Multi-Elimination ILU Preconditioners on GPUs

Dimitar Lukarski
Department of Information Technology,
Uppsala University, Sweden.
Email: dimitar.lukarski@it.uu.se

Hartwig Anzt, Stanimire Tomov, Jack Dongarra
Innovative Computing Lab,
University of Tennessee, Knoxville, USA.
hanzt@icl.utk.edu, tomov@cs.utk.edu, dongarra@eecs.utk.edu

***Abstract*—Iterative solvers for sparse linear systems often benefit from using preconditioners. While there are implementations for many iterative methods that leverage the computing power of accelerators, porting the latest developments in preconditioners to accelerators has been challenging. In this paper we develop a self-adaptive multi-elimination preconditioner for graphics processing units (GPUs). The preconditioner is based on a multi-level incomplete LU factorization and uses a direct dense solver for the bottom-level system. For test matrices from the University of Florida matrix collection, we investigate the influence of handling the triangular solvers in the distinct iteration steps in either single or double precision arithmetic. Integrated into a Conjugate Gradient method, we show that our multi-elimination algorithm is highly competitive against popular preconditioners, including multi-colored symmetric Gauss-Seidel relaxation preconditioners, and (multi-colored symmetric) ILU for numerous problems.**

I. Introduction

Developing efficient solvers for large linear systems is one of the main challenges in scientific computing. The reason is that in many scientific algorithms simulating chemical, physical, or biological processes, the solution of discretized partial differential equations poses the main work from the computational point of view. The partial differential equations modeling various physical phenomena are often discretized by finite element or finite difference methods, which usually results in large, sparse systems of equations. To solve them, iterative methods are often preferred to direct solvers, as iterative solvers are often able to provide a solution approximation of sufficient accuracy for a significantly lower computational cost. However, to achieve this, iterative methods often rely on preconditioners. A popular class is based on the incomplete LU factorization (ILU) [20]. While using ILU without fill-ins can lead to appealing convergence improvement to the top-level iterative method, it may also fail due to its rather rough approximation properties, e.g., when solving linear systems arising from complex applications like computational fluid dynamics [19]. To enhance the accuracy of the preconditioner, one can allow for additional fill-in in the preconditioning matrix (ILU(m), see [20]). Additional fill-in usually reduces the amount of parallelism in ILU(m) compared to ILU(0), but there are a number of techniques designed to retain it, such as the level-scheduling techniques [20], [17] or the multi-coloring algorithms for the ILU factorization with levels based on the power(q)-pattern method [16]. Another approach is the idea of multi-elimination [19], [21], which is based on successive independent set coloring [12]. The motivation is that in a step of the Gaussian elimination, there usually exists a large set of rows that can be processed in parallel. This set is called the independent set. For multi-elimination, the idea is to determine this set, and then eliminate the unknowns in the respective rows simultaneously, to obtain a smaller reduced system. To control the sparsity of the factors, multi-elimination uses an approximate reduction based on a standard threshold strategy. Recursively applying this step, one obtains a sequence of linear systems with decreasing dimension and increasing fill-in. On the lowest level, the system must be solved, e.g., either by an iterative method, or by a direct solver based on an LU factorization. Especially for long linear system cascades and a permissive threshold, the small but dense lowest-level problem may preferably be solved using a direct method. While there exists some work on multi-elimination preconditioners using iterative solve for the bottom-level system [19], [21], little attention has been put on solving the lowest level using a direct method, and, to the best of our knowledge, all existing implementations are for CPU-based systems. To take into account the hardware development trends, and in particular to leverage the GPU's high-performance potential in scientific computing applications [13], [3], motivated us to investigate the potential of multi-elimination preconditioned Krylov subspace solvers on GPU-accelerated systems. In this first attempt to port multi-elimination preconditioners to accelerators, we focus particularly on using long reduction sequences. We also argue that for GPUs a combination of a sparse and direct factorization during the preconditioner setup phase, followed by corresponding forward-backward triangular solves during the iterations, may provide performance advantages for various problems vs. solving iteratively at the bottom level as initially proposed in [19]. Furthermore, we investigate the influence of employing the forward and backward triangular solves in a lower precision arithmetic. The motivation stems from trading-off solver-accuracy in an iterative process for acceleration through a more efficient hardware use.

The rest of the paper is structured as follows. First, we provide more details about the multi-elimination methods and motivate the use of mixed precision in scientific computing. Then, we describe the software we employed and details about the hardware and the linear systems we target in our experiments. In the experimental section we first address the preprocessing phase of setting up the multi-elimination preconditioner framework using a self-adaptive level depth. We then analyze convergence and top-level solver performance with respect to a threshold controlling the amount of fill-in. Motivated by the more efficient hardware use, we investigate the impact of handling the bottom-level system in lower precision than working precision arithmetic and compare the hybrid approach to iterative bottom-level solvers. Finally, we compare against some popular preconditioners for symmetric and positive definite problems.

II. Multi-Elimination Preconditioners

While we want to recall the central ideas of the multi-elimination concept, a detailed derivation can be found in [19]. The underlying concept is to use permutations P to bring the original matrix A of the system $Ax = b$ that we want to solve into the form

$$PAP^T \equiv \begin{pmatrix} D & F \\ E & C \end{pmatrix},$$

where D is preferably a diagonal or at least an easy to invert matrix, so that

$$PAP^T \equiv \begin{pmatrix} D & F \\ E & C \end{pmatrix} = \begin{pmatrix} I & 0 \\ ED^{-1} & I \end{pmatrix} \times \begin{pmatrix} D & F \\ 0 & \hat{A} \end{pmatrix}$$

where

$$\hat{A} = C - ED^{-1}F$$

are easy to compute. One way to achieve this is by using an independent set ordering [12], [14], [23], [18], where non-adjacent unknowns of the original matrix A are determined. In the adjacency graph of A, two vertices i and j corresponding to unknowns are adjacent, if there exists an edge connecting them (i.e., $a_{i,j} \neq 0$ or $a_{j,i} \neq 0$). A greedy algorithm [11] for independent set ordering is given in Algorithm 1.

