A Sparse Symmetric Indefinite Direct Solver for GPU Architectures

JONATHAN D. HOGG, EVGUENI OVTCHINNIKOV, and JENNIFER A. SCOTT,
STFC Rutherford Appleton Laboratory

In recent years, there has been considerable interest in the potential for graphics processing units (GPUs) to speed up the performance of sparse direct linear solvers. Efforts have focused on symmetric positive-definite systems for which no pivoting is required, while little progress has been reported for the much harder indefinite case. We address this challenge by designing and developing a sparse symmetric indefinite solver SSIDS. This new library-quality LDL^T factorization is designed for use on GPU architectures and incorporates threshold partial pivoting within a multifrontal approach. Both the factorize and the solve phases are performed using the GPU. Another important feature is that the solver produces bit-compatible results. Numerical results for indefinite problems arising from a range of practical applications demonstrate that, for large problems, SSIDS achieves performance improvements of up to a factor of 4.6× compared with a state-of-the-art multifrontal solver on a multicore CPU.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Parallel algorithms*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Sparse, structured and very large systems (direct and iterative methods)*; G.4 [**Mathematical Software**]: *Algorithm design and analysis, Parallel and vector implementations*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Bit compatibility, GPU, indefinite symmetric systems, LDL^T factorization, multifrontal direct solver, sparse linear systems

ACM Reference Format:
Jonathan D. Hogg, Evgueni Ovtchinnikov, and Jennifer A. Scott. 2016. A sparse symmetric indefinite direct solver for GPU architectures. ACM Trans. Math. Softw. 42, 1, Article 1 (January 2016), 25 pages.
DOI: http://dx.doi.org/10.1145/2756548

1. INTRODUCTION

The solution of large sparse linear systems of equations, $Ax = b$, lies at the heart of many scientific and engineering problems. Solving such systems frequently represents a computational bottleneck, and this has resulted in significant effort being invested over the past 50 years in the development of both iterative and direct solution methods. While the former have the advantage of requiring limited memory and are able to achieve good performance on modern machines through the efficient implementation of matrix-vector products, the latter are popular because of their robustness, allowing them to be frequently used as "black-box" codes within industrial applications. Over the years, much effort has gone into exploiting novel computing architectures to improve

This work was supported by the gift of the C2050 card from NVIDIA, and EPSRC funding through the CCP ASEArch project under grant EP/J010553/1.
Authors' addresses: J. D. Hogg, E. Ovtchinnikov, and J. A. Scott, Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Oxford, Didcot, Oxfordshire, OX11 0QX, United Kingdom; emails: {e.ovtchinnikov, jennfier.scott}@stfc.ac.uk. Correspondence to: jonathan.hogg@stfc.ac.uk.

DOI: http://dx.doi.org/10.1145/2756548

the performance of sparse direct algorithms. In this article, we seek to accelerate the factorization and subsequent solution of large sparse indefinite matrices using graphics processing units (GPUs).

Modern GPUs offer a significant advantage over mass-market CPUs in terms of both performance per dollar and performance per watt. Comparing systems utilizing similar amounts of power, GPUs can offer roughly $5\times$ more floating-point performance and memory bandwidth. They also provide a test bed for the alternative programming models likely to be required as the core count in future CPUs increases. Current GPUs can be modeled as vector machines, with each set of cores sharing a common program counter and other resources.

In this article, we consider only NVIDIA GPUs, as these are by far the most predominant ones used for high-performance computing at present. Resources on such a GPU are arranged into a number of pools called streaming multiprocessors (SMs). Each SM has up to 192 floating-point cores that share (to varying degrees) hardware resources. All cores in an SM are connected to the same pool of fast cache, which also doubles as a shared memory area, allowing efficient exchange of information between them. The hardware supports running multiple threads per core (up to 32) and uses fast context switching to hide both instruction and memory latency.

