure 3.4: Matrix vector multiplication with coalescing

used to launch the kernel. This is then the index of the next row the thread computes. This continues until all rows have been processed. This allows for matrices of arbitrary size and for kernels to be launched with an arbitrary amount of thread blocks.

3.4 Other Kernels

The other two kernels are relatively straightforward. The first is used to compute the elements of x corresponding to the first block. It takes two vectors, divides the first by the second and writes back the results. It uses a thread per element and uses grid stride loops so vectors of arbitrary lengths are supported. Memory accesses are fully coalesced as each subsequent thread accesses the subsequent vector element.

The second kernel is similar and computes the elements of x corresponding to the other blocks. It takes a third vector and subtracts this from the first, before the division takes place.

3.5 The Full Solver

The full solver works by first finding the blocks. This is done by the CPU in the way described in Section 2.3. It also immediately checks if the matrix supplied is upper diagonal, because otherwise the solver would

return an incorrect result. Matrices with zero elements on the diagonal are also rejected, as these would result in a division by zero.

All the data is then copied to the GPU. To ensure maximum coalescing, the memory address should be properly aligned. All accesses can only be fully coalesced if the first read of a warp is done on an address that is a multiple of 2^n. This means every column of the two 2-dimensional arrays containing the matrix needs to start at a memory address that is a multiple of 2^n. To achieve this, a buffer for every block of the matrix is allocated by the solver. Every column is padded to ensure the next one starts at a memory address that is a multiple of 2^n. The matrix is then copied block for block to these buffers.

The diagonal is copied to its own buffer. This is done because the diagonal is not involved in the matrix vector multiplication. If its elements were in the same buffers as the blocks, this multiplication would have to start from the second column. This would result in reads that are not properly aligned for coalescing.

Then the first block is used to compute the last elements of x. Following this the solver loops over every block and performs the matrix vector multiplication and computes the next elements of x. When the solver is done, it copies the result back to the host.

All kernels use thread blocks of 256 threads. The amount of blocks is then set to how many are needed to have enough threads for all the work each kernel needs to do. If this amount exceeds the maximum amount of blocks, the maximum is used instead. With the grid stride loops this still give the correct results.

It is common to solve multiple equations with the same matrix, but with different vectors. The solver therefore also supports solving with multiple vectors at the same time. As the computations for different vectors are completely independent of each other, streams are used. A stream is a sequence of operations that execute on the GPU in the order they are issued. Multiple of these streams can be overlapped and this means executing different kernels in parallel. All kernels for a given vector are therefore executed on their own stream. This way the kernels that are independent of each other can be overlapped.

Chapter 4

Experiments

To measure the performance of our back substitution solver, benchmarks have been performed to test the performance. These same benchmarks have been performed using the sparse matrix library cuSPARSE that also make uses of CUDA. First cuSPARSE is described in more detail. Next we discuss the generation of the test matrices. Finally all the different benchmarks are described.

4.1 cuSPARSE

cuSPARSE is a closed source library that is part of the CUDA toolkit. For the benchmarks we used the triangular solver it provides. Like our solver, an analysis of the matrix needs to be performed before the solver can be used. As cuSPARSE supports independent stream, these are used to overlap computations when solving multiple vectors for the same matrix.

4.2 Test Matrix Generation

To perform our benchmarks, we need suitable matrices. The matrices used were all obtained from the University of Florida sparse matrix collection [DH]. For these matrices we used level scheduling, as described in Section 2.3, followed by ILU(0) factorization. The resulting U factors are used in the experiments.

The matrices used are listed in Table 4.1. They were chosen to have a variety of different matrix and block sizes and different amounts of blocks. The table lists the names of the matrices that were transformed, the blocks in the matrices and the amount of rows. It also includes the number of elements per row when converting the matrices to the Ellpack-Itpack format.

Matrix	Rows	Blocks	Row Length (ELL)
1138_bus	1138	21	17
bodyy6	19366	552	8
nopoly	10774	875	11
swang-2	3169	164	10
t2dal	4257	296	8
bcsstm39	46772	1	1
nemeth10	9506	9505	35
parabolic_fem	528825	7	7
Rail_1357	1357	7	11
shallow_water1	81920	1016	4
ASIC_320ks	312671	35	190

Table 4.1: Matrices used for benchmarking.

