Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU

Maxim Naumov
NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050

Abstract

A novel algorithm for solving in parallel a sparse triangular linear system on a graphical processing unit is proposed. It implements the solution of the triangular system in two phases. First, the analysis phase builds a dependency graph based on the matrix sparsity pattern and groups the independent rows into levels. Second, the solve phase obtains the full solution by iterating sequentially across the constructed levels. The solution elements corresponding to each single level are obtained at once in parallel. The numerical experiments are also presented and it is shown that the incomplete-LU and Cholesky preconditioned iterative methods, using the parallel sparse triangular solve algorithm, can achieve on average more than 2× speedup on graphical processing units (GPUs) over their CPU implementation.

1 Introduction

The solution of sparse triangular linear systems is an important building block of many numerical linear algebra algorithms. It arises in the direct solution of linear systems and least squares problems [22]. It also arises in splitting based iterative schemes, such as Gauss-Seidel, and in the preconditioning of iterative methods using incomplete-LU and Cholesky factorizations [18]. In the former case the solution is often performed once, while in the latter it is computed multiple times for a single or multiple right-hand-sides.

NVIDIA Technical Report NVR-2011-001, June 2011.

Although the forward and back substitution is an inherently sequential algorithm for dense triangular systems, the dependencies on the previously obtained elements of the solution do not necessarily exist for the sparse triangular systems. For example consider a diagonal matrix, where the solution elements corresponding to all rows are independent and can be computed at once in parallel. The realistic sparse triangular matrices often have sparsity patterns that can also be exploited for parallelism.

The parallel solution of sparse triangular linear systems has been considered by many authors with two overarching strategies. The first strategy takes advantage of the lack of dependencies in the forward and back substitution due to the sparsity of the matrix directly. It often consists of a preprocessing step where the sparsity pattern is analysed and a solve step that uses the computed information to exploit available parallelism [2, 8, 13, 14, 17, 19, 23]. The second strategy expresses the triangular matrix as a product of sparse factors. The different partitioning of the triangular matrix into these factors results in different numerical algorithms [1, 10, 12, 16]. A related work for parallel solution of dense and banded triangular linear systems has also been done in [9, 5, 20].

In this paper we focus on the situation where we need to solve the same linear system multiple times with a single right-hand-side. For example, this situation arises in the preconditioning of iterative methods using incomplete-LU and Cholesky factorizations. We pursue the first strategy described above and split the solution process into two phases. The *analysis* phase builds the data dependency graph that groups independent rows into levels based on the matrix sparsity pattern. The modified topological sort, breadth-first-search and other graph search algorithms can be used to construct this directed acyclic graph [4, 6, 7]. The *solve* phase iterates across the constructed levels one-by-one and computes all elements of the solution corresponding to the rows at a single level in parallel. Notice that by construction the rows within each level are independent of each other, but are dependent on at least one row from the previous level. The *analysis* phase needs to be performed only once and is usually significantly slower than the *solve* phase, which can be performed multiple times.

The sparse triangular linear system solve is implemented using CUDA parallel programming paradigm [11, 15, 21], which allows us to explore the computational resources of the graphical processing unit (GPU). This new algorithm, the well studied sparse matrix-vector multiplication [3, 24] as well as other standard sparse linear algebra operations are exposed as a set of routines in the CUSPARSE library [25]. Also, it is worth mentioning here that the corresponding dense linear algebra operations are exposed in the CUBLAS library [25].

Although, the parallelism available during the *solve* phase depends highly on the sparsity pattern of the triangular matrix at hand. In the numerical

experiments section it will be shown that in an iterative scheme the CUSPARSE library parallel implementation of the sparse triangular solve can outperform the MKL algorithm [26]. Moreover, it will be shown that the incomplete-LU and Cholesky preconditioned iterative methods can achieve on average more than 2× speedup using the CUSPARSE and CUBLAS libraries on the GPU over their MKL implementation on the CPU.

Since the solution of the lower and upper triangular linear systems is very similar, we focus only on the former in the next sections, where we describe the *analysis* and *solve* phases of the algorithm as well as its implementation.

2 Analysis and Solve Phases

We are interested in solving the linear system

$$L\mathbf{x} = \mathbf{f} \tag{1}$$

where $L \in \mathbb{R}^{n\times n}$ is a nonsingular lower triangular matrix and $\mathbf{x}, \mathbf{f} \in \mathbb{R}^n$ are the solution and right-hand-side vectors, respectively. In further discussion, we denote the elements of the lower triangular coefficient matrix $L = [l_{ij}]$, with $l_{ij} = 0$ for $i < j$.

We can represent the data dependencies in the solution of a lower triangular linear system as a directed graph, where the nodes represent rows and the arrows represent the data dependencies between them. This directed graph is constructed so that there is an arrow from node j to node i if there is an element $l_{ij} \neq 0$ present in the matrix. Notice that because the matrix is triangular there are no circular data dependencies in it, consequently there are no cycles in the graph. Also, notice that because the matrix is nonsingular $l_{ii} \neq 0$ for $i = 1, \ldots, n$, in other words, each row contains at least one non-zero element on the matrix main diagonal.

Let us consider the following linear system as an example

$$\begin{pmatrix} l_{11} & & & & & & & & \\ & l_{22} & & & & & & & \\ & & l_{33} & & & & & & \\ l_{41} & & & l_{44} & & & & & \\ l_{51} & & & & l_{55} & & & & \\ & l_{62} & & & & l_{66} & & & \\ & & l_{73} & & & & l_{77} & & \\ & & & l_{84} & l_{85} & & & l_{88} & \\ & & & l_{94} & l_{95} & & & & l_{99} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ f_3 \\ f_4 \\ f_5 \\ f_6 \\ f_7 \\ f_8 \\ f_9 \end{pmatrix} \tag{2}$$

The directed acyclic graph (DAG) illustrating the data dependencies in the lower triangular coefficient matrix in (2) is shown in Fig. 1. Notice that even though the sparsity pattern in (2) might look sequential at first glance, there is plenty of parallelism to be explored in it.