Symmetric indefinite sparse linear systems arise in a wide variety of important technical and scientific applications. These include mixed finite-element methods in engineering fields such as fluid and solid mechanics, and interior point algorithms in both linear and nonlinear optimization. Given a sparse symmetric indefinite matrix A, an LDL^T factorization algorithm computes a unit lower triangular matrix L and a block diagonal matrix D, with 1×1 and 2×2 blocks on the diagonal, such that $A = LDL^T$. The solution process is completed by solving $Ly = b$ for y, then $Dz = y$ for z, and, finally, $L^T x = z$ for x. In practice, the system matrix is prescaled to help with numerical stability and preordered to minimize the amount of fill in the L factor, so that $\bar{A} = PSASP^T$ is factorized, where S is a diagonal scaling matrix and P is a permutation matrix. For simplicity of notation, throughout this article, we work only with A but employ scaling and ordering when performing numerical experiments.

The multifrontal method [Duff et al. 1989; Duff and Reid 1983] is often used to perform an indefinite factorization. This method is built upon the decomposition of the factorization into a large number of small dense submatrices (frontal matrices) that are able to exploit high-level BLAS operations, leading to high flop rates and efficient performance. The computational workflow is structured as a tree, with a frontal matrix at each node of the tree. The edges represent data movement in which the results from a child node are assembled into the frontal matrix of its parent. As each child of a parent can be computed independently, parallelism can be achieved through both coarse task parallelism across the tree and fine-grained parallelism within the linear algebra at each node. Modern sparse solvers that run on CPUs and use this parallel multifrontal approach include MUMPS [MUMPS 2013], WSMP [Gupta and Avron 2013], and HSL_MA97 [Hogg and Scott 2011, 2013; HSL 2013]. A number of recent papers and reports have considered accelerating the multifrontal algorithm using a GPU [George et al. 2011; Kim and Eijkhout 2013; Krawezik and Poole 2010; Lacoste et al. 2013; Lucas et al. 2012; Yu et al. 2011]. In the symmetric case, these have concentrated on positive-definite systems that do not require pivoting for stability; little attention has been paid to the problem of efficiently incorporating robust pivoting strategies that are needed for the numerical stability of indefinite systems. Furthermore, the proposed implementations port only the computationally intensive part of the algorithm to the GPU; that is, they aim to run the factorization of the largest frontal matrices that are close to the root of the tree on the GPU, while the rest of the factorization (and the solve phase that uses the computed factors) is performed on the host CPU, thus significantly increasing the data movement involved and limiting the overall gains achieved by employing GPUs. Thus, the main contributions of our work are as follows:

—The design and development of an efficient pivoting strategy within a sparse indefinite multifrontal code for running on GPU architectures. We employ threshold partial pivoting: the novelty of our approach lies in the organization of the computation to exploit the GPU. Our data structures are not static since we have to accommodate pivots that are delayed (i.e., pivots that fail the threshold stability test and have to be delayed until later in the factorization when, after further update operations, they can be used stably). This not only increases the complexity of the coding but also can adversely affect performance, so we employ scaling and ordering strategies to minimize delayed pivots [Hogg and Scott 2014].

—An implementation that allows all the frontal matrices, including the small ones at the leaf nodes of the tree, to be efficiently factorized on the GPU.

—A solve phase that is implemented on the GPU. As far as we are aware, this issue has not been discussed before as all attention has been on the factorization phase that, traditionally, has been the most time-consuming part of the solution process. However, if the factorization is speeded up sufficiently and if multiple solves in sequence are required following the factorization, a serial solve phase can become a bottleneck. The solve phase is memory bound (see Hogg and Scott [2010] for a discussion); implementing it on a GPU allows exploitation of the increased memory bandwidth.

—A multifrontal code that produces bit-compatible (reproducible) results in the sense that two runs of the solver on the same machine with identical input data produces identical output. Not only is this an important aid in debugging and correctness checking, but also some industries (e.g., nuclear, finance) require reproducible results to satisfy regulatory requirements.