4.3 Benchmarks

The solvers were run on all matrices listed in 4.1 with 1, 5, 50 and 100 vectors. The time measured includes the analysis of the matrices and the solving of the matrices on the GPU. Transfer time to and from GPU memory was not measured. To get some more insight in the performance, the analysis section of our solver and of cuSPARSE were also benchmarked separately. Finally, because of the importance of the matrix vector multiplication in our solver, our matrix vector multiplication kernel was also benchmarked against the cuSPARSE implementation for a subset of the matrices. To verify the results, all benchmarks were run multiple times. Subsequent runs gave similar results.

The benchmarks were run on two different cards with a different architecture. These were the GTX 480 and the GTX 980 Ti.

4.4 Benchmark Results

This section contains the results of the benchmarks. All times are given in milliseconds

4.4.1 Matrix Vector Multiplication

1138_bus						
	980Ti			480		
Vectors	MV kernel	cuSPARSE	cuSPARSE difference	MV kernel	cuSPARSE	cuSPARSE difference
1	0.033000	0.055000	+40.000000 %	0.073000	0.065000	-12.307692 %
5	0.054000	0.103000	+47.572816 %	0.122000	0.142000	+14.084507 %
50	0.239000	0.540000	+55.740741 %	0.360000	0.906000	+60.264901 %
100	0.375000	1.010000	+62.871287 %	0.651000	1.610000	+59.565217 %
ASIC_320ks						
	980Ti			480		
Vectors	MV kernel	cuSPARSE	cuSPARSE difference	MV kernel	cuSPARSE	cuSPARSE difference
1	1.590000	0.201000	-691.044776 %	2.820000	0.397000	-610.327456 %
5	7.180000	0.603000	-1090.713101 %	13.100000	1.360000	-863.235294 %
50	53.300000	5.020000	-961.752988 %	128.000000	12.100000	-957.851240 %
100	100.000000	9.980000	-902.004008 %	256.000000	24.100000	-962.240664 %
bcsstm39						
	980Ti			480		
Vectors	MV kernel	cuSPARSE	cuSPARSE difference	MV kernel	cuSPARSE	cuSPARSE difference
1	0.027000	0.065000	+58.461538 %	0.062000	0.090000	+31.111111 %
5	0.041000	0.118000	+65.254237 %	0.125000	0.225000	+44.444444 %
50	0.194000	0.609000	+68.144499 %	0.772000	1.680000	+54.047619 %
100	0.365000	1.150000	+68.260870 %	1.490000	3.310000	+54.984894 %
nemeth10						
	980Ti			480		
Vectors	MV kernel	cuSPARSE	cuSPARSE difference	MV kernel	cuSPARSE	cuSPARSE difference
1	0.046000	0.062000	+25.806452 %	0.168000	0.103000	-63.106796 %
5	0.076000	0.107000	+28.971963 %	0.554000	0.178000	-211.235955 %
50	0.406000	0.549000	+26.047359 %	4.520000	1.140000	-296.491228 %
100	0.771000	1.020000	+24.411765 %	8.880000	2.240000	-296.428571 %
parabolic_fem						
	980Ti			480		
Vectors	MV kernel	cuSPARSE	cuSPARSE difference	MV kernel	cuSPARSE	cuSPARSE difference
1	0.165000	0.202000	+18.316832 %	0.966000	0.586000	-64.846416 %
5	0.571000	0.788000	+27.538071 %	4.110000	2.240000	-83.482143 %
50	5.020000	7.360000	+31.793478 %	40.000000	20.800000	-92.307692 %
100	9.980000	14.600000	+31.643836 %	80.600000	41.400000	-94.685990 %