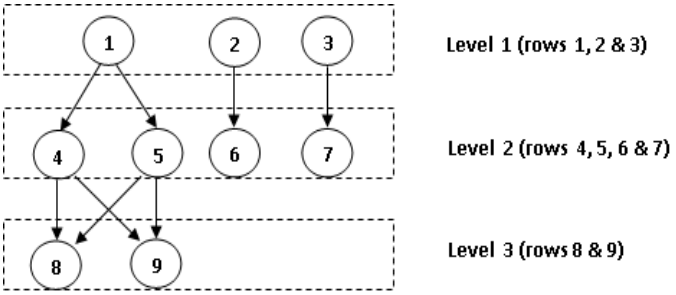


Figure 1: The data dependency DAG of the lower triangular linear system

In practice we do not need to construct the data dependency DAG because it is implicit in the matrix. It can be traversed using for example a modified breadth-first-search (BFS) shown in Alg. 1. Notice that in this algorithm the node's children are visited only if they have no data dependencies on the other nodes. The independent nodes are grouped into levels, which are shown with dashed lines in Fig. 1. This information is passed to the *solve* phase, which can process the nodes belonging to the same level in parallel.

Algorithm 1 Analysis Phase

```
1: Let n and e be the matrix size and level number, respectively.
2: e ← 1
3: repeat                                   ▷ Traverse the Matrix and Find the Levels
4:     for i ← 1, n do                                          ▷ Find Root Nodes
5:         if i has no data dependencies then
6:             Add node i to the list of root nodes.
7:         end if
8:     end for
9:     for i ∈ the list of root nodes do                        ▷ Process Root Nodes
10:        Add node i to the list of nodes on level e.
11:        Remove the data dependency on i from all other nodes.
12:    end for
13:    e ← e + 1
14: until all nodes have been processed.
```

In the *solve* phase we can explore the parallelism available in each level using multiple threads, but because the levels must be processed sequentially one-by-one, we must synchronize all threads across the level boundaries as shown in Alg. 2.

Algorithm 2 Solve Phase

```
1: Let k be the number of levels.
2: for e ← 1, k do
3:     list ← the sorted list of rows in level e.
4:     for row ∈ list in parallel do                ▷ Process a Single Level
5:         Compute the element of the solution corresponding to row.
6:     end for
7:     Synchronize threads.                         ▷ Synchronize between Levels
8: end for
```

In the next section we focus on the details of the implementation of the *analysis* and *solve* phases using CUDA parallel programming paradigm.

3 Implementation on the GPU

We assume that the matrix and all the intermediate data structures are stored in the device (GPU) memory, with the exception of a small control data structure stored in the host (CPU) memory. Also, we assume that the matrices are stored in the compressed sparse row (CSR) storage format [18].

For example, the coefficient matrix in (2) is stored as

$$
\begin{aligned}
rowPtr &= \begin{pmatrix} 1 & 2 & 3 & 4 & 6 & 8 & 10 & 12 & 15 & 18 \end{pmatrix} \\
colInd &= \begin{pmatrix} 1 & 2 & 3 & 1 & 4 & 1 & 5 & 2 & 6 & 3 & 7 & 4 & 5 & 8 & 4 & 5 & 9 \end{pmatrix} \\
Val &= \begin{pmatrix} l_{11} & l_{22} & l_{33} & l_{41} & l_{51} & l_{55} & l_{62} & l_{66} & l_{73} & l_{77} & l_{84} & l_{85} & \dots & l_{99} \end{pmatrix}
\end{aligned}
$$

The *analysis* phase generates a set of levels, the sorted list of rows belonging to every level and a small control data structure that we call *chain*.

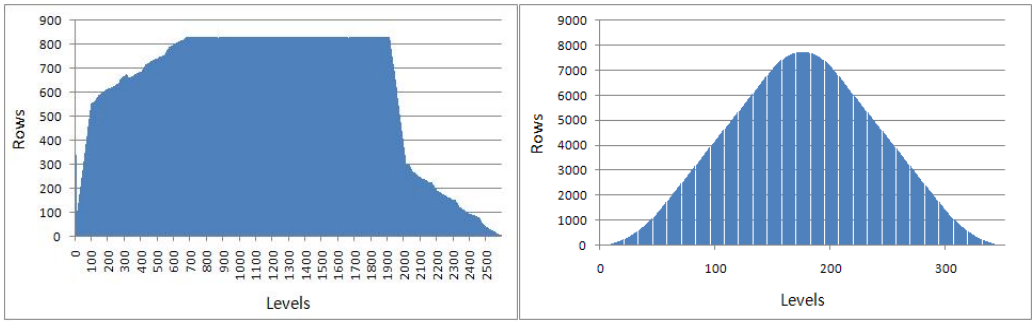
The first two data structures are obtained by traversing the matrix to find the root nodes, the rows that do not have data dependencies, and grouping them into levels. In practice to find the root nodes, we do not need to visit all rows at every iteration as shown in Alg. 1, because we can keep track of a short list of root candidates and visit only them, in all but the first iteration. This approach requires us to keep two separate buffers for storing the root nodes. The first buffer is used for reading the current root nodes, while the second is used for

writing the next level root nodes. Notice that in order to avoid copying memory between these buffers, we can flip-flop the corresponding pointers at the end of every iteration.

Let us now understand the purpose of the *chain* data structure, assuming that we have already obtained a set of levels and the sorted list of rows belonging to every level. Recall that we can explore the parallelism available in each level using a single or multiple CUDA thread blocks, but we must synchronize all threads across the level boundaries.

On one hand, if we are using a single thread block to process all levels, then we can use *__syncthreads()* to synchronize across levels, but we must assign the threads in a cyclic fashion between the rows of each level. On the other hand, if we are using multiple thread blocks, we can assign each thread to a single row, but we must return to the host (CPU) and then launch a new kernel to process the following level.

There is a tradeoff between parallelism and light-weight synchronization above. The former approach is well suited for sparsity patterns with limited parallelism, while the latter performs well when there are thousands of rows belonging to a single level. The realistic sparsity patterns however often show a mix of levels with a few and many rows. For example, the distribution of rows into levels for the incomplete-LU lower and upper triangular factors of the matrix atmosmodd from Tab. 2 is shown in Fig. 2.