Algorithm 1 Greedy algorithm in pseudocode for independent set ordering [19].

```
1: $S = \emptyset$
2: let $n_0, n_1 \dots n_n$ be an ordering of the nodes
3: for($i = 0; i < n; i++$){
4:     if(node $n_i$ not marked){
5:         $S = S \bigcup n_i$
6:         mark $n_i$ and all its neighbors
7:     }
8: }
```

Denoting $A \equiv A_{nlev} and \hat{A} = A_{nlev-1}$, it is possible to define a recursive factorization from level `nlev` down to level 1. In particular, for any level j one must determine the independent set for that level, and to decompose A_j using the permutation P_j into the desired form

$$P_j A_j P_j{}^T = \begin{pmatrix} I & 0 \\ E_j D_j^{-1} & I \end{pmatrix} \times \begin{pmatrix} D_j & F_j \\ 0 & A_{j-1} \end{pmatrix} \tag{1}$$

with

$$A_{j-1} = C_j - E_j D_j^{-1} F_j. \tag{2}$$

On the lowest level, the linear system associated with the matrix A_1 must be solved. This can be done approximately by an iterative method, including by a single application of a preconditioner to the corresponding right-hand side, or via a direct solver. The significantly smaller dimension of A_1 however comes at the price of fill in. To control the fill-in, one often applies dropping strategies when forming the reduced systems. A simple dropping strategy, for example, arises by neglecting any introduced fill-in if smaller than a prescribed tolerance τ. Namely, one replaces (2) by

$$A_{j-1} = C_j - E_j D_j^{-1} F_j + R_j, \tag{3}$$

where R contains the fill-in that is dropped in this reduction step. Instead of storing the complete sequence A_j to the lowest level, it is more efficient to generate only the bottom-level problem A_1, and to store the transformation between the levels:

$$B_{j-1} = \begin{pmatrix} D_j & F_j \\ E_j D_j^{-1} & 0 \end{pmatrix}. \tag{4}$$

While the accuracy of the multi-elimination preconditioner depends on the threshold, the general concept of the preprocessing phase is given in Algorithm 2. In the iteration phase (see Algorithm 3) the sequence of transformations and the low-level solver are applied to approximate the solution of the original problem. This is achieved by applying the series of permutations $P_{nlev}, \dots, P_1$ to the right-hand-side (RHS), and overwriting the result on the highest level with the current solution vector x_0. For an efficient implementation we apply, on the respective level j, the decomposition [19]

$$x_j := \begin{pmatrix} y_j \\ x_{j-1} \end{pmatrix} \tag{5}$$

and compute according to the partitioning in (1) the forward sweep as [19]:

$$x_{j-1} := x_{j-1} - E_j D_j^{-1} y_j. \tag{6}$$

Consequently, backward solution for y_j hence becomes [19]:

$$y_j := D_j^{-1}\left(y_j - F_j x_{j-1}\right). \tag{7}$$

Algorithm 2 Preprocessing phase of the multi-elimination build phase $MCILU(A, level)$ in pseudocode [15].

```
1: find independent set ordering $P_{level}$ for $A_{level}$
2: apply $P_{level}$ to $A_{level}$ to obtain the desired form (1)
3: extract $D_{level}, F_{level}, E_{level}, C_{level}$
4: compute $A_{level-1}$ and $B_{level-1}$ according to (3), (4)
5: drop-off elements $R_{level-1}$ in $A_{level-1}$
6: if ($level > 1$) : $MCILU(A_{level-1}, level - 1)$
7: if ($level = 1$) : LU factorization of $A_{level-1}$
```

The major difference between our implementation and [19] is, that we base our implementation on a fully recursive organization of the building and the solving phase [15].

Algorithm 3 Iteration phase of the multi-elimination preconditioner $MCILU-Solve(A, level, x, b)$ in pseudocode [15].

```
1: apply permutation to $b$ and copy into $x$
2: split $x$ into $y$ and $\hat{x}$ according to (5)
3: $\hat{x} = \hat{x} - E_{level} D_{level}^{-1} y$ (forward sweep)
4: if $(level > 1)$ : $MCILU-Solve(A_{level}, level-1, \hat{x}, \hat{x})$
5: if $(level = 1)$ : Direct solve $A_{level}\hat{x} = \hat{x}$
6: $y = D_{level}^{-1}(y - F_{level}\hat{x})$ (backwards solution)
7: apply permutation to obtain solution $x$
```

As the sparsity and the dimension of the reduced systems decrease, especially when using long cascades of factorizations, direct methods may become attractive for the solution process on the lowest level. Indeed, the fact that the exact LU factorization has to be computed only once in the preprocessing phase, and reused for the preconditioning steps in forward and backward triangular matrix solvers, makes this approach appealing. On the other hand, the system on the lowest level does not have to be solved to full double precision accuracy for many problems. Indeed, in most implementations the bottom-level solves are handled by iterative methods with a prescribed residual stopping criterion. This justifies an approach that implements the LU factorization in the preconditioner setup in double precision, but performs the forward and backward triangular solves at each iteration in single precision arithmetic (possibly enhanced by mixed-precision iterative refinement, including falling back to high precision arithmetic if needed, to solve with prescribed residual stopping criterion). We will motivate in the next section that using arithmetic of lower than the working precision may accelerate the iteration process, and potentially improve the time-to-solution performance. For this reason, we will add a survey on the potential of using a low-precision bottom-level solver to the experimental section.

