Asynchronous Iterative Algorithm for Computing Incomplete Factorizations on GPUs

Edmond Chow[1], Hartwig Anzt[2(✉)], and Jack Dongarra[2]

[1] Georgia Institute of Technology, Atlanta, GA, USA
[2] University of Tennessee, Knoxville, TN, USA
hanzt@icl.utk.edu

Abstract. This paper presents a GPU implementation of an asynchronous iterative algorithm for computing incomplete factorizations. Asynchronous algorithms, with their ability to tolerate memory latency, form an important class of algorithms for modern computer architectures. Our GPU implementation considers several non-traditional techniques that can be important for asynchronous algorithms to optimize convergence and data locality. These techniques include controlling the order in which variables are updated by controlling the order of execution of thread blocks, taking advantage of cache reuse between thread blocks, and managing the amount of parallelism to control the convergence of the algorithm.

1 Introduction

Asynchronous algorithms, with their ability to tolerate memory latency, form an important class of algorithms for modern computer architectures. In this paper, we develop a GPU implementation for a recently proposed asynchronous iterative incomplete factorization algorithm [4]. In particular, we consider the following techniques to enhance data locality and convergence that may be considered non-traditional as they are not strictly allowed in standard GPU implementations.

- In an asynchronous iterative method, variables are updated using values of other variables that are currently available, rather than waiting for the most updated values of these other variables (this will be made more precise later). The rate of convergence of the method may depend on the order in which variables are updated. In traditional GPU computations, there is no defined ordering in which thread blocks are executed, making it impossible to control the update order. For the NVIDIA K40c GPU, however, we were able to determine the order in which thread blocks are executed, thereby allowing us to control the order of the updates of the variables.
- Efficient GPU performance requires that GPU thread blocks reuse data brought into shared memory. Shared memory must be configured as cache when the working set is large, otherwise few thread blocks can run in parallel.

J.M. Kunkel and T. Ludwig (Eds.): ISC High Performance 2015, LNCS 9137, pp. 1–16, 2015.
DOI: 10.1007/978-3-319-20119-1_1

However, without control over the order in which thread blocks are executed, *temporal locality* cannot be properly exploited. Since we are able to control the execution order of thread blocks on the K40c GPU, we can assure that data in cache is efficiently used. As mentioned, traditionally this order is not known.
- In GPU computing, the amount of parallelism is typically not controlled; work is scheduled onto all multiprocessors. Using less parallelism and updating fewer variables simultaneously in order to use more recently updated variables may lead to faster convergence than using more parallelism. We investigate this tradeoff by using the unconventional approach of controlling the occupancy, or the fraction of the maximum number of threads executing simultaneously per multiprocessor.

There has been some past work on asynchronous algorithms on GPUs for linear iteration methods. See [1] for a comprehensive study on block-asynchronous Jacobi iterations. Venkatasubramanian et al. [18] solve a 2D Poisson equation on a structured grid using asynchronous stencil operations on a hybrid CPU/GPU system. Contassot-Vivier et al. [5] investigated the impact of asynchronism when solving an advection-diffusion problem with a GPU-accelerated PDE solver. The work in this paper on investigating data locality and update order for asynchronous algorithms on GPUs is new.

This paper is organized as follows. In Sect. 2, we provide background on incomplete factorizations and the recently proposed iterative ILU algorithm. Then in Sect. 3, we discuss the implementation of the algorithm on GPUs and consider how convergence and data locality are impacted by the order in which computations are performed. Experimental tests with this implementation are reported in Sect. 4, and we conclude in Sect. 5.

2 Iterative ILU Algorithm

Preconditioners are a critical part of solving large sparse linear systems via iterative methods. A popular class of preconditioners is incomplete LU (ILU) factorizations. These preconditioners are generally computed using a Gaussian elimination process where small non-diagonal elements are dropped in some way. The problem-independence of ILU preconditioners makes this class attractive for a wide range of problems, particularly for optimization problems.

An ILU factorization is the approximate factorization of a nonsingular sparse matrix A into the product of a sparse lower triangular matrix L and a sparse upper triangular matrix U ($A \approx LU$), where nonzeros or fill-in are only permitted in specified locations, (i, j) of L and U. Define the sparsity pattern S to be the set of matrix locations where nonzeros are allowed, i.e., $(i, j) \in S$ implies that entry l_{ij} in matrix L is permitted to be nonzero ($i \geq j$), or that entry u_{ij} in matrix U is permitted to be nonzero ($i \leq j$). The set S always includes the diagonal of the L and U factors so that these factors are nonsingular. The basic algorithm, called ILU(0), approximates the LU factorization by allowing only nonzero elements in L and U that are nonzero in A. To enhance the accuracy of

the preconditioner, one can allow for additional fill-in in the incomplete factors. The choice of S can be made either before the factorization, or may be made dynamically, during elimination. The classical factorization algorithm, based on Gaussian elimination, is inherently sequential, but some parallelism does exist, as it is usually possible to find multiple rows that can be eliminated in parallel, i.e., those that only depend on rows that already have been eliminated; see [17] for an overview. Parallelism is usually reduced when more fill-in is allowed. Multicoloring and domain decomposition reordering can be used to enhance the available parallelism [2,7,11,16]. However, these approaches generally have limited scalability, as they are unable to exploit the computing performance of thousands of light-weight cores that we expect in future HPC architectures [3].