(a) Lower triangular factor (b) Upper triangular factor

Figure 2: The distribution of rows into levels for atmosmodd matrix

Therefore, we would like to combine these two approaches. It turns out that in practice the distribution of rows into levels is often such that there are prolonged groups of levels with a few rows in them. Hence, if there exist consecutive levels, such that each of them contains only enough parallelism for

a single thread block, we group these levels into a *chain*. This chain of levels is processed with a single kernel call and inside of it the synchronization across the levels is performed using *__syncthreads()*. On the other hand, the synchronization across different chains is performed by returning to the host and then launching a new kernel.

The *chain* data structure allows us to significantly reduce the number of kernel calls without sacrificing parallelism. For example, the ratio between the number of levels and chains, in other words, the ratio between the number of kernel calls without and with the *chain* data structure for the matrix atmosmodd is shown in Tab. 1. Notice that for the lower triangular factor we are making almost 8× less kernel calls with the new approach.

Matrix	Lower triangular factor	Upper triangular factor
atmosmodd	7.76	1.05

Table 1: Ratio between number of levels and chains for atmosmodd matrix

Finally, the resulting *analysis* phase pseudo-code that obtains all three data structures is shown in Alg. 3 and 4.

Algorithm 3 Analysis Phase (Part 1)

Let b and e be the number of thread blocks and level number, respectively.
Let $levelInd[]$ contain the list of sorted rows in each level.
Let $levelPtr[]$ contain the starting index (into the array $levelInd$) of each level (and an extra element to indicate the end of the last level).
Let $chainPtr[]$ contain the starting index (into the array $levelPtr$) of each chain (and an extra element to indicate the end of the last chain).
Let $rRoot[]$ and $wRoot[]$ be the read and write root buffers, respectively.
Let $cRoot[]$ be the list of candidate nodes for being roots.

$e \leftarrow 1$
FIND_ROOTS$\lll b, \ldots \ggg(e, rRoot)$
repeat ▷ Find the Levels
    ANALYSE$\lll b, \ldots \ggg(e, rRoot, cRoot, levelInd, levelPtr, chainPtr)$
    FIND_ROOTS_IN_CANDIDATES$\lll b, \ldots \ggg(cRoot, wRoot)$
    Flip-flop $rRoot[]$ and $wRoot[]$ buffer pointers.
    $e \leftarrow e + 1$
until all nodes have been processed.

Algorithm 4 Analysis Phase (Part 2)

15: **procedure** FIND_ROOTS ▷ CUDA Kernel
16: ...
17: **end procedure**

18: **procedure** ANALYSE ▷ CUDA Kernel
19: ...
20: **end procedure**

21: **procedure** FIND_ROOTS_IN_CANDIDATES ▷ CUDA Kernel
22: ...
23: **end procedure**

The output of the *analysis* phase are the arrays *chainPtrHost*, *levelPtr* and *levelInd* that have the beginning and end of the chains, levels and the list of sorted rows belonging to every level, respectively. The array *chainPtrHost* determines the properties and the number of kernels to be launched in the *solve* phase and therefore must be present in the host (CPU) memory. It is usually a relatively short array when compared to the matrix size and is the only data structure present in the host memory.

For example, for (2) these arrays are

$$\begin{aligned} chainPtrHost &= \begin{pmatrix} 1 & 4 \end{pmatrix} \\ levelPtr &= \begin{pmatrix} 1 & 4 & 8 & 10 \end{pmatrix} \\ levelInd &= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix} \end{aligned}$$

Notice that in this particular example there are only a few rows belonging to every level, so that all of the levels are linked into a single chain, and consequently can be processed with a single kernel in the *solve* phase.

The *solve* phase accepts as an input a set of levels, the sorted list of rows belonging to every level and the *chain* data structure. It determines the optimal number of thread blocks b needed to process each chain and launches a single $b = 1$ or multiple $b > 1$ thread block kernels in a loop until all chains have been processed. Notice that multiple thread blocks kernel always processes a single level, while a single thread block kernel can process one or more levels.

It is worth mentioning that in general rows might be grouped into levels without preserving their original ordering, in other words, an earlier row can be assigned to a latter level (consider for example the coefficient matrix in (2) augmented with an extra row with a single diagonal element in it). Although

we do not have control over assignment of rows across levels, the sorting of rows within a level that is done in the *analysis* phase improves the coalescing of memory reads without affecting parallelism.

Finally, the resulting *solve* phase pseudo-code is shown in Alg. 5.

Algorithm 5 Solve Phase

Let b and k be the number of thread blocks and chains, respectively.
Let $levelInd[]$ contain the list of rows in each level.
Let $levelPtr[]$ contain the starting index (into the array $levelInd$) of each level (and an extra element to indicate the end of the last level).
Let $chainPtrHost[]$ contain the starting index (into the array $levelPtr$) of each chain (and an extra element to indicate the end of the last chain).

for $i \leftarrow 0, k$ **do** ▷ Process the Chains
    $start \leftarrow chainPtrHost[i]$
    $end \leftarrow chainPtrHost[i+1]$
    if single block is enough **then**
        PROCESS_LEVEL_SINGLEBLOCK$\lll 1, \ldots \ggg(start, end)$
    else
        PROCESS_LEVEL_MULTIBLOCK$\lll b, \ldots \ggg(start)$
    end if
end for

procedure PROCESS_LEVEL_SINGLEBLOCK$(start, end)$ ▷ CUDA Kernel
    for $e \leftarrow start, end$ **do**
        ...
        __syncthreads()
    end for
end procedure

procedure PROCESS_LEVEL_MULTIBLOCK(e) ▷ CUDA Kernel
    ...
end procedure

4 Numerical Experiments

In this section we study the performance of the parallel sparse triangular solve as a standalone algorithm and as a part of a preconditioned iterative method.

We use twelve matrices selected from The University of Florida Sparse Matrix Collection [27] in our numerical experiments. The seven symmetric positive definite (s.p.d.) and five nonsymmetric matrices with the respective number of rows (m), columns (n=m) and non-zero elements (nnz) are grouped and shown according to their increasing order in Tab. 2.