Most of the past studies on multifrontal solvers on GPUs have produced software prototypes, rather than software that has been made generally available. By contrast, our new solver, which is called `SSIDS` (**S**parse **S**ymmetric **I**ndefinite **D**irect **S**olver), is an open-source package and is part of the SPRAL library; it can be downloaded from http://www.numerical.rl.ac.uk/spral. The code is written in a mixture of Fortran 2003 and CUDA, using double-precision arithmetic throughout. It is fully documented, tested, and maintained by the Numerical Analysis Group at the STFC Rutherford Appleton Laboratory.

The remainder of this article is organized as follows. Section 2 provides a brief introduction to the multifrontal method for indefinite systems. In Section 3, we discuss the design of `SSIDS` and, in particular, we discuss the assembly of the child nodes into the parent, the stable partial factorization of the frontal matrices, the subsequent update operations, and the implementation of the solve phase. In Section 4, we present numerical results for a range of indefinite problems arising from practical applications. The performance of our GPU-accelerated LDL^T factorization and solver is compared with that of our state-of-the-art OpenMP multifrontal code `HSL_MA97`. Results are given for both the factorize and solve phases of `SSIDS`; both demonstrate performance improvements of up to 4.6× compared to `HSL_MA97`. Finally, we summarize our findings in Section 5.

2. OVERVIEW OF THE MULTIFRONTAL METHOD

In common with other sparse direct methods, the multifrontal method comprises the following phases:

—An ordering phase that exploits the sparsity (nonzero) structure of A to determine a pivot sequence (i.e., the order in which the Gaussian eliminations will be performed). The choice of pivot sequence significantly influences the memory requirements of the factorization, the fill in the factor L, and the number of floating-point operations required to carry out the factorization.
—An analyze phase that uses the pivot sequence and sparsity pattern of A to establish the workflow and data structures for the factorization.
—A factorization phase that optionally scales the matrix before performing the numerical factorization. Following the analyze phase, more than one matrix with the same sparsity pattern may be factorized.
—A solve phase that performs forward elimination followed by back substitution using the stored factors. Repeated calls to the solve phase with one or more right-hand sides may follow the factorization phase. This is typically used to implement iterative refinement (see, e.g., Golub and van Loan [1996]) to improve the accuracy of the computed solution.

2.1. Selecting an Ordering

Considerable effort in the last 30 years has concentrated on the development of algorithms that generate good pivot orders. These include methods based on the minimum degree algorithm [Amestoy et al. 1996, 2004; Liu 1985; Tinney and Walker 1967] and on nested dissection [George 1973]. The former perform well on many small and medium-sized problems and on highly sparse matrices, while the latter has been found to give better orderings for very large problems, particularly those from three-dimensional discretizations. Furthermore, a parallel solver is generally better able to exploit parallelism if a nested dissection ordering is used. Many direct solvers offer a choice of orderings, including either their own implementation of nested dissection or an explicit interface to an established nested-dissection library such as the METIS graph partitioning package [Karypis and Kumar 1998, 1999].

2.2. The Analyze Phase

The analyze phase is designed to construct the sparsity pattern of the factor L. It does this by building an *elimination tree*. This graph describes the structure of L in terms of the data dependence between pivotal columns. It permits permutations of the pivot order that do not affect the number of entries in L to be identified and allows fast algorithms to be used in determining the exact structure of L.

A *supernode* is a set of contiguous columns of L with the same sparsity structure below a dense (or nearly dense) triangular submatrix. This trapezoidal matrix has zero rows corresponding to variables that are eliminated later in the pivot sequence at supernodes that are not ancestors in the elimination tree. This matrix can be compressed by holding only the nonzero rows, each with an index held in an integer. The condensed version of the elimination tree consisting of supernodes is referred to as the *assembly tree*. Supernodes can be exploited to facilitate the use of efficient dense linear algebra kernels and, in particular, BLAS kernels. These can offer such a large performance increase that it is often advantageous to amalgamate supernodes that have similar (but not exactly the same) nonzero patterns, despite this increasing the fill in L and the operation count. For convenience, throughout the remainder of this article, we use the term "node" when referring to a supernode.