III. Mixed Precision Methods

Most scientific simulation codes are implemented in IEEE 754 double precision arithmetic. This usually allows for efficient hardware usage while providing simulation results of sufficient accuracy. However, especially with the integration of coprocessor technology into the computation process, it is not always obvious whether using double precision throughout the complete algorithm is necessary and beneficial. Especially when solving linear systems of equations iteratively, using a less complex precision format for parts of the solution process may improve performance without sacrificing the accuracy of the final solution approximation. One popular example is the usage of single instead of double precision when solving the error correction equation in an iterative refinement process [9], [6], [5]. This technique works very well in accelerating dense linear algebra solvers on GPUs as well, e.g., see the 3 to 4× performance improvements reported on a GTX 280 GPU [22]. The motivation often stems from hardware that offers higher computing performance in single precision. In the past, GPUs often rendered significant speedup factors when switching from double to single precision, especially if double precision was not inherently supported but emulated [10], [4]. While in recent years, similarly to CPUs, these differences have decreased to two

(e.g., the NVIDIA Fermi GPUs), particularly for the high-end products integrated in large supercomputers, the difference in the newest NVIDIA GPUs, e.g., the K20 and K40, is increased to three.

Use of lower precision arithmetic, in addition to higher compute power, also reduces storage, and hence data communication times. The memory bandwidth, often the bottleneck in numerical simulation codes, can handle twice as many floating point numbers when using single instead of double precision. As this fact applies to all levels of nowadays often very sophisticated memory hierarchies, a baseline speedup factor of two can be expected when switching from double to single precision for memory-bound applications. As discussed above, the speedups can further increase for compute-bound applications [9], [22].

The central question when using lower precision arithmetic is whether the demanded accuracy for the solution approximation can be achieved. While one of the most commonly used workarounds is offered by the aforementioned mixed precision iterative refinement, we aim for using single precision only in the explicit solution process on the lowest level of the multi-elimination solver. The motivation is that the lowest-level system in a multi-elimination preconditioner could also be handled by an iterative method for many problems, and high accuracy in those cases may not be required for the convergence of the top-level iterative solver.

IV. Experiment Setup

A. Implementation Details

We implement the multi-elimination preconditioner using mixed precision for the solution of the lowest-level problem by using the combination of two open source software libraries: PARALUTION [15] and MAGMA [2]. PARALUTION is a high-level software library for sparse linear algebra on multicore any manycore systems developed at the University of Uppsala. In the current version it supports CUDA, and an OpenCL backend which allows for outsourcing computational kernels to accelerators like graphics processing units or the Xeon Phi (MIC). It provides, in addition to efficient implementations of the most commonly used iterative solvers, a large variety of preconditioners, including multi-elimination with flexible dropping strategies. The top-level solver we consider in this paper is a Conjugate Gradient method (CG, see [20]), which is among the most efficient Krylov subspace solvers for symmetric, positive definite systems [20]. Instead of using an iterative solver on the lowest level, we employ an interface that enables the usage of sparse and dense linear algebra operations implemented in MAGMA. While the sparse matrices on all levels are handled in CSR format [7], we convert the system on the lowest level into dense format and apply the LU factorization provided by the MAGMA library in double precision. During all operations in the preconditioner setup phase, the data resides in the GPU memory. In the iteration process, the forward and backward triangular solve of the linear system is executed in either double or single precision. After the solution of the lowest-level problem, the vectors are converted back to double precision and the multi-elimination preconditioner framework of PARALUTION is resumed.

B. Hardware Platform

We performed the experiments on a Tesla K40c GPU that belongs to the Atlas line of NVIDIA's hardware accelerators. The GPU consists of 2880 CUDA cores, and runs at 876 MHz [1]. The theoretical peaks are $1,682$ GFlop/s for double precision arithmetic and $5,046$ GFlop/s for single precision. It is equipped with 12 GB of GDDR5 memory accessed with a peak bandwidth of 288 GB/s. The host processor is an Intel Xeon E5 (codename: Sandy Bridge, model `0x2D`, family `0x06`) in a two-socket configuration featuring 8 cores in each socket with HyperThreading enabled and the nominal frequency of 2.6 GHz.

C. Solver Parameters

All experiments solve the linear system $Ax = b$ where we set the initial right-hand-side to $b \equiv 1$, start with the initial guess $x \equiv 0$ and run the iteration process until we achieve a relative residual accuracy of $1e - 6$. The algorithm-specific GPU-kernels were based on CUDA in version 5.5, while only the standard matrix and vector operations are handled by NVIDIA's CUBLAS and CUSPARSE libraries. In the preprocessing phase of the multi-elimination, the identification of an independent set via a graph algorithm is handled by the CPU of the host system, the factorization process itself, including the permutation and the generation of the lower-level systems via a sparse matrix-matrix multiplication is implemented on the GPU.

D. Test Matrices

For the experiments, we use a set of symmetric, positive definite (SPD) test matrices taken either from the University of Florida matrix collection (UFMC)[1], Matrix Market[2], or generated as finite difference discretization (LAPLACE_2D_1M). The test matrices are listed along with some key characteristics in Table I.