For large amounts of parallelism, a new algorithm for computing ILU factorizations was recently proposed [4], and is the focus of this paper. This algorithm uses the property of ILU factorizations that

$$(LU)_{ij} = a_{ij}, \quad (i,j) \in S \tag{1}$$

where $(LU)_{ij}$ denotes the (i,j) entry of the product of the computed factors L and U, and a_{ij} is the corresponding entry in matrix A. For $(i,j) \in S$, the iterative ILU algorithm computes the unknowns l_{ij} (for $i > j$) and u_{ij} (for $i \leq j$) using the bilinear constraints

$$\sum_{k=1}^{\min(i,j)} l_{ik} u_{kj} = a_{ij}, \quad (i,j) \in S \tag{2}$$

which corresponds to enforcing the property (1). We use the normalization that the diagonal of the lower triangular L is fixed to one. Thus we need to solve a system of $|S|$ equations in $|S|$ unknowns.

To solve this system, Ref. [4] proposed writing

$$l_{ij} = \frac{1}{u_{jj}} \left(a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right), \ i > j \qquad u_{ij} = a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}, \ i \leq j \tag{3}$$

which has the form $x = G(x)$, where x is a vector containing the unknowns l_{ij} and u_{ij} for $(i,j) \in S$. The equations are then solved by using the fixed-point iteration $x^{(p+1)} = G(x^{(p)})$, for $p = 0, 1, \ldots$, starting with some initial $x^{(0)}$. See [4] for details on the convergence of this method. In brief, it can be proven that the iteration is locally convergent for standard (synchronous) and asynchronous iterations [9].

The iterative ILU algorithm, which solves the Eq. (2), is given in Algorithm 1. The actual implementation that served as the basis for this study can be found in the MAGMA open-source software library [10]. Each fixed-point iteration updating all the unknowns is called a "sweep." In this algorithm, an important question is how to choose the initial values of the l_{ij} and u_{ij} variables (line 1). In many applications, a natural initial guess is available; for example, in time-dependent problems, the L and U computed at the previous time step may be an

excellent initial guess for the current time step. In other cases, there may be no natural initial guess. In [4], the matrix A is symmetrically scaled to have a unit diagonal, and the initial guess for L and U are then chosen to be the lower and upper parts of A, respectively. We also use this initial guess for the experiments in this paper.

Algorithm 1. Fine-Grained Parallel Incomplete Factorization

```
1  Set unknowns l_ij and u_ij to initial values
2  for sweep = 1, 2, ... until convergence do
3      parallel for (i, j) ∈ S do
4          if i > j then
5              l_ij = ( a_ij − Σ_{k=1}^{j−1} l_ik u_kj ) / u_jj
6          else
7              u_ij = a_ij − Σ_{k=1}^{i−1} l_ik u_kj
8          end
9      end
10 end
```

We conclude this section by pointing out several features of this algorithm that make it different than existing methods, and relevant to parallel computing on emerging architectures:

- The algorithm is fine-grained, allowing for scaling to very large core counts, limited only by the number of nonzero elements in the factorization.
- The algorithm does not need to use reordering to enhance parallelism, and thus reorderings that enhance the accuracy of the incomplete factorization can be used.
- The algorithm can utilize an initial guess for the ILU factorization, which cannot be exploited by conventional ILU factorization algorithms.
- The bilinear equations do not need to be solved very accurately since the ILU factorization itself is only an approximation.

3 GPU Implementation

3.1 General Parallelization Strategy

When multiple processors are available, an iteration based on $x^{(p+1)} = G(x^{(p)})$ may be parallelized by assigning each processor to compute a subset of the components of $x^{(p+1)}$ using values of $x^{(p)}$, such that each component is updated by exactly one processor. This is called a synchronous iteration, as all values of $x^{(p)}$ must generally be computed before the computation of $x^{(p+1)}$ may start. In contrast, an asynchronous iteration is one where the computation of components of x may use the latest components of x that are available. Convergence may

be faster than the synchronous iteration because more updated values are used (e.g., Gauss-Seidel type of iteration compared to Jacobi type) or may be slower than synchronous iteration in the case that some components are rarely updated. In general, there may be a tradeoff between parallelism and convergence: with less parallel resources, the asynchronous iterations tend to use "fresher" data when computing updates; with more parallel resources, the iterations tend to use older data and thus converge more slowly.

For GPU computing, subsets of the components of x are assigned to GPU thread blocks. Each thread block updates the components of x assigned to it. The thread blocks are scheduled onto GPU multiprocessors. Within each thread block, the components of x are updated simultaneously (in Jacobi-like fashion). As there are generally more thread blocks than multiprocessors, some thread blocks are processed before others, and thus update their components of x before others. Thus some thread blocks within one fixed-point sweep may use newer data than others (in Gauss-Seidel-like fashion). However, there is no guarantee of the order in which thread blocks are scheduled onto multiprocessors. Overall, the iteration may be considered to be "block-asynchronous" [1].