4.4.2 Solvers

1138_bus						
	980Ti			480		
Vectors	Back S. Solver	cuSPARSE	cuSPARSE difference	Back S. Solver	cuSPARSE	cuSPARSE difference
1	0.390000	2.950000	+86.779661 %	0.410000	5.390000	+92.393321 %
5	0.764000	3.090000	+75.275081 %	1.580000	4.510000	+64.966741 %
50	6.940000	5.120000	-35.546875 %	14.900000	9.690000	-53.766770 %
100	13.400000	7.410000	-80.836707 %	29.900000	14.600000	-104.794521 %
ASIC_320ks						
	980Ti			480		
Vectors	Back S. Solver	cuSPARSE	cuSPARSE difference	Back S. Solver	cuSPARSE	cuSPARSE difference
1	4.460000	14.700000	+69.659864 %	9.060000	18.100000	+49.944751 %
5	9.930000	15.900000	+37.547170 %	32.400000	21.300000	-52.112676 %
50	55.200000	21.400000	-157.943925 %	296.000000	57.500000	-414.782609 %
100	107.000000	32.500000	-229.230769 %	590.000000	97.600000	-504.508197 %
bcsstm39						
	980Ti			480		
Vectors	Back S. Solver	cuSPARSE	cuSPARSE difference	Back S. Solver	cuSPARSE	cuSPARSE difference
1	0.167000	1.790000	+90.670391 %	0.226000	4.470000	+94.944072 %
5	0.142000	1.770000	+91.977401 %	0.244000	3.990000	+93.884712 %
50	0.175000	4.390000	+96.013667 %	0.592000	7.650000	+92.261438 %
100	0.269000	6.070000	+95.568369 %	0.972000	11.200000	+91.321429 %
bodyy6						
	980Ti			480		
Vectors	Back S. Solver	cuSPARSE	cuSPARSE difference	Back S. Solver	cuSPARSE	cuSPARSE difference
1	5.350000	14.400000	+62.847222 %	7.960000	12.800000	+37.812500 %
5	9.960000	14.100000	+29.361702 %	34.800000	12.300000	-182.926829 %
50	176.000000	22.200000	-692.792793 %	367.000000	18.100000	-1927.624309 %
100	368.000000	23.000000	-1500.000000 %	664.000000	22.800000	-2812.280702 %
nemeth10						
	980Ti			480		
Vectors	Back S. Solver	cuSPARSE	cuSPARSE difference	Back S. Solver	cuSPARSE	cuSPARSE difference
1	766.000000	190.000000	-303.157895 %	453.000000	139.000000	-225.899281 %
5	404.000000	18.400000	-2095.652174 %	1880.000000	166.000000	-1032.530120 %
50	1950.000000	349.000000	-458.739255 %	18100.000000	193.000000	-9278.238342 %
100	4100.000000	40.900000	-9924.449878 %	36100.000000	294.000000	-12178.911565 %

nopoly						
	980Ti			480		
Vectors	Back S. Solver	cuSPARSE	cuSPARSE difference	Back S. Solver	cuSPARSE	cuSPARSE difference
1	14.400000	18.100000	+20.441989 %	14.100000	16.600000	+15.060241 %
5	27.300000	18.500000	-47.567568 %	72.200000	16.200000	-345.679012 %
50	272.000000	33.300000	-716.816817 %	559.000000	45.400000	-1131.277533 %
100	545.000000	48.500000	-1023.711340 %	1100.000000	72.500000	-1417.241379 %
parabolic_fem						
	980Ti			480		
Vectors	Back S. Solver	cuSPARSE	cuSPARSE difference	Back S. Solver	cuSPARSE	cuSPARSE difference
1	1.190000	14.800000	+91.959459 %	4.840000	27.700000	+82.527076 %
5	1.570000	15.800000	+90.063291 %	7.790000	32.200000	+75.807453 %
50	6.500000	32.500000	+80.000000 %	40.400000	83.200000	+51.442308 %
100	12.000000	50.300000	+76.143141 %	77.100000	138.000000	+44.130435 %
rail_1357						
	980Ti			480		
Vectors	Back S. Solver	cuSPARSE	cuSPARSE difference	Back S. Solver	cuSPARSE	cuSPARSE difference
1	0.110000	2.660000	+95.864662 %	0.173000	3.850000	+95.506494 %
5	0.268000	2.840000	+90.563380 %	0.671000	4.290000	+84.358974 %
50	2.160000	4.840000	+55.371901 %	6.250000	9.350000	+33.155080 %
100	4.240000	5.310000	+20.150659 %	12.300000	12.600000	+2.380952 %
shallow_water						
	980Ti			480		
Vectors	Back S. Solver	cuSPARSE	cuSPARSE difference	Back S. Solver	cuSPARSE	cuSPARSE difference
1	11.300000	26.200000	+56.870229 %	13.000000	21.000000	+38.095238 %
5	31.200000	31.200000	+0.000000 %	76.000000	30.300000	-150.825083 %
50	366.000000	90.300000	-305.315615 %	624.000000	136.000000	-358.823529 %
100	550.000000	28.900000	-1803.114187 %	1200.000000	258.000000	-365.116279 %
swang2						
	980Ti			480		
Vectors	Back S. Solver	cuSPARSE	cuSPARSE difference	Back S. Solver	cuSPARSE	cuSPARSE difference
1	2.750000	4.840000	+43.181818 %	2.660000	6.010000	+55.740433 %
5	4.910000	4.170000	-17.745803 %	10.600000	6.080000	-74.342105 %
50	53.600000	7.800000	-587.179487 %	105.000000	7.280000	-1342.307692 %
100	105.000000	7.630000	-1276.146789 %	210.000000	9.000000	-2233.333333 %