#	Matrix	m,n	nnz	s.p.d.	Application
1.	offshore	259,789	4,242,673	yes	Geophysics
2.	af_shell3	504,855	17,562,051	yes	Mechanics
3.	parabolic_fem	525,825	3,674,625	yes	General
4.	apache2	715,176	4,817,870	yes	Mechanics
5.	ecology2	999,999	4,995,991	yes	Biology
6.	thermal2	1,228,045	8,580,313	yes	Thermal Simulation
7.	G3_circuit	1,585,478	7,660,826	yes	Circuit Simulation
8.	FEM_3D_thermal2	147,900	3,489,300	no	Mechanics
9.	thermomech_dK	204,316	2,846,228	no	Mechanics
10.	ASIC_320ks	321,671	1,316,085	no	Circuit Simulation
11.	cage13	445,315	7,479,343	no	Biology
12.	atmosmodd	1,270,432	8,814,880	no	Atmospheric Model.

Table 2: Symmetric positive definite (s.p.d.) and nonsymmetric test matrices

In the following experiments we use the hardware system with NVIDIA C2050 (ECC on) GPU and Intel Core i7 CPU 950 @ 3.07GHz, using the 64-bit Linux operating system Ubuntu 10.04 LTS, CUSPARSE library 4.0 and MKL 10.2.3.029. The MKL_NUM_THREADS and MKL_DYNAMIC environment variables are left unset to allow MKL to use the optimal number of threads.

4.1 Sparse Triangular Solve (as a Standalone Algorithm)

Let us first analyse the performance of the standalone sparse triangular solve on the lower and upper triangular factors resulting from the incomplete-LU and Cholesky factorizations with 0 fill-in obtained using MKL `csrilu0` routine.

The absolute time in seconds (s) taken to solve the sparse triangular linear systems on the CPU using the MKL `csrsv` routine and on the GPU using the CUSPARSE library `csrsv_analysis` and `csrsv_solve` routines is given in Tab. 3. Notice that for the s.p.d. matrices we show the time taken to solve the incomplete-Cholesky upper (R) triangular factor, while for the nonsymmetric matrices we show the time taken to solve the incomplete-LU lower (L) and upper (U) triangular factors. The total time taken by the CUSPARSE library

sparse triangular solve is the sum of the time taken by the *analysis* and the *solve* phases performed by `csrsv_analysis` and `csrsv_solve` routines, respectively.

	CUSPARSE			MKL
#	`csrsv_analysis` time (s)	`csrsv_solve` time (s)	total time (s)	`csrsv` time (s)
1. (R)	0.06044	0.02567	0.08616	0.01287
2. (R)	0.09441	0.02635	0.12081	0.02754
3. (R)	0.03750	0.00207	0.03961	0.01166
4. (R)	0.04962	0.00738	0.05705	0.01533
5. (R)	0.06965	0.01441	0.08412	0.01931
6. (R)	0.08830	0.01427	0.10261	0.04060
7. (R)	0.10284	0.02042	0.12331	0.03342
8. (L)	0.05722	0.02417	0.08145	0.00702
8. (U)	0.05714	0.02423	0.08141	0.00624
9. (L)	0.02904	0.00458	0.03367	0.00829
9. (U)	0.02872	0.00403	0.03279	0.00907
10.(L)	0.02330	0.00288	0.02618	0.00703
10.(U)	0.02126	0.00145	0.02271	0.00727
11.(L)	0.04535	0.00415	0.04955	0.01775
11.(U)	0.04444	0.00445	0.04894	0.01609
12.(L)	0.08448	0.01099	0.09552	0.01636
12.(U)	0.08467	0.01109	0.09581	0.02664

Table 3: Time taken by MKL `csrsv` and CUSPARSE `csrsv_analysis` and `csrsv_solve`

In practice, we are often interested in solving the linear systems of the form $R^T R\mathbf{x} = \mathbf{f}$ and $LU\mathbf{x} = \mathbf{f}$. The time taken by the CUSPARSE library and MKL for this combination of the lower and upper triangular solves is summarized in Fig. 3.

Although MKL outperforms the CUSPARSE library if we consider a full single solve, there are many cases where the solution of the sparse triangular linear system requires the *solve* phase to be performed multiple times, while the analysis phase can be performed only once. In this case it is important to focus on the speedup obtained by the *solve* phase, because it can offset the initial slowdown resulting from the *analysis* phase. For this reason, we highlight in red the fastest time between CUSPARSE `csrsv_solve` and MKL `csrsv` in Tab. 3 and show the corresponding speedup in Fig. 4.

The estimated number of iterative steps needed to catch up with MKL, defined as (*analysis* time)/(MKL - *solve* time), is shown in Fig. 5. Although as