3.2 Component Update Order

The convergence rate of an asynchronous fixed-point iteration for the system of equations $x = G(x)$ may depend on the order in which the components of x are updated, particularly if some equations have more dependencies on components of x than others. Specifically, for the computations (3) that are performed in the new ILU formulation, there is a tree of dependencies between the variables (components) being updated. Thus convergence will be faster if the asynchronous updates of the variables are ordered roughly following these dependencies. Such orderings are called "Gaussian elimination orderings" in [4].

On GPUs, each thread block is responsible for updating a given set of variables. Unfortunately, there is no guarantee of thread block execution order for current generation GPU programs. Using a simple kernel that records the order in which thread blocks are executed, we observed on the NVIDIA K40c GPU that block indices are always assigned in order of execution of the thread blocks. (On some earlier GPU models, we observed that the order of execution appears random.) Using this result, we can explicitly set a component update order that will be approximately followed by the asynchronous iterations (approximate because the iterations are asynchronous). In Sect. 4, we will test the effect of component update order on convergence rate.

3.3 Data Locality and Cache Reuse

GPU access of data in global memory (off-chip DRAM) is expensive, and the performance of any GPU code depends on how efficiently memory is reused after it has been brought into cache or shared memory. In this section, we consider how to partition the computational work into thread blocks in order to maximize

data reuse. We note that such partitionings also affect component update order and thus convergence.

Each thread is associated with an element $(i, j) \in S$. Each thread thus computes either l_{ij} or u_{ij} in Eq. (3). This computation requires reading row i of L and column j of U. This row and column may be reused to compute other elements in S. Thus we have the problem of partitioning the $(i, j) \in S$ among a given number of thread blocks such that threads in the thread block require, in aggregate, as few rows of L or columns of U as possible. (In the above, only part of the row or column is needed by each thread, due to the upper limit on the summations in (3), but this will only affect our analysis by a constant factor.)

Another way to view this problem is to maximize the number of times each row of L and each column of U is reused by a thread block. We define the reuse factor of thread block l as

$$f_{\text{reuse}}(l) := \frac{1}{2}\left(\frac{|S_l|}{m_l} + \frac{|S_l|}{n_l}\right)$$

where $|S_l|$ is the number of elements of S assigned to thread block l, and where m_l and n_l are the number of rows of L and columns of U, respectively, required by thread block l. The first term in the brackets is the average number of times that the rows are reused, while the second is the average number of times that the columns are reused. If the elements of S are assigned arbitrarily to the thread blocks, then the reuse factor is 1 in the worst case. For simplicity, we assume below that $m_l = n_l$, since m_l and n_l approximately equal will give higher reuse factors than m_l and n_l being very different.

We first consider a matrix corresponding to a 2D mesh problem using a 5-point stencil. Figure 1 (middle) shows the sparsity pattern of the matrix corresponding to a 6×6 mesh (left). The mesh has been partitioned into 9 subdomains of size 2×2, and the rows and columns of the matrix are ordered such that the rows and columns corresponding to the same subdomain are ordered together. Horizontal and vertical lines are used to separate the rows and columns corresponding to subdomains.

Now, assuming the ILU(0) case, each nonzero in the matrix corresponds to an element of S to assigned to a thread block. If we use the partitioning of the mesh into subdomains just described, then thread block l is assigned the nonzeros a_{ij} corresponding to edges (i, j) in the mesh within and emanating from subdomain l. For one subdomain, these correspond to the nonzeros marked as red squares in Fig. 1 (middle). For a partitioning of the mesh into subdomains of size $b \times b$, each thread block is assigned $5b^2$ edges (b^2 nodes in the subdomain and 5 edges per node). A typical thread block (corresponding to an interior subdomain) also requires $(b + 1)^2 - 1$ rows of L and the same number of columns of U to be read. Thus the reuse factor is $f_{\text{reuse}}(l) := \frac{5b^2}{(b+1)^2-1}$ in this case. The limit of the maximum size of the reuse factor is 5, for large values of b. Note that $b \times b$ is a blocking of the mesh, while $m_l \times m_l$ used above is a blocking of the matrix.