V. Experimental Results

In this section, we first analyze the preprocessing phase of the multi-elimination preconditioner. We then investigate the influence of the drop off τ on the convergence rate, and address the issue of whether double precision is necessary throughout the complete algorithm. We show that handling the forward and backward triangular solve of the bottom-level problem A_1 in single precision may improve the time-to-solution performance by compensating for additional iterations by a faster execution of the triangular solve. Finally, we compare, for suitable parameter choices, time-to-solution performance against some popular preconditioners for symmetric and positive definite problems.

[1]UFMC; see http://www.cise.ufl.edu/research/sparse/matrices/

[2]see http://math.nist.gov/MatrixMarket/

Matrix	#nonzeros (nnz)	Size (n)	nnz/n
APACHE_2	4,817,870	715,176	6.74
ECOLOGY_2	4,995,991	999,999	5.00
G2_CIRCUIT	726,674	150,102	4.83
G3_CIRCUIT	7,660,826	1,585,478	4.83
GR_30_30	7744	900	8.60
LAPLACE_2D_1M	4,996,000	1,000,000	4.99
NOS4	594	100	5.94
OFFSHORE	4,242,673	259,789	16.33
SERENA	64,131,971	1,391,349	46.09
STOCF-1465	21,005,389	1,465,137	14.34
THERMAL2	8,580,313	1,228,045	6.99

TABLE I: Description and properties of the test matrices.

A. Preconditioner Setup

Using sophisticated preconditioners like the multi-elimination often requires a comparably expensive preprocessing phase for setting up the preconditioner [16]. While this may make them unattractive for problems where a linear system is solved only once, many scientific simulations address dynamic problems where the solution of the linear problem with a changing right-hand-side is required, e.g. fluid flow problems [8]. For these non-stationary problems, conceding an expensive preprocessing phase usually pays off after a few time steps. During the setup phase for the multi-elimination preconditioner, first the transformation matrix P_{level} must be generated, which requires finding an independent set, see Algorithm 2. Next, the transformation is applied to A_{level} to obtain the desired form (1), and after extracting the respective matrices, $A_{level-1}$ is formed according to (3). Recursively applying these steps, we can obtain a sequence of linear problems down to the bottom-level problem that is solved by a direct dense method. In Figure 1 we visualize the sparsity pattern of the matrix sequences generated by running the preconditioner setup phase and applying the permutations on the distinct levels on the matrix GR_30_30 and NOS4, respectively.

While the sparsity structure is well-preserved in the sequence of reduced systems, the size reductions (45% and 61%, respectively) come at the cost of an increasing number of fill-ins, see Table II. As already mentioned before, an efficient workaround for this is given by applying some dropping strategy to control the fill-in in the reduction process. Using suitable parameters τ, it is possible to maintain the number of nonzero entries in the original system throughout the sequence of factorizations.

The preprocessing phase also includes the factorization of the bottom-level problem. For this purpose, we convert it to dense format and then apply the routines provided by MAGMA. While the level-depth can be optimized to specifics of the linear problem at hand and the available hardware, we