t2dal						
	980Ti			480		
Vectors	Back S. Solver	cuSPARSE	cuSPARSE difference	Back S. Solver	cuSPARSE	cuSPARSE difference
1	4.120000	7.910000	+47.914033 %	4.480000	7.380000	+39.295393 %
5	9.290000	7.930000	-17.150063 %	22.300000	7.490000	-197.730307 %
50	91.600000	9.190000	-896.735582 %	175.000000	9.640000	-1715.352697 %
100	184.000000	10.100000	-1721.782178 %	361.000000	11.900000	-2933.613445 %

4.4.3 Analysis time

Matrix	Find Blocks - 980Ti	cuSPARSE analysis - 980Ti	Find Blocks - 480	cuSparse Analysis 480
1138_bus	0.027	2.49	0.0229	5.23
bodyy6	0.208	30	0.245	10.4
nopoly	0.056	10.4	0.139	14
Swang-2	0.068	4.53	0.0473	4.51
t2dal	0.0728	6.33	0.0616	6.49
bcsstm39	0.146	2.53	0.202	4.53
nemeth10	0.405	108	0.418	127
parabolic_fem	3.27	49.8	5.09	26.8
Rail_1357	0.0155	2.29	0.0193	3.97
shallow_water1	0.341	30.3	0.709	17
ASIC_320ks	3.32	21.3	10.5	17.3

Chapter 5

Discussion

Focusing only on our solver and our matrix vector multiplication kernel the results show that the GTX 980 Ti performs better than the GTX 480. This was the expected result as the 980 Ti is a newer card with more compute power than the 480.

Looking at the performance of our matrix vector multiplication kernel shows an interesting result. The performance is more dependent on how big the rows are then on how many rows there are. The matrix vector multiplication is fastest for the matrix bcsstm39. This is the matrix with the least elements per row as there are only elements on the diagonal. The kernel performs slowest for ASIC_320ks. This matrix has 190 elements per row which is the most of all the test matrices. It is however not the matrix with the most rows. That is the matrix parabolic_fem which has only 7 elements per row. This is logical when we look at how our kernel works. Every extra row is computed by a new thread, which can therefore be done in parallel. So the time increase from an extra row is limited. More elements per row however, means that each thread has to loop over extra elements. This increases the total compute time.

Looking at just our solver, a reasonable assumption would be that it should perform faster for matrices with fewer blocks. In Table 5.1 the matrices have been sorted by performance of our solver for 5 vectors on the GTX 980 Ti. In general, it shows the assumption to be true. With more blocks, the solver performs worse.