In general, if a regular mesh is partitioned into subdomains of size $b \times b$ and an s-point stencil is used, then there are $s \cdot b^2$ matrix entries corresponding to the

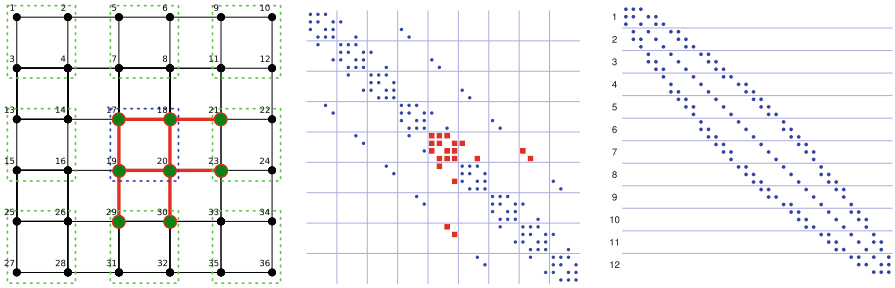

Fig. 1. A 6×6 mesh using a 5-point stencil partitioned into 9 subdomains of size 2×2 (left). The corresponding matrix (middle) orders rows and columns subdomain by subdomain. The bold edges in the mesh correspond to the nonzeros in the matrix marked with squares, and identify the elements assigned to one of the thread blocks. The right side shows the partitioning of the matrix after applying reverse Cuthill-McKee (RCM) and using 12 partitions.

subdomain, and the number of rows needed by the subdomain is b^2 (highest order term). The reuse factor therefore approaches s for large block sizes, showing that a larger stencil gives rise to a larger reuse factor.

In the 3D case, if a regular mesh is partitioned into subdomains of size $b \times b \times b$ and an s-point stencil is used, then there are $s \cdot b^3$ matrix entries corresponding to the subdomain, and the number of rows needed by the subdomain is b^3 (highest order term). The reuse factor again approaches s for large block sizes. As apparent from this analysis, 3D problems do not inherently have a larger reuse factor than 2D problems, except for generally using larger stencils. We note that in the sparse case, the maximum reuse factors are bounded independent of the partitioning, whereas for the dense case, the reuse factors increase with the size of the partitioning, may however be limited by the size of the shared memory.

3.4 Cache Reuse Between Thread Blocks

The working set of rows of L and columns of U needed by each thread block can be large, especially for large values of the blocking parameter b. However, if this working set can be shared between thread blocks by using the shared L2 cache [13], the communication volume from global memory can be reduced, compared to the case where each thread block uses its own scratchpad memory. To this end, in this section, we explore the idea of sharing the cache between thread blocks, i.e., one thread block using data brought into the L2 cache by another thread block. This idea is non-traditional because GPU programs generally assume that thread blocks that are not communicating through global memory are completely independent. This is because there is no guarantee of the order in which thread blocks are executed.

By using our previous observation in Sect. 3.2 that thread blocks are assigned indices in order of execution, we were able to verify through a simple performance

test that thread blocks can indeed reuse data brought into the multiprocessor cache by an earlier thread block. Thus we can assign work to thread blocks such that reuse of cache between thread blocks is high. The right side of Fig. 1 shows a simple example for the matrix corresponding to the 6×6 mesh on the left. The matrix has been reordered to reduce its bandwidth using reverse Cuthill-McKee (RCM) ordering. Horizontal lines show a partitioning of the elements of S, and assume that the partitions are assigned to thread blocks such that the partitions are executed from top to bottom. From the figure, it can be observed that each partition requires different rows of L; these rows are reused within a thread block but not across thread blocks. We also observe that partitions numbered nearby use a very similar set of columns of U. These columns of U may be shared across thread blocks using cache. Experiments are reported in Sect. 4 to test the cache behavior of this partitioning of S and this ordering of the thread blocks.

3.5 Parallelism Vs Convergence

The convergence rate of an asynchronous iteration depends on the amount of parallelism used in the iteration, with more parallelism usually resulting in a reduced convergence rate. We aim to control the amount of parallelism in order to investigate the tradeoff between parallelism and convergence. NVIDIA provides no interface for direct manipulation of parallelism; thus we will control the amount of parallelism indirectly.

To quantify the parallelism of a code running on GPUs, we may use the concept of "occupancy." While occupancy is defined as the ratio between the number of threads (grouped in thread blocks) that are scheduled in parallel and the hardware-imposed thread limit (threads per streaming multiprocessor, `tpsm`), certain algorithms may benefit from using less than the maximum occupancy [19]. A metric for quantifying the *actual* parallelism is the number of executed instructions per cycle (IPC), which reflects not only the number of active threads, but also stalls due to communication bottlenecks.

To indirectly control the number of thread blocks that are simultaneously scheduled onto a multiprocessor, note that the available shared memory, registers, the number of individual threads and thread blocks are limited for each streaming multiprocessor, which bounds the total number of thread blocks being executed in parallel. Thus we can artificially reduce parallelism by explicitly allocating scratchpad memory that we do not use in the kernel execution.

The multiprocessor's local memory is partitioned between scratchpad memory and L1 cache. We have configured shared memory to maximize cache and minimize scratchpad memory. The minimum amount of shared memory that can be configured is 16,384 bytes per multiprocessor on the K40X architecture [13]. In compute capability 3.5, the value of `tpsm` is 2048. Therefore, requesting 1,024 bytes or less of scratchpad memory for each thread block consisting of 128 threads results in 16 thread blocks running in parallel and 100 % multiprocessor occupancy, while doubling the allocated scratchpad memory reduces the number of active blocks to 8, and decreases the occupancy to 50 %. In Sect. 4.6, we use this technique to control parallelism and observe its affect on the convergence rate and performance of the asynchronous iterations.