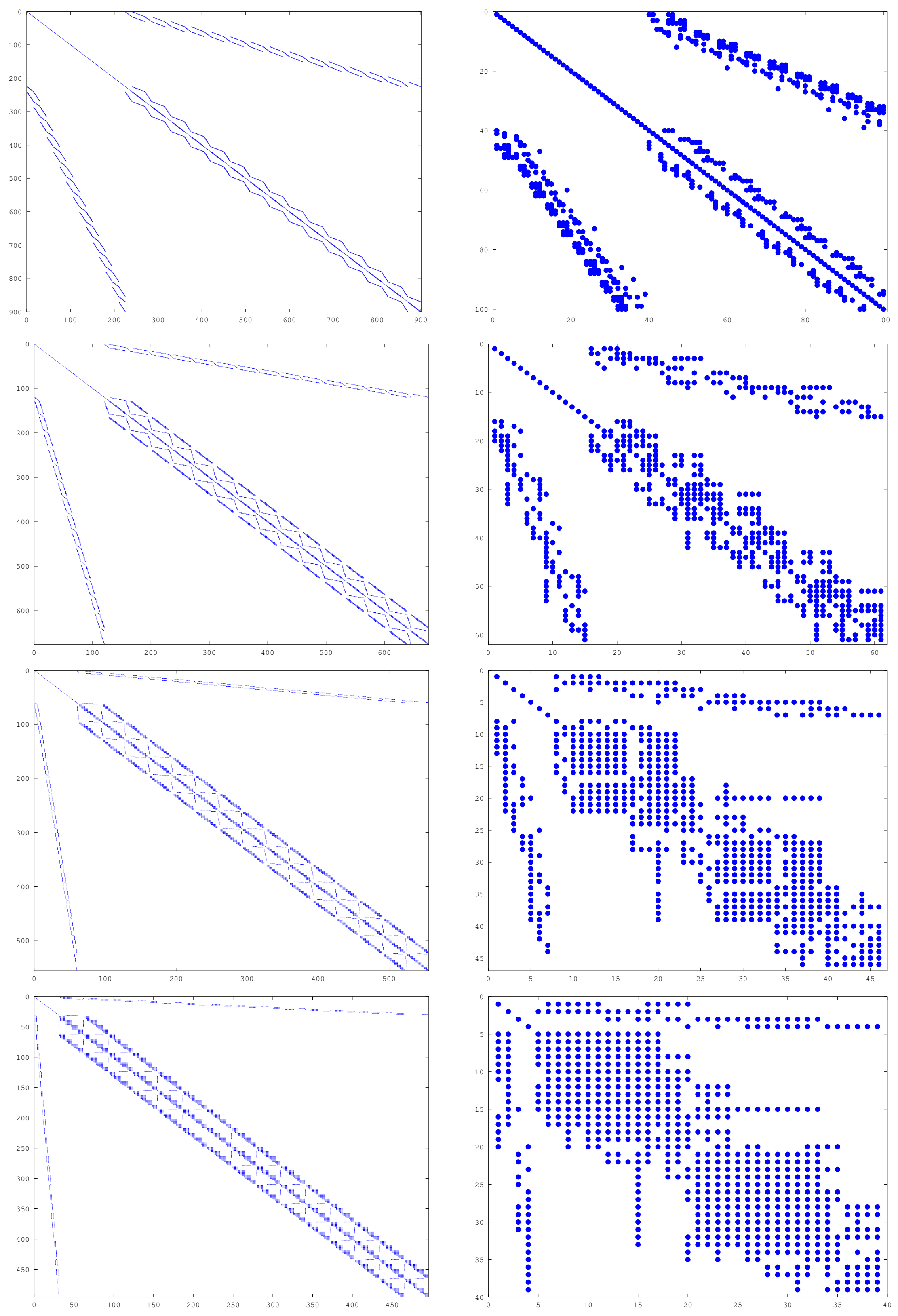

Fig. 1: Sparsity pattern of the recursive factorization applied to GR_30_30 (left) and NOS4 (right), respectively.

base our implementation on a self-adaptive strategy continuing the recursive multi-elimination factorization process until a system size smaller than $n < 12,000$ is reached. Note that allowing for complete fill-in when forming the linear system to be factorized improves the accuracy, and we therefore do not apply any dropping technique for the bottom-level.

	GR_30_30		NOS4	
level	n	nnz	n	nnz
4	900	7744	100	594
3	675	9809	61	599
2	555	10989	46	676
1	495	14585	39	709

TABLE II: Properties of the sequence of matrices obtained by the recursive factorization.

B. Choosing the Threshold τ

As mentioned before, the sparsity pattern of the sequence of factorizations can be efficiently controlled by applying a dropping criteria technique. Choosing the dropping parameter τ has significant influence on the method's convergence and performance characteristics: larger values for τ allow for additional fill-in, resulting in a larger reduced system, a longer sequence of factorizations to the bottom-level system, and a computationally more expensive reduction and prolongation operation between the levels. The computational cost of every iteration is the combination of the level transfers and the bottom-level triangular solves. Hence, dropping more elements may accelerate the execution of every iteration (assuming the same size of the bottom-level problem), but due to the reduced accuracy of the preconditioner, the top-level iteration method may require more iterations to reach the demanded accuracy. However, choosing the threshold τ and the level depth is more involved than just trading off iteration count and the cost of the preconditioner. An inappropriate choice may result in the sequence of linear systems losing regularity, and a singular bottom-level system will cause the breakdown of the preconditioner. A second constraint is given by the limited GPU memory size, that must be large enough to store the level transformations as well as the LU factorization of the bottom-level problem.

While different threshold strategies can be implemented, we choose to relate the threshold to the average absolute value of the nonzero element:

$$\tau = \beta \cdot \frac{\sum_{a_{ij}\neq 0} |a_{ij}|}{nnz}, \tag{8}$$

where β can be used to increase or decrease τ. In Tables III and IV, we investigate the influence of β on the factorization sequence and the top-level solver for the matrices OFFSHORE and STOCF-1465, respectively. As expected, choosing larger β, which results in dropping more elements in the factorization sequence, we decrease the number of levels necessary to obtain a system of size $n < 12,000$ (see column labelled l). This, however, comes at the price of a higher iteration count. In the end, the trade-off determines the time-to-solution performance, and from the results visualized in Figure 2 we deduce, that this is problem-dependent. For the test case OFFSHORE we observe the overall trend that dropping more elements improves the

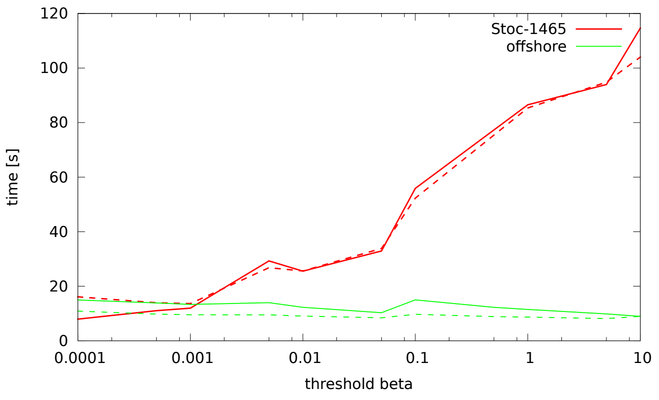

Fig. 2: Time-to-solution performance of the multi-elimination preconditioned Conjugate Gradient using double or single precision triangular solve (solid and dashed lines, respectively).