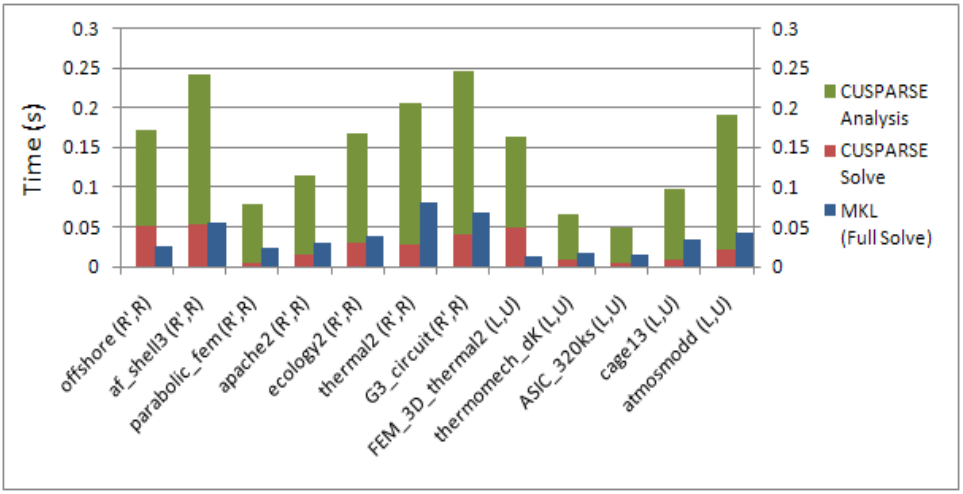


Figure 3: CUSPARSE (*analysis* and *solve*) and MKL (full solve) time

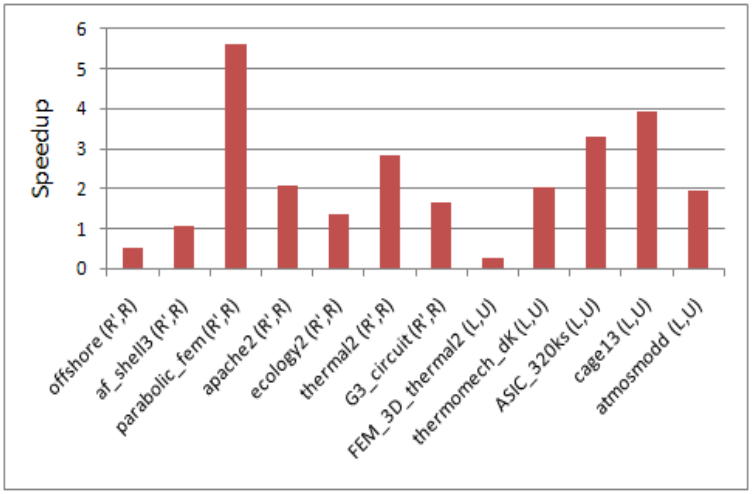


Figure 4: Speedup of CUSPARSE `csrsv_solve` versus MKL `csrsv`

indicated by "N/A" for offshore and FEM_3D_thermal2 matrices we are unable to match MKL, notice that for the majority of other matrices we are able to catchup with MKL in a few steps and will ultimately outperform it.

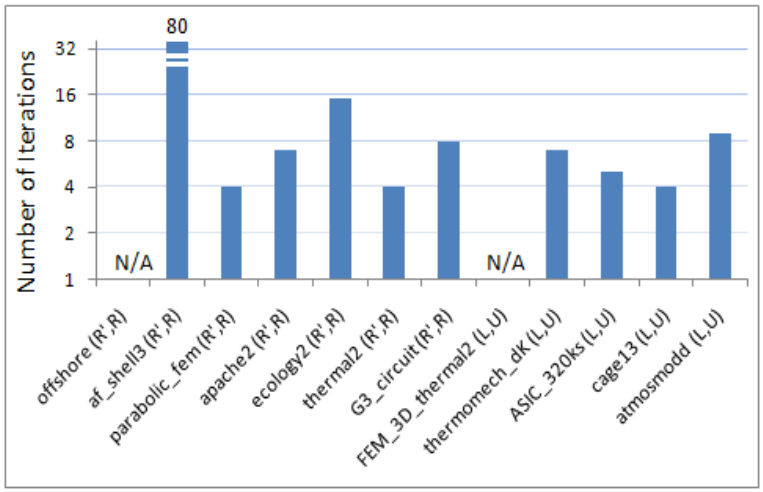


Figure 5: Estimated number of iterative steps needed to catchup with MKL

4.2 Sparse Triangular Solve (in the Iterative Method)

Let us now analyse the performance of the sparse triangular solve, in the context of solving the linear system

$$A\mathbf{x} = \mathbf{f} \tag{3}$$

using preconditioned Bi-Conjugate Gradient Stabilized (BiCGStab) and Conjugate Gradient (CG) iterative methods for nonsymmetric and s.p.d. systems, respectively. We precondition these methods using the incomplete-LU $A \approx LU$ and Cholesky $A \approx R^T R$ factorizations computed by the MKL routines `csrilu0` and `csrilut` with 0 and threshold fill-in, respectively.

In the `csrilut` routine we allow three different levels of fill-in denoted by $(5, 10^{-3})$, $(10, 10^{-5})$ and $(20, 10^{-7})$. In general, the (k, tol) fill-in is based on $nnz/n + k$ maximum allowed number of elements per row and the dropping of elements with magnitude $|l_{ij}|, |u_{ij}| < tol \times ||\mathbf{a}_i^T||_2$, where l_{ij}, u_{ij} and $\mathbf{a}_i^T$ are the elements of the lower L, upper U triangular factors and the i-th row of the coefficient matrix A, respectively.

We compare the implementation of the BiCGStab and CG iterative methods using the CUSPARSE and CUBLAS libraries on the GPU and MKL on the CPU. In our experiments we let the initial guess be zero, the right-hand-side $\mathbf{f} = A\mathbf{e}$ where $\mathbf{e}^T = (1, \ldots, 1)^T$, and the stopping criteria be the maximum number of iterations 2000 or relative residual $||\mathbf{r}_i||_2/||\mathbf{r}_0||_2 < 10^{-7}$, where $\mathbf{r}_i = \mathbf{f} - A\mathbf{x}_i$ is the residual at i-th iteration.

	ilu0		CPU			GPU			Speedup
#	fact. time(s)	copy time(s)	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	vs. ilu0
1	0.38	0.02	0.72	8.83E-08	25	1.52	8.83E-08	25	0.57
2	1.62	0.04	38.5	1.00E-07	569	33.9	9.69E-08	571	1.13
3	0.13	0.01	39.2	9.84E-08	1044	6.91	9.84E-08	1044	5.59
4	0.12	0.01	35.0	9.97E-08	713	12.8	9.97E-08	713	2.72
5	0.09	0.01	107.	9.98E-08	1746	55.3	9.98E-08	1746	1.92
6	0.40	0.02	155.	9.96E-08	1656	54.4	9.79E-08	1656	2.83
7	0.16	0.02	20.2	8.70E-08	183	8.61	8.22E-08	183	2.32
8	0.32	0.02	0.13	5.25E-08	4	0.52	5.25E-08	4	0.53
9	0.20	0.01	72.7	1.96E-04	2000	40.4	2.08E-04	2000	1.80
10	0.11	0.01	0.27	6.33E-08	6	0.12	6.33E-08	6	1.59
11	0.70	0.03	0.28	2.52E-08	2.5	0.15	2.52E-08	2.5	1.10
12	0.25	0.04	12.5	7.33E-08	76.5	4.30	9.69E-08	74.5	2.79