3.6 Implementation Issues

Our CUDA kernel is designed to perform one sweep of the iterative ILU algorithm (Algorithm 1) for computing the factorization $A \approx LU$. Additional sweeps are performed by launching this kernel multiple times. The input to the kernel are arrays corresponding to matrices A, L, and U. The arrays for L and U, stored in CSR and CSC format, respectively, contain an initial guess on input and the approximate factorizations on output.

The matrix A, stored in coordinate (COO) format, can be regarded as a "task list." The thread with global thread index `tid` is assigned to update the variable corresponding to element `tid` in the COO array. Effectively, thread block 0 is assigned the first `blockDim` elements in the COO array, thread block 1 is assigned the next `blockDim` elements, etc., (where `blockDim` is the number of threads in a thread block). By changing the order in which the elements are stored in the COO array, the order of the updates can be controlled. Note that changing this order does not change the ordering of the rows and columns of the matrix A.

Reads of the arrays corresponding to matrix A are coalesced. However, each thread in general reads different rows of L and columns of U that are not coalesced, so that we rely on this data being read into cache and reused as much as possible, as described above.

Significant thread divergence will occur if the partial sum (lines 5 and 7 of Algorithm 1) computed by a thread contains a different number of addends than for other threads in the same warp. Thus, to minimize the cost of divergence, the partial sums for the updates handled by threads in the same warp should be of similar length. This, however, conflicts with optimizing component assignment for cache reuse, and we experimentally identified that data locality is more important.

To end this section, we summarize the parameters that affect the convergence behavior and data locality of the kernel.

- Task list ordering: the ordering of the elements of S in the task list. Except when specified otherwise, the *default task list ordering* is as follows: the elements of S are ordered such that if the elements were placed in a matrix, the elements are ordered row by row, from left to right within a row. This is a Gaussian elimination ordering. The order in which variables are updated can be changed by changing the task list ordering.
- Thread block order: the order of execution of the thread blocks. Except when specified otherwise, the thread blocks are ordered by increasing thread block index, i.e., the first thread block is assigned the first chunk of the task list, etc. We can also order the thread blocks in reverse order or in a random, arbitrary order.
- Matrix ordering: the ordering of the rows and columns of the matrix. Changing the matrix ordering changes the problem. We use the symmetric RCM ordering of the matrix, except for the finite difference discretizations of the Laplace problem L2D and L3D, for which the natural ordering can be used.

The RCM and natural orderings are good choices for computing accurate incomplete factorization preconditioners [8].

4 Experimental Results

4.1 Experimental Setup

Our experimental setup is an NVIDIA Tesla K40c GPU (Rpeak 1,682 GFLOP/s) with a two socket Intel Xeon E5-2670 (Sandy Bridge) host. The iterative incomplete factorization GPU kernels were implemented in CUDA version 6.0 [15] and use a default thread block size of 128. The conjugate gradient linear solver and the iterative incomplete factorization routine are taken from the MAGMA open-source software library [10]. The sparse triangular solve routines are from the NVIDIA cuSPARSE library [14]. The results below used double precision computations for the iterative ILU code; timings were 20–40 percent slower than for single precision computations.

The test matrices were selected from Tim Davis's sparse matrix collection [6] and include the problems used by NVIDIA for testing the ILU factorization code in cuSPARSE [12]. We have reordered these test matrices using RCM reordering. Additionally, we use test matrices arising from a finite difference discretization of the Laplace operator in 2D and 3D with Dirichlet boundary conditions. For the 2D case, a 5-point stencil was used on a 1024×1024 mesh, and for the 3D case, a 27-point stencil was used on a $64 \times 64 \times 64$ mesh. All test matrices (Table 1) are symmetric positive definite and we used the symmetric, or incomplete Cholesky (IC) version of the iterative ILU algorithm [4]. However, we still refer to the algorithm as the *iterative ILU algorithm* for generality.

Table 1. Test matrices.

Name	Abbrev.	Nonzeros n_z	Size n
APACHE2	APA	4,817,870	715,176
ECOLOGY2	ECO	4,995,991	999,999
G3_CIRCUIT	G3	7,660,826	1,585,478
OFFSHORE	OFF	4,242,673	259,789

Name	Abbrev.	Nonzeros n_z	Size n
PARABOLIC_FEM	PAR	3,674,625	525,825
THERMAL2	THM	8,580,313	1,228,045
LAPLACE2D	L2D	5,238,784	1,048,576
LAPLACE3D	L3D	6,859,000	262,144

4.2 Convergence Metrics

For an incomplete Cholesky factorization, $A \approx LL^T$, we measure how well the factorization works as a preconditioner for the preconditioned conjugate gradient (PCG) method for solving linear systems. Thus we will report PCG solver iteration counts. Linear systems were constructed using a right-hand side of all ones. The iterations start with a zero initial guess, and the iterations are stopped when the residual norm relative to the initial residual norm has been reduced beyond 10^{-6}.