There are two notable exceptions though. Despite rail_1357 and parabolic_fem having the same amount of blocks, the solver is slower for parabolic_fem. It is even slower than for 1138_bus, even though this matrix has more blocks. The probable cause is that parabolic_fem is the largest matrix in the test set. This means the blocks are also much larger. This results in different performance of the kernels. When the amount of rows is small, the threads can all be run at the same time on the GPU. When the blocks become too large however, the amount of threads exceeds the maximum amount of threads that can be active at once. The maximum amount of threads that can be run on a GPU at once is determined by the amount of multiprocessors and the

Time 5	Vectors Matrix	Rows	Blocks	Row Length (ELL)
0.085	bcsstm39	46772	1	1
0.268	Rail_1357	1357	7	11
0.764	1138_bus	1138	21	17
1.57	parabolic_fem	528825	7	7
4.91	swang-2	3169	164	10
9.29.2	t2dal	4257	296	8
9.93	ASIC_320ks	312671	35	190
9.96	bodyy6	19366	552	8
27.3	nopoly	10774	875	11
31.2	shallow_water1	81920	1016	4
404	nemeth10	9506	9505	35

Table 5.1: Sorted by time for back substitution solver for 5 vector on GTX 980 Ti.

maximum amount of threads per multiprocessor. The GTX 980 Ti has 22 multiprocessors and a maximum of 2048 threads per multiprocessor. This means a total of 22 * 2048 = 45056 threads. All blocks in the matrix parabolic_fem exceed this amount while none of the blocks in rail_1357 do. The rows of the blocks in parabolic_fem can therefore not be processed all at once while the blocks in rail_1357 can.

ASICK_320ks is the other exception. It has only a single block that exceeds the maximum amount of simultaneously active threads, so this is unlikely to be the cause. As mentioned above however, it has a high amount of elements per row when compared to the other matrices. This results in a slower matrix vector multiplication and therefore slower solver performance.

The table also shows a worst case and best case scenario for our solver. The best case is the matrix bcsstm39. This is a matrix with only elements on the diagonal and has therefore only a single block. Thus equations with this matrix can be solved very quickly. The worst case is the matrix nemeth10. All but one of the blocks of this matrix have a size of 1. The solver therefore performs significantly worse on this matrix than on the others. This also makes it very clear that the amount of blocks determines the the performance more than the amount of rows in a matrix as bcsstm39 has many rows more than nemeth10.

When comparing the performance of our matrix vector multiplication kernel and the one provided by cuSPARSE on the GTX 980 Ti, we see that our kernel performs well when the amount of elements per row remains small. With the matrix ASICK_320ks however, our kernel is much slower than cuSPARSE. cuSPARSE appears to scale better with bigger rows.

On the GTX 480 however, cuSPARSE becomes faster for certain matrices. We keep the advantage when it comes to smaller matrices. When looking at larger matrices however, the performance of our kernel decreases more when moving from the GTX 980 Ti to the GTX 480 than the performance of cuSPARSE does. This could be caused by the maximum amount of threads that each GPU can run at once. This amount is larger for the GTX 980 Ti than for the GTX 480. For our kernel this means that less rows are computed in parallel on the GTX 480 when using larger matrices as it uses a thread per row. This then results in bigger performance

decreases then the ones cuSPARSE suffers from. As the source to cuSPARSE is not available, it is difficult to determine why this is the case. The biggest difference in performance occurs again when benchmarking with ASIC_320ks.

When comparing the performance of our solver with cuSPARSE, it is often the case that only for 1 and sometimes for 5 vectors our solver is faster. The reason becomes clear when we look at the performance of the analysis functions. Ours is faster than cuSPARSE's. This analysis function however needs to be run only once for a matrix. After that cuSPARSE is faster when it comes to actually solving the equation. So when the amount of vector increases cuSPARSE eventually becomes faster than our solver.

There are a few exceptions: bcsstm39, rail_1357 and parabolic_fem. These are matrices that are favourable for our solver as they have few blocks. Especially bccstm39 and parabolic_fem are suited, because they have a large amount of rows. However the speedup still becomes smaller when the amount of vectors increases for parabolic_fem and rails_1357. Even for the most suitable matrices, it appears that our solve time is larger than cuSPARSE's. So if the amount of vectors is increased enough, cuSPARSE will become faster than our solver for these matrices at some point. This is not the case for bccstm39 for which the solver uses only a single kernel launch.