Table 4: `csrilu0` preconditioned CG and BiCGStab methods

	ilut$(5, 10^{-3})$		CPU			GPU			Speedup	
#	fact. time(s)	copy time(s)	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	vs. ilut$(5, 10^{-3})$	vs. ilu0
1	0.14	0.01	1.17	9.70E-08	32	1.82	9.70E-08	32	0.67	0.69
2	0.51	0.03	49.1	9.89E-08	748	33.6	9.89E-08	748	1.45	1.39
3	1.47	0.02	11.7	9.72E-08	216	6.93	9.72E-08	216	1.56	1.86
4	0.17	0.01	67.9	9.96E-08	1495	26.5	9.96E-08	1495	2.56	5.27
5	0.55	0.04	59.5	9.22E-08	653	71.6	9.22E-08	653	0.83	1.08
6	3.59	0.05	47.0	9.50E-08	401	90.1	9.64E-08	401	0.54	0.92
7	1.24	0.05	23.1	8.08E-08	153	24.8	8.08E-08	153	0.93	2.77
8	0.82	0.03	0.12	3.97E-09	2	1.12	3.97E-09	2	0.48	1.10
9	0.10	0.01	54.3	5.68E-03	2000	24.5	1.58E-01	2000	2.21	1.34
10	0.12	0.01	0.16	4.89E-11	4	0.08	6.45E-11	4	1.37	1.15
11	4.99	0.07	0.36	1.40E-08	2.5	0.37	1.40E-08	2.5	0.99	6.05
12	0.32	0.03	39.2	7.05E-08	278.5	10.6	8.82E-08	270.5	3.60	8.60

Table 5: `csrilut`$(5, 10^{-3})$ preconditioned CG and BiCGStab methods

The results of the numerical experiments are shown in Tab. 4 – 7, where we state the speedup obtained by the iterative method on the GPU over CPU (speedup), number of iterations required for convergence (# it.), achieved relative residual ($\frac{||\mathbf{r}_i||_2}{||\mathbf{r}_0||_2}$) and time in seconds taken by the factorization (fact.),

	ilut$(10, 10^{-5})$		CPU			GPU			Speedup	
#	fact. time(s)	copy time(s)	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	vs. ilut$(10, 10^{-5})$	vs. ilu0
1	0.15	0.01	1.06	8.79E-08	34	1.96	8.79E-08	34	0.57	0.63
2	0.52	0.03	60.0	9.86E-08	748	38.7	9.86E-08	748	1.54	1.70
3	3.89	0.03	9.02	9.79E-08	147	5.42	9.78E-08	147	1.38	1.83
4	1.09	0.03	34.5	9.83E-08	454	38.2	9.83E-08	454	0.91	2.76
5	3.25	0.06	26.3	9.71E-08	272	55.2	9.71E-08	272	0.51	0.53
6	11.0	0.07	44.7	9.42E-08	263	84.0	9.44E-08	263	0.59	1.02
7	5.95	0.09	8.84	8.53E-08	43	17.0	8.53E-08	43	0.64	1.68
8	2.94	0.04	0.09	2.10E-08	1.5	1.75	2.10E-08	1.5	0.64	3.54
9	0.11	0.01	53.2	4.24E-03	2000	24.4	4.92E-03	2000	2.18	1.31
10	0.12	0.01	0.16	4.89E-11	4	0.08	6.45E-11	4	1.36	1.18
11	28.9	0.09	0.44	6.10E-09	2.5	0.48	6.10E-09	2.5	1.00	33.2
12	0.36	0.03	36.6	7.05E-08	278.5	10.6	8.82E-08	270.5	3.35	8.04

Table 6: `csrilut`$(10, 10^{-5})$ preconditioned CG and BiCGStab methods

	ilut$(20, 10^{-7})$		CPU			GPU			Speedup	
#	fact. time(s)	copy time(s)	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	solve time(s)	$\frac{\|\|\mathbf{r}_i\|\|_2}{\|\|\mathbf{r}_0\|\|_2}$	# it.	vs. ilut$(20, 10^{-7})$	vs. ilu0
1	0.82	0.02	47.6	9.90E-08	1297	159.	9.86E-08	1292	0.30	25.2
2	9.21	0.11	32.1	8.69E-08	193	84.6	8.67E-08	193	0.44	1.16
3	10.4	0.04	6.26	9.64E-08	90	4.75	9.64E-08	90	1.10	2.36
4	8.12	0.10	15.7	9.02E-08	148	22.5	9.02E-08	148	0.78	1.84
5	8.60	0.10	21.2	9.52E-08	158	53.6	9.52E-08	158	0.48	0.54
6	35.2	0.11	29.2	9.88E-08	162	80.5	9.88E-08	162	0.56	1.18
7	23.1	0.14	3.79	7.50E-08	14	12.1	7.50E-08	14	0.76	3.06
8	5.23	0.05	0.14	1.19E-09	1.5	2.37	1.19E-09	1.5	0.70	6.28
9	0.12	0.01	55.1	3.91E-03	2000	24.4	2.27E-03	2000	2.25	1.36
10	0.14	0.01	0.14	9.25E-08	3.5	0.07	7.19E-08	3.5	1.28	1.18
11	218.	0.12	0.43	9.80E-08	2	0.66	9.80E-08	2	1.00	247.
12	15.0	0.21	12.2	3.45E-08	31	4.95	3.45E-08	31	1.35	5.93

Table 7: `csrilut`$(20, 10^{-7})$ preconditioned CG and BiCGStab methods

iterative solution of the linear system (solve), and `cudaMemcpy` of the lower and upper triangular factors to the GPU (copy). We include the time taken to compute the incomplete-LU and Cholesky factorization as well as to transfer the triangular factors from the CPU to the GPU memory in the computed speedup.

The comparison of the speedup obtained for BiCGStab and CG iterative methods preconditioned with different incomplete factorizations is shown in Fig. 6, where "*" indicates that the method did not converge to the required tolerance. Notice that for iterative methods preconditioned with incomplete-LU and Cholesky factorizations with 0 fill-in, the obtained speedup is usually similar to the one obtained for the *solve* phase in Fig. 4. The difference is due to the presence of other linear algebra operations, such as matrix-vector multiplication, in the iterative method and also the time taken by the *analysis* phase and factorization, which is included in these results.

Also, notice that the best speedup is obtained for the incomplete-LU and Cholesky factorization with 0 fill-in. In general, the speedup for the incomplete factorizations decreases as the threshold parameters are relaxed and the factorization becomes more dense, thus inhibiting parallelism due to data dependencies between rows.