We also measure the *nonlinear residual norm* $\|(A - LL^T)_S\|_F$, where the Frobenius norm is taken only over the elements in S. The expression $(A - LL^T)_S = 0$ corresponds to the nonlinear equations being solved by the fixed-point sweeps of the asynchronous iterative algorithm. Finally, we can also measure the *ILU residual norm* $\|A - LL^T\|_F$, which has been shown to correlate well with the number of PCG iterations in the SPD case [8]. This quantity is relatively expensive to compute since all elements of LL^T are required, but it can be useful for diagnostic purposes. Note that this quantity is not zero for an incomplete factorization.

We show how the above quantities change for matrices L computed using different numbers of asynchronous fixed-point sweeps, beginning with an initial guess for L (corresponding to sweep 0). In this paper, the level 0 factorizations are computed, although any sparsity pattern S can be used in the GPU kernel. We use the notation "IC" to indicate results for an incomplete factorization computed "exactly" using the conventional Gaussian elimination process.

4.3 Preconditioner Quality and Timings

We begin by demonstrating the convergence and performance of the iterative ILU algorithm using our GPU implementation. Figure 2 shows the convergence of the nonlinear residual norm (left) and ILU residual norm (right) as a function of the number of nonlinear sweeps, for several test matrices. The results show that the nonlinear residual norm converges very steadily. Also, the ILU residual norm converges very rapidly, i.e., after a small number of steps, to the ILU residual norm of the conventional ILU factorization (which is necessarily nonzero because the factorization is approximate). This suggests that the factorization computed by the new algorithm after a small number of steps may be comparable to the factorization computed by the conventional algorithm.

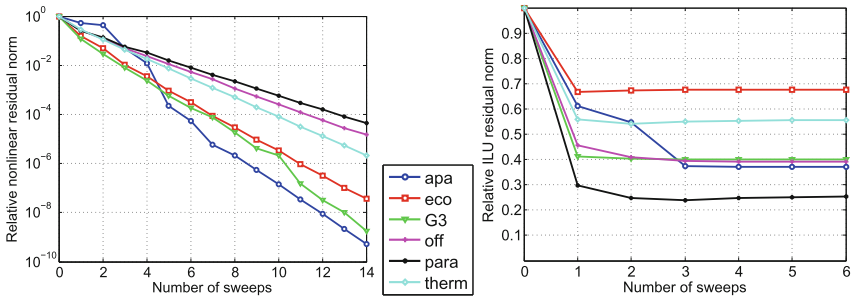

Fig. 2. Relative nonlinear residual norm (left) and relative ILU residual norm (right) for different numbers of sweeps.

This is indeed the case, as shown on the left side in Table 2, which shows the PCG solver iteration counts when the incomplete factors are used as a preconditioner. The iteration counts indicate that only a few sweeps are usually sufficient

to generate a preconditioner similar in quality to the preconditioner computed conventionally. This is consistent with the ILU residual norm typically having converged after the first 5 iterations. It is thus not necessary to fully converge the nonlinear residual.

The right side of Table 2 shows the timings for IC computed using the NVIDIA cuSPARSE library, and for the 5 sweeps of the new algorithm. Significant speedups are seen over the cuSPARSE implementation, which uses level scheduling to exploit parallelism.

Table 2. PCG solver iteration counts using preconditioners constructed with up to 5 sweeps, and timings for 5 sweeps. IC denotes the exact factorization computed using the NVIDIA cuSPARSE library. Speedup shown is that of 5 sweeps relative to IC.

	Solver iteration counts for given number of sweeps							Timings [ms]		
	IC	0	1	2	3	4	5	IC	5 swps	speedup
APA	958	1430	1363	1038	965	960	958	61.	8.8	6.9
ECO	1705	2014	1765	1719	1708	1707	1706	107.	6.7	16.0
G3	997	1254	961	968	993	997	997	110.	12.1	9.1
OFF	330	428	556	373	396	357	332	219.	25.1	8.7
PAR	393	763	636	541	494	454	435	131.	6.1	21.6
THM	1398	1913	1613	1483	1341	1411	1403	454.	15.7	28.9
L2D	550	653	703	664	621	554	551	112.	7.4	15.2
L3D	35	43	37	35	35	35	35	94.	47.5	2.0

4.4 Thread Block Ordering and Convergence

We now show the effect that thread block ordering has on the convergence of the iterative ILU algorithm. First, the ordering of the tasks follows a Gaussian elimination ordering, which is beneficial for convergence. This task list is linearly partitioned into thread blocks. We tested three thread block orderings: (1) a forward ordering of the thread blocks, i.e., the first thread block is associated with the first chunk of the task list, etc., (2) a backward ordering of the thread blocks, and (3) a random ordering of the thread blocks. For the random ordering, the results have been averaged over several trials.

Table 3 reports the time and the relative nonlinear residual norm after 5 sweeps for various problems. The forward ordering leads to the best convergence rate and the backward ordering leads to the worst convergence rate. The random ordering, corresponding to the convergence behavior of an unpredictable GPU thread block scheduling gives results in between these two. Note that there is no significant difference between the timings using the three different orderings.