2.3. The Factorize Phase

The factorization of A proceeds using a succession of assembly operations into small dense frontal matrices, interleaved with the partial factorizations of these matrices. The basic outline of the algorithm is as follows:

```
for each node of assembly tree working from leaves to root do
    Assemble frontal matrix $\mathcal{F}$ from child nodes' contribution blocks and original matrix $A$.
    Factorize fully summed part $\begin{pmatrix} \mathcal{F}_1 \\ \mathcal{F}_2 \end{pmatrix}$. Store in factors.
    Update contribution block $\mathcal{F}_S$. Store on temporary stack for use by parent node.
end for
```

The dense frontal matrix consists of those rows of the associated node that are nonzero and the matching set of columns. The aim is to ensure that all columns of the node are *fully summed*, that is, that all the contributions from A and from any child nodes have been summed (assembled). Those columns of the frontal matrix that do not belong to the node (and are there from symmetry with the nonzero rows) are said to be *nonfully summed,* and the assembly aims to sum any contributions from child nodes therein. Once the assembly is complete, a partial factorization of the frontal matrix is performed (i.e., pivots are chosen from the fully summed columns and then eliminations performed). The computed columns of L and D are not needed again until the solve phase (see later) and so can be stored, while the rest of the frontal matrix corresponding to the non-fully summed rows and columns (termed the *generated element* or *contribution block*), together with a list of the variables involved, is stored separately.

At each node of the assembly tree, the $m \times m$ frontal matrix can be expressed in the form

$$\mathcal{F} = \begin{pmatrix} \mathcal{F}_1 & \mathcal{F}_2^T \\ \mathcal{F}_2 & \mathcal{F}_S \end{pmatrix}, \tag{1}$$

where $\mathcal{F}_1$ and $\mathcal{F}_2$ are *fully summed,* while $\mathcal{F}_S$ is the partially summed block. Pivots are chosen from $\mathcal{F}_1$. If $\mathcal{F}_1$ has order $p \leq m$ and $q \leq p$ pivots are chosen, the partial factorization of $\mathcal{F}$ takes the form

$$\mathcal{F} = P \begin{pmatrix} L_1 & 0 \\ L_2 & I \end{pmatrix} \begin{pmatrix} D_1 & 0 \\ 0 & C \end{pmatrix} \begin{pmatrix} L_1^T & L_2^T \\ 0 & I \end{pmatrix} P^T, \tag{2}$$

where D_1 is a block diagonal matrix of order q, P is a permutation matrix, and C is the generated element.

For symmetric positive-definite matrices, the pivot order chosen before the factorization commences can be used without modification. Moreover, the data structures determined by the analyze phase can be static throughout the factorization. For indefinite problems, using the supplied pivot order may be unstable or impossible because of (near) zero diagonal entries and, in general, it must be modified to maintain numerical stability. If symmetry is not to be destroyed, both 1×1 and 2×2 pivots are needed. Ashcraft et al. [1999] showed that bounding the size of the entries of L, together with a backward stable scheme for solving 2×2 linear systems, suffices to give backward stability for the entire solution process. They found that the widely used strategy of Bunch and Kaufman [1977] does not have this property, whereas the threshold pivoting technique first used by Duff and Reid [1983] in their original multifrontal solver does.

Duff and Reid choose the pivots one by one, with the aim of limiting the size of the entries l_{ij} in L:

$$|l_{ij}| < u^{-1}, \tag{3}$$

where the threshold u lies in the range $0 \leq u \leq 1.0$. At a given stage of the factorization with the frontal matrix $\mathcal{F}$ given by Equation (1), let k denote the number of rows and columns of the block diagonal matrix D_1 found so far from $\mathcal{F}$ and let f_{ij}, with $i > k$ and $j > k$, denote an entry of $\mathcal{F}$ after it has been updated by all the permutations and pivot operations so far. For a 1×1 pivot in column $j = k+1$, the requirement for the inequality in Equation (3) corresponds to the stability threshold test

$$|f_{jj}| > u \max_{i>j} |f_{ij}|. \tag{4}$$

In Duff et al. [1991], the corresponding stability test for a 2×2 pivot involving columns $i = k+1$ and $j = k+2$ is