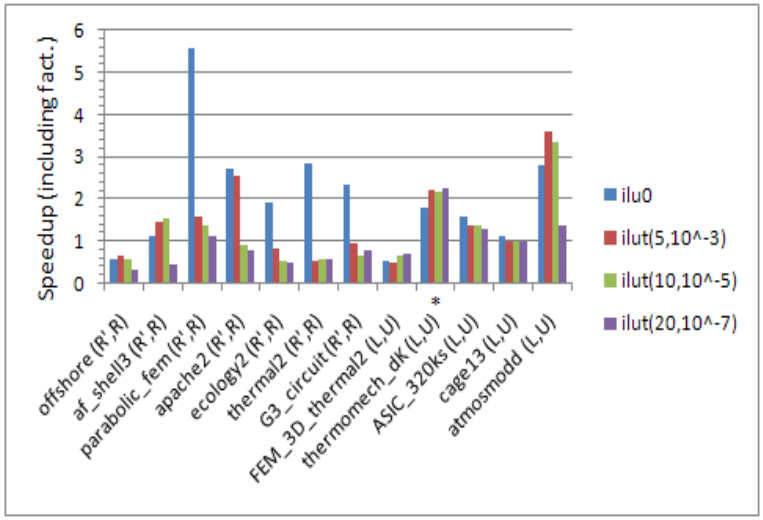


Figure 6: Speedup of BiCGStab and CG with incomplete-LU/Cholesky preconditioning

Although the incomplete factorizations with a more relaxed threshold are often closer to the exact factorization and thus result in fewer iterative steps, they are also much more expensive to compute. Moreover, notice that even though the number of iterative steps decreases, each step is more computationally expensive. As a result of these tradeoffs the total time, the sum of the time taken by the factorization and the iterative solve, for the iterative method shown on Fig. 7 does not necessarily decrease with a more relaxed threshold in our numerical experiments.

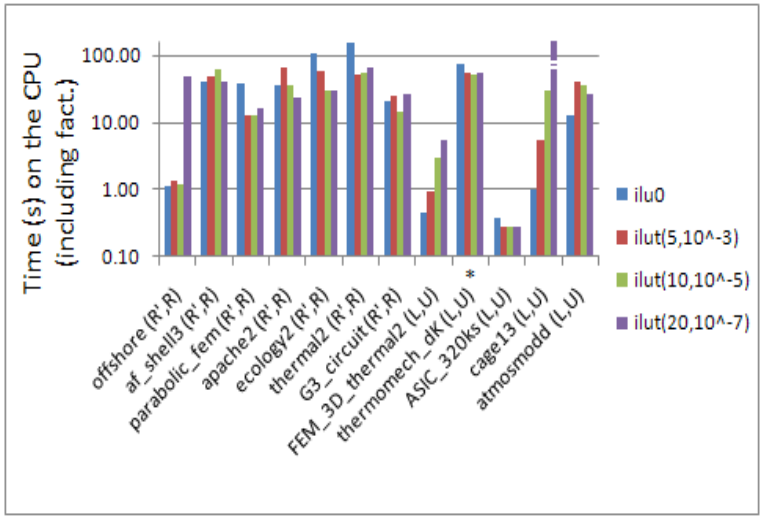


Figure 7: Total time taken by the preconditioned iterative method on the CPU

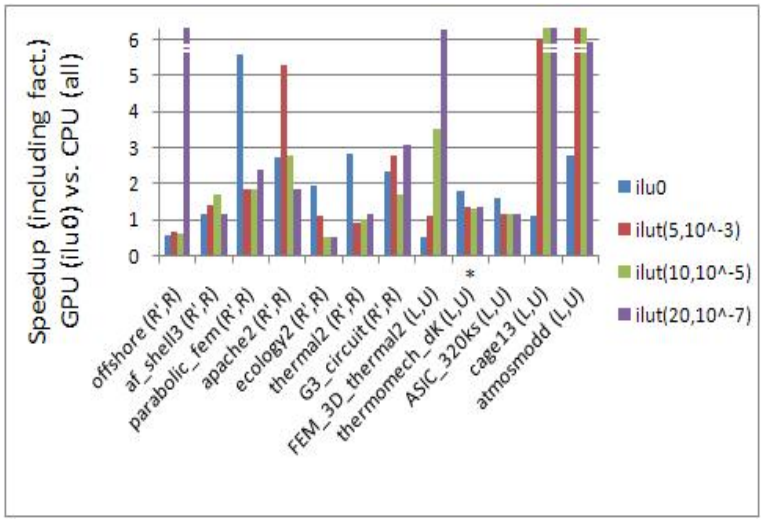


Figure 8: Speedup of prec. BiCGStab & CG on GPU (with `csrilu0`) vs. CPU (with all)

The speedup based on the total time taken by the preconditioned iterative method on the GPU with `csrilu0` preconditioner and CPU with all four preconditioners is shown in Fig. 8. Notice that for majority of matrices in our numerical experiments the implementation of the iterative method using the CUSPARSE and CUBLAS libraries does indeed outperform the MKL.

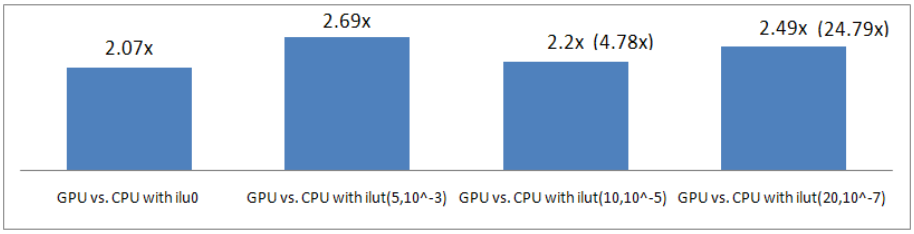


Figure 9: Average speedup of BiCGStab and CG on GPU (with `csrilu0`) and CPU (with all)

Finally, the average of the obtained speedups is shown in Fig. 9, where we have excluded the runs with cage13 matrix for $\texttt{ilut}(10, 10^{-5})$ and runs with offshore and cage13 matrices for $\texttt{ilut}(20, 10^{-7})$ incomplete factorizations because of their disproportional speedup. However, the speedup including these runs is shown in parenthesis on the same plot. Consequently, we can conclude that the incomplete-LU and Cholesky preconditioned BiCGStab and CG iterative methods obtain on average more than 2× speedup on the GPU over their CPU implementation.