Table 3. Comparison of different thread block orderings, showing time and relative nonlinear residual norm after 5 sweeps of the iterative ILU algorithm. Forward ordering gives the fastest convergence rate, and timings are not impacted.

	Order	Time [s]	Res. norm
APA	Forward	9.04e-03	2.23e-04
	Backward	8.84e-03	1.03e-02
	Random	8.68e-03	4.46e-03
ECO	Forward	6.62e-03	9.36e-04
	Backward	6.61e-03	6.23e-03
	Random	6.77e-03	3.82e-03
G3	Forward	1.20e-02	6.31e-04
	Backward	1.19e-02	6.98e-03
	Random	1.30e-02	5.09e-03
OFF	Forward	2.49e-02	1.16e-02
	Backward	2.46e-02	7.21e-02
	Random	2.50e-02	5.93e-01

	Order	Time [s]	Res. norm
PAR	Forward	5.90e-03	1.54e-02
	Backward	5.97e-03	2.83e-02
	Random	6.06e-03	2.32e-02
THM	Forward	1.54e-02	7.25e-03
	Backward	1.53e-02	1.44e-02
	Random	1.64e-02	1.04e-02
L2D	Forward	7.38e-03	1.58e-04
	Backward	7.36e-03	2.30e-03
	Random	7.35e-03	1.27e-03
L3D	Forward	4.73e-02	1.35e-03
	Backward	4.71e-02	7.17e-03
	Random	4.73e-02	3.25e-03

4.5 Data Locality

In this section we show the effect of the ordering of the task list, which affects data locality. In Sect. 3.3, orderings of the task list were proposed for problems that are discretized on regular meshes. These are based on partitioning the graph corresponding to the matrix (using $b \times b$ blockings of the 2D regular mesh was the prototypical example). In Sect. 3.4, it was proposed to enhance cache reuse *between thread blocks* by using a RCM reordering of the matrix combined with the default task list ordering.

Results are shown along with metrics from NVIDIA's NVPROF profiler in Table 4 for the L3D problem. We used a randomly permuted task list as an extreme case of disordered memory access and $b \times b \times b$ blockings of the mesh.

The results show that large block sizes give better performance. RCM ordering also gives good performance. The random case is the slowest, due to low L2 hit rate and low global load throughput, resulting in low executed instructions per cycle. For the blockings, the high L2 hit rate indicates that most data is already present in local memory, and only small amounts must be reloaded from DRAM. Luckily, we also find that orderings that have better memory access locality also lead to better convergence rates.

4.6 Parallelism Vs Convergence

Increased parallelism may result in slower convergence because variables being updated at the same time use "older" rather than "refreshed" values in their update formulas. As described in Sect. 3.6, the amount of parallelism can be controlled explicitly by allocating different amounts of scratchpad memory (that will not be used) in the kernel code. Table 5 shows several kernel configurations, for different amounts of requested scratchpad memory, and the result of running these configurations for the L3D problem. As can be observed, the theoretical

Table 4. Comparison of strategies for enhancing data locality for the L3D problem. The time and relative nonlinear residual norm are for 5 nonlinear sweeps. RCM and orderings for large block sizes $m \times m \times m$ give best results in terms of both timing and convergence.

Task list ordering	Random	2	4	8	16	32	64	RCM
Time $\times 10^{-2}$ [s]	20.0	5.54	5.37	5.15	4.96	4.89	4.78	4.92
Rel. nonlin. res. norm $\times 10^{-3}$	2.66	3.08	4.35	4.86	4.63	1.87	1.35	1.58
Global load throughput [GB/s]	83.35	237.34	247.71	249.45	251.38	253.00	258.00	252.97
DRAM read throughput [GB/s]	119.00	16.70	15.93	16.43	17.13	16.62	22.10	20.02
L2 read throughput [GB/s]	83.35	237.34	247.71	249.45	251.38	253.00	258.00	252.97
L2 hit rate (L1 reads) [%]	19.02	96.68	96.97	96.91	96.79	96.93	95.88	96.02
Global store throughput [GB/s]	1.08	3.70	3.79	3.88	3.98	4.05	3.86	3.74
Ex. instructions per cycle (IPC)	0.21	0.75	0.78	0.81	0.85	0.85	0.87	0.86

and observed occupancy decreases with increasing requested memory. For our algorithm, the IPC also generally decreases with increasing requested memory, but the relation is not exact. For example, config_1 has the highest occupancy but not the highest IPC. As expected, reducing the parallelism slightly improves the convergence rate, but at the same time, the time scales almost linearly with the inverse of parallelism quantified by IPC. Figure 3 graphs the relative nonlinear residual norm as a function of time, for different configurations. The results show that the degradation in convergence due to additional parallelism is small, and the penalty is more than compensated by the additional parallelism.