$$\left| \begin{pmatrix} f_{ii} & f_{ij} \\ f_{ij} & f_{jj} \end{pmatrix}^{-1} \right| \begin{pmatrix} \max_{i>j} |f_{ii}| \\ \max_{i>j} |f_{ij}| \end{pmatrix} < \begin{pmatrix} u^{-1} \\ u^{-1} \end{pmatrix}, \tag{5}$$

where the absolute value notation for a matrix refers to the matrix of corresponding absolute values. The choice of u controls the balance between stability and sparsity: in general, the smaller u is, the sparser L will be, while the larger u is, the more stable the factorization will be. Typically, a value of 0.01 is recommended.

In addition to bounding the size of entries in L, the ability to stably apply the inverse of D to a vector is required. This is trivially the case for 1×1 pivots, but for 2×2 pivots we must check that:

(1) the determinant $|f_{ii} f_{jj} - f_{ij} f_{ij}|$ is sufficiently large; and
(2) cancellation does not occur during the application of the inverse.

The indefinite factorization package HSL_MA64 [Reid and Scott 2011] uses the following test to check these conditions:

$$|f_{ii} f_{jj} - f_{ij} f_{ij}| \geq \max\left(10^{-20} f_{ii}, 10^{-20} f_{ij}, 10^{-20} f_{jj}, \frac{f_{ii} f_{jj}}{2}, \frac{f_{ij} f_{ij}}{2}\right). \tag{6}$$

An alternative pivoting test is used by Kim and Eijkhout [2012] that explicitly checks the condition in Equation (3). This is slightly weaker than the test in Equation (5) but is useful because it does not require that the largest off-diagonal entries in the candidate pivot columns are found before the pivot is applied. Observe, however, that the main drawback of the Kim and Eijkhout approach is that if Equation (3) is not satisfied, some mechanism must be in place to undo the pivot application. We use this *a posteriori pivoting* in our GPU solver; this is discussed further in Section 3.2.

If some of the pivot candidates at a node fail the stability test, they are passed as part of the generated element up the assembly tree to the parent. This means that the data structure at the parent must be modified to accommodate the extra rows and columns. If there are a large number of delayed pivots, this can have serious consequences in terms of time as well as the memory and flops required for the factorization. Thus, it is important to choose an ordering and scaling that aims to limit the number of delayed pivots [Hogg and Scott 2014].

2.4. The Solve Phase

The solve phase iterates over the assembly tree, first solving $Ly = b$ (forward) by iterating from the leaf nodes toward the root, then solving backward $DL^T x = y$ (backward) by iterating from the root to the leaf nodes. In the forward solve, at each node a triangular solve is performed with the diagonal block of the factors, followed by a matrix-vector multiplication with the remainder. In the backward solve, the order is reversed and transpose operations applied. Multiple right-hand sides can be solved simultaneously by replacing the vectors x, y, b with matrices X, Y, Z, transforming the memory-bound matrix-vector operations with more efficient compute-bound matrix-matrix ones.