5 Conclusion

A novel parallel algorithm for solution of sparse triangular linear systems was developed. It splits the solution of a triangular linear system in two phases. The slower *analysis* phase needs to be performed only once, while the faster *solve* phase can be performed multiple times. The performance of the sparse triangular solve depends highly on the sparsity pattern of the triangular matrix at hand. Although, there are sparsity patterns for which the computation is inherently sequential (consider for example a bidiagonal matrix), there are many other realistic sparsity patterns where enough parallelism is available.

The new algorithm is ideally suited for the splitting based iterative schemes and incomplete-LU and Cholesky preconditioned iterative methods. In this setting the CUDA implementation of the algorithm on the GPU can outperform the MKL implementation of the sparse triangular solve on the CPU. Also, in our numerical experiments the incomplete-LU and Cholesky preconditioned iterative methods implemented on the GPU using the CUSPARSE and CUBLAS libraries achieved an average of 2× speedup over their MKL implementation.

To conclude, it is worth mentioning that the use of multiple-right-hand-sides would increase the available parallelism and can result in a significant relative performance improvement in the sparse triangular solve. Also, the development of incomplete-LU and Cholesky factorizations using CUDA parallel programming paradigm can further improve the speedup obtained by the preconditioned iterative methods on the GPU.

6 Acknowledgements

The author would like to acknowledge Ujval Kapasi, Philippe Vandermersch, Alex Fit-Florea, Marcin Andrychowicz, Lung Sheng Chien, Nathan Whitehead, Elif Albuz, Nathan Bell, Jonathan Cohen and Michael Garland for their comments and suggestions.

References

[1] F. L. Alvarado and R. Schreiber, Optimal Parallel Solution of Sparse Triangular Systems, SIAM J. Sci. Comput., pp. 446-460 (14), 1993.

[2] E. Anderson and Y. Saad Solving Sparse Triangular Linear Systems on Parallel Computers, Int. J. High Speed Comput., pp. 73-95, 1989.

[3] N. Bell and M. Garland, Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors, Proc. Conf. HPC Networking, Storage and Analysis (SC09), ACM, pp. 1-11, 2009.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, Introduction to Algorithms, The MIT Press, Cambridge, MA, 2nd Ed., 2001.

[5] S. C. Eisenstat, M. T. Heath, C. S. Henkel and C. H. Romine, Modified Cyclic Algorithms for Solving Triangular Systems on Distributed-Memory Multiprocessors, SIAM J. Sci. Statist. Comput., pp. 589-600 (9), 1988.

[6] T. A. Davis, Direct Methods for Sparse Linear Systems, SIAM, Philadelphia, PA, 2006.

[7] R. K. Ghosh and G. P. Bhattacharjee, A Parallel Search Algorithm for Directed Acyclic Graphs, BIT Numerical Mathematics, pp. 134-150 (24), 1984.

[8] A. Greenbaum, Solving Sparse Triangular Linear Systems using Fortran with Parallel Extensions on the NYU Ultracomputer Prototype, Report 99, NYU Ultracomputer Note, New York University, NY, April, 1986.

[9] M. T. Heath and C. H. Romine, Parallel Solution of Triangular Systems on Distributed-Memory Multiprocessors, SIAM J. Sci. Statist. Comput., pp. 558-588 (9), 1988.

[10] N. J. Higham and A. Pothen, Stability of the Partitioned Inverse Method for Parallel Solution of Sparse Triangular Linear Systems, SIAM J. Sci. Comput., pp. 139-148 (15), 1994.

[11] D. B. Kirk and W. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach, Elsevier, 2010.

[12] G. Li and T. F. Coleman, A New Method for Solving Triangular Systems on Distributed-Memory Message-Passing Multiprocessors, SIAM J. Sci. Statist. Comput., pp. 382-396 (10), 1989.

[13] J. Mayer, Parallel Algorithms for Solving Linear Systems with Sparse Triangular Matrices, Computing, pp. 291-312 (86), 2009.

[14] R. Mirchandaney, J. H. Saltz and D. Baxter, Run-Time Parallelization and Scheduling of Loops, IEEE Transactions on Computers, pp. (40), 1991.

[15] J. Nickolls, I. Buck, M. Garland and K. Skadron, Scalable Parallel Programming with CUDA, Queue, pp. 40-53 (6-2), 2008.

[16] A. Pothen and F. L. Alvarado, A Fast Reordering Algorithm for Parallel Sparse Triangular Solution, SIAM J. Sci. Statist. Comput., pp. 645-653 (13), 1992.

[17] E. Rothberg and A. Gupta, Parallel ICCG on a Hierarchical Memory Multiprocessor Addressing the Triangular Solve Bottleneck, Parallel Comput., pp. 719-741 (18), 1992.

[18] Y. Saad, Iterative Methods for Sparse Linear Systems, SIAM, Philadelphia, PA, 2nd Ed., 2003.

[19] J. H. Saltz, Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors, SIAM J. Sci. Statist. Comput., pp. 123-144 (11), 1990.

[20] A. H. Sameh and R. P. Brent, Solving Triangular Systems on a Parallel Computer, SIAM J. Numer. Anal., pp. 1101-1113 (14), 1977.

[21] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley, 2010.

[22] C. F. van Loan and G. H. Golub, Matrix Computations, The John Hopkins University Press, 3rd Ed., 1996.

[23] M. Wolf, M. Heroux and E. Boman, Factors Impacting Performance of Multithreaded Sparse Triangular Solve, 9th Int. Meet. HPC Comput. Sci. (VECPAR), 2010.

[24] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms, Parallel Comput., pp. 178-194 (35-3), 2009.

[25] NVIDIA CUSPARSE and CUBLAS Libraries,
http://www.nvidia.com/object/cuda_develop.html

[26] Intel Math Kernel Library,
http://software.intel.com/en-us/articles/intel-mkl

[27] The University of Florida Sparse Matrix Collection,
http://www.cise.ufl.edu/research/sparse/matrices/