ASICK_320ks might also appear suitable for our solver as it is a large matrix with relatively few blocks. However, due to the slow matrix vector multiplication with this matrix, cuSPARSE is faster for higher amounts of vectors. So to improve our solver, a priority should be to improve the scalability of the matrix vector kernel for bigger rows.

When moving from the GTX 980 Ti to the GTX 480 the performance of our solver decreases faster than the performance of cuSPARSE. This means that for instances were our solver is faster than cuSPARSE on the GTX 980 Ti, cuSPARSE is faster on the GTX 480. This can be seen for the matrix bodyy6 when solving with 5 vectors.

Chapter 6

Conclusions

In this thesis we have described a solver for upper triangular sparse matrices with single diagonal blocks on the diagonal. Such matrices can arise as U factor when using preconditioners with an iterative method. The algorithm used by the solver performs a back substitution on multiple rows at once by grouping rows together that are independent of each other. As these rows can be solved independently of each other, these computations have been implemented using CUDA to take advantage of the power of GPUs which specialize in doing many computations in parallel.

The goal was to have a solver that is faster for these types of sparse matrices than libraries that have to solve with arbitrary matrices. To this end benchmarks have been run and the performance has been compared with that of the cuSPARSE library. The result found was that the solver is faster only with a low amount of vectors or only with very suitable matrices. This is due to the fact that the analysis phase of our solver is faster than that of cuSPARSE. cuSPARSE is then faster at the actual solving. As the analysis has to only been done once, cuSPARSE will become faster at some point when enough equations with different vectors are solved. When this happens depends on the suitability of the matrix. The matrices most suitable for our solver are matrices with few blocks and few elements per row. Only these matrices were still faster when solving with a hundred different vectors, although even with these the difference between the two solvers lessens when the amount of vectors increases.

This means our goal is only achieved under very specific circumstances. Because in iterative methods matrix vector equations have to be solved every iteration with new vectors, the solver as it is now is unsuitable for use in these methods. Even when very suitable matrices can be guaranteed to result from preconditioning, the amount of iterations will have to remain limited as with enough different equations cuSPARSE eventually becomes faster again.

However, there appears to still be room for improvement. An important part of the solver is the matrix vector

multiplication. cuSPARSE shows that these can be done faster than we do currently, especially when it comes to matrices that have more elements per row. Future work should be done on improving this, as when the solving is done faster, it will take longer for cuSPARSE to become faster. This then allows for more iterations or the use of matrices that consist of more blocks.

Finally, as our solver currently only works on upper triangular matrices, only U factors resulting from preconditioners can be used with our solver. To use the solver with L factors as well, it will have to be adapted to work with lower triangular matrices. To do this the same algorithm can be used, but it has to start from the top of the matrix and work to the bottom.

References

[BPC+16] Daniele Buono, Fabrizio Petrini, Fabio Checconi, Xing Liu, Xinyu Que, Chris Long, and Tai-Ching Tuan. Optimizing sparse matrix-vector multiplication for large-scale data analytics. In *Proceedings of the 2016 International Conference on Supercomputing*, page 37. ACM, 2016.

[DH] Tim Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. `https://www.cise.ufl.edu/research/sparse/matrices`. Accessed: 2016-08-14.

[Ell] cuSPARSE - Ellpack-Itpack format. `docs.nvidia.com/cuda/cusparse/index.html#ellpack-itpack-format-ell`. Accessed: 2016-08-13.

[GMSW84] Philip E Gill, Walter Murray, Michael A Saunders, and Margaret H Wright. Sparse matrix methods in optimization. *SIAM Journal on Scientific and Statistical Computing*, 5(3):562–589, 1984.

[Har] Mark Harris. CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops. `devblogs.nvidia.com/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops`. Accessed: 2016-08-14.

[Saa00] Yousef Saad. *Iterative Methods for Sparse Linear Systems*, chapter 11, pages 344–348. PWS Publishing Company, 3rd edition, 2000.

[WFF] Nathan Whitehead and Alex Fit-Florea. Floating Point and IEEE 754 Compliance for NVIDIA GPUs. `docs.nvidia.com/cuda/floating-point/index.htmll`. Accessed: 2016-08-14.