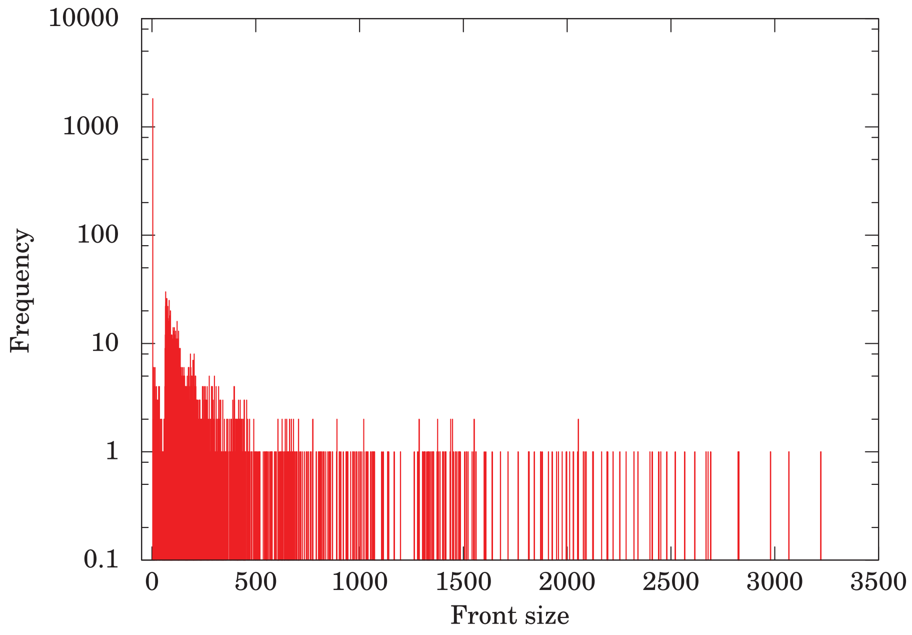


Fig. 1. Plot of the front size distribution for matrix GHS_indef/ncvxqp3. The order of the matrix is 75,000.

2.5. Tree and Node Parallelism

Existing works on parallelism in the multifrontal case exploit two distinct sources of parallelism [Davis 2006]. Tree parallelism exploits the independent nature of different tree branches, which can be worked on simultaneously. Node parallelism refers to exploiting parallelism within the dense linear algebra operation at each node.

Typical problems exhibit a large number of small leaf nodes, reducing to a small number of large nodes near the root of the tree. This leads to a strong synergy between these two types of parallelism. This is illustrated in Figure 1 for the sparse symmetric indefinite matrix GHS_indef/ncvxqp3[1]. There is little benefit in exploiting node parallelism on the small leaf nodes, but lots of tree parallelism. Near the root there is limited scope for tree parallelism, but the larger nodes provide greater benefits from exploiting node parallelism.

2.6. Bit Compatibility

For sequential solvers, achieving bit compatibility (in the sense that two runs on the same machine using the same binary and identical input data should produce identical output) is not a problem. But enforcing bit compatibility can limit dynamic parallelism, and when designing a parallel sparse direct solver, the goal of efficiency potentially conflicts with that of bit compatibility.

Bit compatibility is essential for some users due to regulatory requirements (e.g., financial industries, engineering) or to build trust in their software from nontechnical users (e.g., climate modeling). For others, it is just a desirable feature for debugging purposes. Often LDL^T factorizations are used at the core of much more complicated codes that typically feature heuristics that can be sensitive to very small changes in the linear solutions found. For example, in optimization using nonlinear interior point

[1]All the sparse matrices used in this article are taken from the University of Florida Sparse Matrix Collection [Davis and Hu 2011] and are listed in Table III.

methods, bit incompatibility can sometimes result in completely different maxima being found.

The critical issue is the way in which N numbers (or, more generally, matrices) are assembled:

$$sum = \sum_{j=1}^{N} S_j,$$

where the S_j are computed using one or more processors. The assembly is commutative but, because of the potential rounding of the intermediate results, is not associative so that the result *sum* depends on the order in which the S_j are assembled. A straightforward approach to achieving bit compatibility is to enforce a defined order on each assembly operation, independent of the number of processors. As discussed in Hogg and Scott [2012], this is a viable approach for multifrontal codes and is used by the multicore multifrontal solver HSL_MA97; the same approach is followed by SSIDS (but the two codes are not bit compatible with respect to each other).

3. DESIGN OF SSIDS

Our new GPU sparse indefinite solver SSIDS implements a supernodal multifrontal algorithm, as outlined previously. In this section, we look at some details that are designed to optimize performance of the code on a GPU.

Achieving high performance on the GPU requires that sufficient threads are executing to either saturate memory bandwidth (in the case of the memory-bound assembly operations and the entire solve phase) or floating-point capacity (in the case of compute-bound computations during the factorization). The CUDA API provides two ways of launching a significant number of threads. The most straightforward is to launch a large number of threads all running the same kernel code. The second is to run multiple different kernels in parallel. The mechanism for doing so is to create different *streams* of work. The CUDA API guarantees that all kernels launched within the same stream execute in serial (i.e., one finishes before the next begins) but allows kernels from different streams to run in parallel. However, at most 16 streams (fewer on older devices) can issue instructions at one time.