		A_1		*double ts*		*single ts*	
β	l	n	nnz	# iters	time [s]	# iters	time [s]
$1e-4$	10	9013	1599025	1220	14.95	1223	10.87
$5e-4$	6	8812	967606	1251	13.89	1250	9.80
$1e-3$	5	8703	792275	1242	13.32	1268	9.56
$5e-3$	3	9164	483780	1292	13.97	1295	9.52
$1e-2$	3	8306	421328	1309	12.27	1309	9.08
$5e-2$	3	7017	284637	1347	10.31	1351	8.42
$1e-1$	2	9493	384617	1324	15.01	1314	9.70
$5e-1$	2	8131	306573	1356	12.28	1349	8.88
$1e+0$	2	7640	266560	1375	11.49	1364	8.68
$5e+0$	2	6605	159175	1416	9.85	1404	8.16
$1e+1$	2	6106	114758	1412	8.97	1414	8.96

TABLE III: Influence of β on the depth of the factorization sequence (level depth l), bottom-level linear system (A_1) characteristics and top-level solver performance using either double or single precision triangular solve (labelled *double ts* and *single ts*, respectively). The top-level iteration method is applied to case OFFSHORE with a relative residual stopping criterion of $1e-6$.

time-to-solution performance. For the STOCF-1465 matrix however, it is the other way around: superior performance can be achieved when allowing for more fill-in.

We conclude that without detailed knowledge about the matrix characteristics, it is hard to optimize the drop-off threshold. For this reason, we base our implementation on the default value of $\beta = 0.1$ as using this value we observed convergence for all test cases. However, we keep in mind from the data in Tables III and IV that significant performance increase (up to a speedup factor of 10) can be achieved by optimizing to a specific problem.

Beside tuning the parameter τ (respectively β) for a specific system, an intelligent implementation would handle the GPU memory problem by restarting the preconditioner setup to drop more elements when needed.

C. Mixed Precision Implementations

In the previous section, we have observed that the performance of the multi-elimination preconditioned iterative method is sensitive to the threshold τ, impacting the approx-

				MCGS-GMRES		*double LU solve*		*single LU solve*	
matrix	l	n	nnz	# iters	time [s]	# iters	time [s]	# iters	time [s]
APACHE_2	14	7637	60615	293	7.61	292	3.65	293	3.04
ECOLOGY_2	14	3568	5588	780	8.74	779	16.39	779	10.98
G2_CIRCUIT	7	8871	97817	359	10.03	358	3.62	359	2.49
G3_CIRCUIT	11	3865	4811	511	5.84	511	5.32	512	5.42
LAPLACE_2D_1M	13	5094	15270	338	4.37	338	3.52	338	3.41
OFFSHORE	2	9493	384617	1313	431.20	1323	15.02	1310	9.59
STOCF-1465	4	6218	9012	4399	825.22	4375	55.91	4395	52.06
THERMAL2	12	9891	14375	915	14.57	915	17.87	916	13.93

TABLE V: Factorization sequence characteristics (level count l, n and nnz of the bottom-level system) and top-level solver performance using different types of bottom-level solvers. MCGS-GMRES denotes a framework similar to [19]: a GMRES method using a multi-coloured symmetric Gauss-Seidel preconditioner and a relative residual stopping criterion of $1e-8$. *double LU solve* and *single LU solve* denote the sparse/direct hybrid approach using an LU factorization of the bottom-level problem and solving in either double or single precision. The factorization threshold is set to $\beta = 0.1$.

		A_1		*double ts*		*single ts*	
β	l	n	nnz	# iters	time [s]	# iters	time [s]
$1e-4$	6	4079	4417	682	7.94	1380	16.11
$5e-4$	6	3489	3635	992	11.03	1259	13.93
$1e-3$	6	932	944	1242	11.98	1405	13.62
$5e-3$	5	8351	11311	1830	29.28	1982	26.78
$1e-2$	5	4773	6438	2205	25.54	2230	25.61
$5e-2$	5	2364	3104	3406	32.93	3415	33.86
$1e-1$	4	6218	9012	4375	55.87	4393	52.28
$5e-1$	3	9769	11863	6826	120.41	6853	93.24
$1e+0$	3	4741	5273	7928	86.52	7936	85.35
$5e+0$	3	465	471	11403	93.94	11412	94.81
$1e+1$	3	99	99	12826	114.63	12827	104.00

TABLE IV: Influence of β on the depth of the factorization sequence (level depth l), bottom-level linear system (A_1) characteristics and top-level solver performance using either double or single precision triangular solve (labelled *double ts* and *single ts*, respectively). The top-level iteration method is applied to case STOCF-1465 with a relative residual stopping criterion of $1e-6$.

imation accuracy provided by the preconditioner, which in turn influences the iteration number of the top-level iterative method. In this section we focus on the bottom-level solver. While Saad proposed a SOR-preconditioned GMRES solver for the bottom-level system in the original paper on multi-elimination preconditioners [19], we argue that replacing the iterative bottom-level solver by a one-time factorization phase in the preconditioner setup, followed by triangular solves in the distinct iteration steps, may provide significant performance improvement for implementations using long reduction sequences on graphics processing units.

Motivated by the inherently approximate nature of the multi-elimination using threshold strategies, we address the question of potential performance benefits obtained by implementing the forward and backward triangular solve of the bottom-level problem in the distinct iteration steps in a less complex floating point format than working precision[3]. While we motivated this mixed precision approach with a more efficient hardware usage in Section III, it may be particularly interesting for systems accelerated by the commodity GPUs. We may expect that, using single instead of double precision, we again sacrifice some accuracy of the preconditioner, resulting in the need of additional top-level iterations. In the end, the trade-off between additional iterations and accelerated triangular solves determines the superiority of either implementation. In Table III and IV of the previous section we provide iteration count and runtime not only for the double precision implementation, but also the modification using single precision triangular solves for the bottom-level linear system (see columns labelled *double ts* and *single ts*, respectively). As expected, using single triangular solve, we usually need some additional iterations of the top-level solver to achieve the demanded accuracy. Especially when allowing for large fill-in via low thresholds, the top-level solver suffers from reduced accuracy of the bottom-level solve. Larger thresholds, on the other hand, already introduce rounding errors in the factorization sequence that whitewash the accuracy loss of the bottom-level solver (see the relative iteration increase in the tables). In the end, the higher iteration rate (number of iterations per second) when using single precision triangular solve may compensate for the additional iterations, resulting in a reduced time-to-solution (see Figure 2).