Table 5. Several configurations of the kernel code to control parallelism, and the corresponding runtime and relative nonlinear residual norm for 5 sweeps. Results are for the L3D problem.

		config_1	config_2	config_3	config_4	config_5	config_6
Requested scratchpad mem. [B]		1024	2048	3072	4096	6144	9216
Active thread blocks		16	8	5	4	2	1
Active threads		2048	1024	640	512	256	128
Theoretical occupancy [%]		100.0	50.0	31.3	25.0	12.5	6.3
Reported occupancy [%]		98.7	49.6	31.0	24.8	12.4	6.2
Global load throughput [GB/s]		258.00	269.34	209.70	175.00	92.32	46.34
DRAM read throughput [GB/s]		22.10	23.23	18.06	15.07	7.98	4.01
L2 read throughput [GB/s]		258.00	269.34	209.70	175.00	92.32	46.34
Global store throughput [GB/s]		3.86	4.04	3.15	2.62	1.38	0.69
IPC		0.8572	0.91184	0.7120	0.5941	0.3134	0.1598
5 sweeps	Time:	4.88e-02	4.57e-02	5.89e-02	7.01e-02	1.33e-01	2.62e-01
	Rel. nonlin. res. norm:	1.35e-03	1.22e-03	1.16e-03	1.14e-03	9.84e-04	8.00e-04

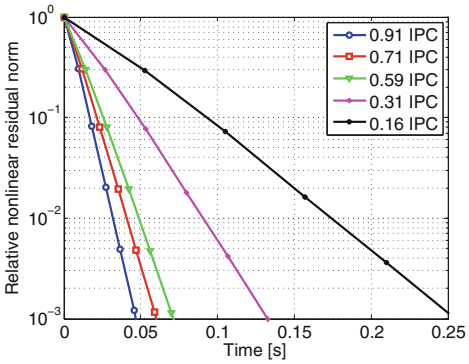

Fig. 3. Relationship between runtime and nonlinear residual norm for different amounts of parallelism, for the L3D problem.

5 Conclusions

A GPU implementation of an asynchronous iterative algorithm for computing incomplete factorizations has been presented. Significant speedups over the level scheduling implementation of the cuSPARSE library were reported. Several techniques were discussed for controlling the order of the asynchronous computations to enhance convergence and data locality. These techniques may be applied in general to other asynchronous iterative algorithms.

Acknowledgments. This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC-0012538 and DE-SC-0010042. Support from NVIDIA is also acknowledged.

References

1. Anzt, H., Tomov, S., Dongarra, J., Heuveline, V.: A block-asynchronous relaxation method for graphics processing units. J. Parallel Distrib. Comput. **73**(12), 1613–1626 (2013)
2. Benzi, M., Joubert, W., Mateescu, G.: Numerical experiments with parallel orderings for ILU preconditioners. Electron. Trans. Numer. Anal. **8**, 88–114 (1999)
3. Bergman, K. et al.: ExaScale computing study: technology challenges in achieving exascale systems peter kogge, editor & study lead (2008)
4. Chow, E., Patel, A.: Fine-grained parallel incomplete LU factorization. SIAM J. Sci. Comput. **37**, C169–C193 (2015)
5. Contassot-Vivier, S., Jost, T., Vialle, S.: Impact of asynchronism on gpu accelerated parallel iterative computations. In: Jónasson, K. (ed.) PARA 2010, Part I. LNCS, vol. 7133, pp. 43–53. Springer, Heidelberg (2012)
6. Davis, T.A.: The University of Florida Sparse Matrix Collection. NA DIGEST 92 (1994). http://www.netlib.org/na-digesthtml/

7. Doi, S.: On parallelism and convergence of incomplete LU factorizations. Appl. Numer. Math. **7**(5), 417–436 (1991)
8. Duff, I.S., Meurant, G.A.: The effect of ordering on preconditioned conjugate gradients. BIT **29**(4), 635–657 (1989)
9. Frommer, A., Szyld, D.B.: On asynchronous iterations. J. Comput. Appl. Math. **123**, 201–216 (2000)
10. Innovative Computing Lab: Software distribution of MAGMA, July 2015. http://icl.cs.utk.edu/magma/
11. Lukarski, D.: Parallel Sparse Linear Algebra for Multi-core and Many-core Platforms - Parallel Solvers and Preconditioners. Ph.D. thesis, Karlsruhe Institute of Technology (KIT), Germany (2012)
12. Naumov, M.: Parallel incomplete-LU and Cholesky factorization in the preconditioned iterative methods on the GPU. Technical report. NVR-2012-003, NVIDIA (2012)
13. NVIDIA Corporation: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Whitepaper (2012)
14. NVIDIA Corporation: CUSPARSE LIBRARY, July 2013
15. NVIDIA Corporation: NVIDIA CUDA TOOLKIT V6.0, July 2013
16. Poole, E.L., Ortega, J.M.: Multicolor ICCG methods for vector computers. SIAM J. Numer. Anal. **24**, 1394–1417 (1987)
17. Saad, Y.: Iterative Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, Philadelphia (2003)
18. Venkatasubramanian, S., Vuduc, R.W.: Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In: Proceedings of the 23rd International Conference on Supercomputing, ICS 2009, pp. 244–255. ACM, New York (2009)
19. Volkov, V.: Better performance at lower occupancy. In: GPU Technology Conference (2010)