For further investigation, we compare in Table V the performance of the multi-elimination preconditioner using different bottom-level solvers integrated into a CG top-level solver. While we choose a self-adaptive level depth (recursive factorization until matrix size $n < 12,000$), the dropping threshold τ is fixed by equation (8).

Porting the method proposed in [19] to GPUs, we do some minor modifications allowing for a fair comparison and higher efficiency. First, we replaced the SOR preconditioner in the bottom-level GMRES solver by a multi-coloured symmetric Gauss-Seidel preconditioner, as the sequential nature of SOR would obviously result in very poor performance of the GPU implementation. Furthermore, we choose different relative residual stopping criterion of $1e-8$ for the preconditioned GMRES bottom-level solver. This allows a fair comparison to the hybrid approach, as it enables the top-level solver

[3]The LU factorization in the preconditioner preprocessing phase was implemented in double precision for both implementations, using single precision triangular solves; the matrices were converted afterwards.

converging within a similar number of iterations, like the implementation using an LU factorization for the bottom-level problem, and the double- or single-precision triangular solved in the distinct iteration steps. While the overall solver performance is determined by the trade-off between preconditioner accuracy and global iteration count, a computationally more expensive top-level solver benefits from reducing iterations.

A first observation from the performance results in Table V is that all bottom-level solver implementations provide similar preconditioner accuracy: we observe only small variances in the iteration count of the top-level solver. As expected, the single precision triangular solves may provide less accurate results, which results in a few additional iterations of the top-level CG solver. For the OFFSHORE matrix however, *lucky rounding* causes even faster convergence. In terms of runtime, the mixed precision approach is superior for all problems except the G3_CIRCUIT test case. Comparing against the implementation using a multi-coloured Gauss-Seidel preconditioned GMRES bottom-level solver, the double precision triangular solve is superior for all problems except for the ECOLOGY_2and THERMAL2 matrix. Switching to the mixed precision implementation, we outperform the preconditioned GMRES bottom-level solver also for THERMAL2, but fail in the ECOLOGY_2 case. The reason might be that the multi-elimination factorization sequence results in an almost diagonal bottom-level problem, which favors iterative solvers. While we alleviate all other problems by replacing the iterative bottom-level solver with the mixed precision direct solution process, the improvements depend on the sparsity of the bottom-level problem. Lower thresholds allowing for higher accuracy and increased fill-in result in dense matrices on the bottom-level, especially for unstructured matrices like OFFSHORE and STOCF-1465. For these, we achieve significant speedup factors of 44 and 15, respectively.

From the experimental results, we conclude that when implementing multi-elimination preconditioners on GPUs and allowing for long factorization sequences, it is beneficial to replace the iterative solvers by a direct solve for the bottom-level problem. Furthermore, it seems to be reasonable to use single precision triangular solves as the default configuration when comparing against other preconditioners in the next section.

D. Performance Comparison

In this section we want to quantify the performance of our mixed precision multi-elimination implementation for GPUs by comparing with well-established preconditioners. The reference implementations we compare against are, like the multi-elimination preconditioned Conjugate Gradient we benchmark as the top-level solver, taken from the PARALUTION [15] (version 0.4.0) open source library, which ensures a fair comparison through sharing the same routines for the vector and matrix operations, and the same level of GPU-specific optimization.

In Table VI we compare iteration count and runtime with a plain CG solver, and enhanced versions using a multi-colored symmetric Gauss-Seidel [16], ILU-0 [20], [17] or multi-colored ILU-(0) [16] preconditioner, respectively. For the multi-elimination, we chose a self-adaptive level-depth resulting in a bottom-level linear system of size $n < 12,000$, and a threshold controlling the fill-in according to (8) and $\beta = 0.1$ in Section V-B. Like previously stated, we execute the triangular solve in the iterations using single precision.

The runtime results in Table VI show that our self-adaptive mixed precision multi-elimination implementation is highly competitive to the other preconditioners. In direct comparison to the, algorithmically most similar, other algebraic preconditioners (ILU-0 and multi-colored ILU-0), multi-elimination outperforms the multi-colored ILU-0 for APACHE_2, G3_CIRCUIT and THERMAL2, and the plain ILU-0 for all problems. The premise that preconditioners must be chosen carefully with respect to a particular problem is supported by the optimization potential revealed in Section V-B: choosing an optimized threshold, the multi-elimination preconditioner outperforms the multi-colored ILU-0 also for the STOCF-1465 matrix.

VI. Summary and Future Research

Porting multi-elimination preconditioners to GPUs, we have proposed to replace the iterative solution of the bottom level by a direct solve in the preconditioner setup phase and forward-backward triangular solves in the distinct iteration steps. Solving the bottom-level system via a direct solver offers the connection to dense linear algebra, and the possibility to precondition the distinct iteration steps by highly-optimized triangular solves based on a factorization in the preprocessing phase. Furthermore we have proposed to employ these triangular solves in single precision for higher resource utilization and revealed in numerical tests that in a mixed precision implementation the additional top-level iterations are often compensated for by a higher iteration rate. Numerical experiments confirmed that this hybrid approach of mixing direct and iterative methods is in many cases beneficial to the runtime performance of the multi-elimination preconditioner. Implementing a self-adapting level-depth, we have shown that the mixed precision multi-elimination preconditioner is able to efficiently leverage the computational power of the GPU and outperforms the un-preconditioned top-level solver and some of the most commonly used preconditioning techniques for numerous problems. Future research directions are given by the fact that the performance of a multi-elimination preconditioned solver is very dependent on the threshold controlling the fill-in, and the time-to-solution can be decreased significantly when optimizing to a specific problem. We will furthermore investigate how to modify the multi-elimination framework for nonlinear problems.

Acknowledgments

This work has been supported by the Linnaeus centre of excellence UPMARC, Uppsala Programming for Multicore Architectures Research Center, DOE grant #DE-SC0010042,

	CG		MCGS-CG		ILU0-CG		MCILU0-CG		MPME-CG	
matrix	# iters	time [s]	# iters	time [s]	# iters	time [s]	# iters	time [s]	# iters	time [s]
APACHE_2	3970	5.02	1677	5.22	643	9.63	1438	4.07	293	3.04
ECOLOGY_2	5391	8.20	2784	8.94	1700	64.03	2854	8.35	799	10.98
G2_CIRCUIT	8901	3.84	907	1.29	481	6.02	859	1.12	359	2.49
G3_CIRCUIT	12712	29.68	1329	9.01	680	33.77	1242	7.72	512	5.42
LAPLACE_2D_1M	1633	2.53	817	2.63	537	19.30	817	2.38	338	3.41
OFFSHORE	-	-	628	4.92	365	23.22	487	3.57	1314	9.59
STOCF-1465	-	-	66042	1187.59	2338	158.37	16698	290.38	4388	52.06
THERMAL2	4587	9.63	2151	18.33	1945	54.13	2096	16.79	916	13.93

TABLE VI: Runtime and iteration comparison for (preconditioned) CG using a multi-colored symmetric Gauss-Seidel (MCGS-), ILU-0 (ILU0-), multi-colored ILU-0 (MCILU0-) or the mixed precision multi-elimination (MPME-) preconditioner. In the MPME-CG, the level-depth is self-adaptive, and the parameter β was chosen according to (8).

NVIDIA, and the National Science Foundation.

REFERENCES

[1] Whitepaper: Nvidia's next generation cuda compute architecture: Kepler gk110, 2012.

[2] Software distribution of MAGMA version 1.4. http://icl.cs.utk.edu/magma/, 2013.

[3] Hartwig Anzt. *Asynchronous and Multiprecision Linear Solvers - Scalable and Fault-Tolerant Numerics for Energy Efficient High Performance Computing* . PhD thesis, Karlsruhe Institute of Technology, Institute for Applied and Numerical Mathematics, Nov. 2012.

[4] Hartwig Anzt, Tobias Hahn, Vincent Heuveline, and Björn Rocker. GPU Accelerated Scientific Computing: Evaluation of the NVIDIA Fermi Architecture; Elementary Kernels and Linear Solvers. In *Proceedings of HipHaC 2011*, 2011.

[5] Hartwig Anzt, Vincent Heuveline, and Björn Rocker. An error correction solver for linear systems: evaluation of mixed precision implementations. In *VECPAR'10*, pages 58–70, 2010.

[6] Marc Baboulin, Alfredo Buttari, Jack J. Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, Jakub Kurzak, and Stanimire Tomov. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications*, 180(12):2526–2533, 2009.

[7] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[8] Dietrich Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*, volume 3. Cambridge University Press, 2007.

[9] Alfredo Buttari, Jack J. Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak. Mixed precision iterative refinement techniques for the solution of dense linear systems. *Int. J. of High Perf. Comp. & Appl.*, 21(4):457–486, 2007.

[10] D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware–oriented native–, emulated– and mixed–precision solvers in FEM simulations. *Int. J. of Parallel, Emergent and Distr. Systems*, 22(4):221–256, 2007.

[11] Magnús Halldórsson and Jaikumar Radhakrishnan. Greed is good: approximating independent sets in sparse and bounded-degree graphs. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, STOC '94, pages 439–448, New York, NY, USA, 1994. ACM.

[12] Michael R Leuze. Independent set orderings for parallel matrix factorization by gaussian elimination. *Parallel Computing*, 10(2):177 – 191, 1989.

[13] Ruipeng Li and Yousef Saad. Gpu-accelerated preconditioned iterative linear solvers.

[14] M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1053, 1986.

[15] Dimitar Lukarski. PARALUTION project. http://www.paralution.com/.

[16] Dimitar Lukarski. *Parallel Sparse Linear Algebra for Multi-core and Many-core Platforms - Parallel Solvers and Preconditioners*. PhD thesis, Karlsruhe Institute of Technology (KIT), Germany, 2012.

[17] Maxim Naumov. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. Technical report, NVIDIA, 2011.

[18] J.M Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425 – 440, 1986.

[19] Y. Saad. Ilum: A multi-elimination ilu preconditioner for general sparse matrices. *SIAM J. Sci. Comput*, 17:830–847, 1999.

[20] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[21] Yousef Saad and Jun Zhang. Bilum: Block versions of multi-elimination and multi-level ilu preconditioner for general sparse linear systems. *SIAM J. SCI. COMPUT*, 20:2103–2121, 1997.

[22] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

[23] Lu Yao, Wei Cao, Zongzhe Li, Yongxian Wang, and Zhenghua Wang. An improved independent set ordering algorithm for solving large-scale sparse linear systems. In *Intelligent Human-Machine Systems and Cybernetics (IHMSC), 2010 2nd International Conference on*, volume 1, pages 178–181, 2010.