Toward the leaves of the assembly tree, the nodes are very typically small, with the larger nodes at or near the root(s) of the tree. Given the large number of these very small nodes, just using streams will be insufficient to ensure that the majority of the GPU is kept busy (the kernel launch overhead far outweighs the execution time of the kernel for small nodes). Instead, we manually handle the parallelism through the use of an array that describes the parameters of each front and the operations that need to be performed upon it. A kernel with a sufficient number of CUDA thread blocks to cover the available work is launched, and the first action of each thread block is to extract the parameters of its associated task from this data structure. This mechanism allows us to fully exploit both tree- and node-level parallelism.

It would be attractive to have a single kernel that was able to perform any of the tasks required for the factorization using an underlying DAG for scheduling. This would facilitate good algorithmic load balance and minimize the occasions on which the GPU is waiting for work to be released for execution. However, doing this would have two drawbacks: first, the resource requirements of such a multikernel would be determined by the worst subkernel it contained, severely limiting physical thread occupancy; second, the code would be so complex that it would overflow the small number of registers available on older Fermi cards, leading to register spill and associated slowdown. For these reasons, each kernel we discuss is implemented and launched independently.

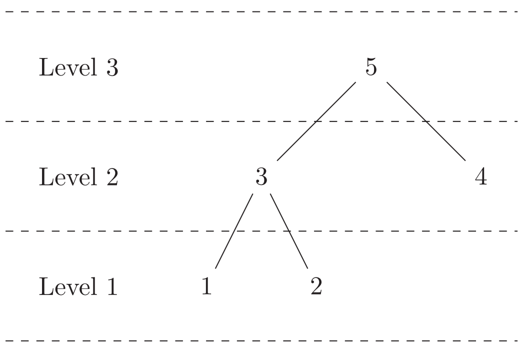


Fig. 2. Allocation of nodes to levels.

Using our manual approach, synchronization is (normally) achieved through the launch of kernels within the same stream. Independent subtrees can be scheduled using different streams; however, exploiting this in our tests usually led to a slowdown. Where a dependency exists between two tasks, this is enforced by allocating the later operation to a future kernel launch. However, in a limited number of cases, we implement synchronization through GPU global memory and spin-locks (see, e.g., the assembly operations described later).

To simplify task generation and memory management, task lists are generated through a level-based approach. Nodes of the assembly tree are allocated to a level depending on their distance from the root (this is illustrated in Figure 2 where the root node is 5). As the generated elements from the child nodes can be discarded once the frontal matrix at the parent node has been assembled, this can be done using only two arrays: one for the even-numbered levels and one for the odd. At each level, one of these arrays holds the generated elements from the child nodes, while the other holds those generated at this level, swapping roles at the next level.

Beyond the strict distance-based approach we use, it is possible to move some nodes between levels as long as dependencies are satisfied. For example, in Figure 2, node 4 could be assigned either to level 1 or level 2. This can be exploited to improve load balance; however, the best allocation is a discrete optimization problem, the solution of which is beyond the scope of this article.

We split the work to be performed at each node into three distinct operations: assembly into the frontal matrix $\mathcal{F}$, dense factorization of the fully summed columns of $\mathcal{F}$, and formation of the generated element. Each of these operations is itself composed of a number of tasks. Thus, in memory, we have per-node data structures reflecting the current representation of the node and associated mutable data, and per-task data structures reflecting operations to be performed and on what data. In the following sections, we explain how the operations for a single node are handled, although, as described earlier, all the nodes that belong to the same level of the assembly tree are treated simultaneously. To avoid confusion, we refer to *blocks* of CUDA threads that specify a unit of work that is allocated to an SM, and *tiles* of a matrix (often referred to as blocks in the literature on sparse direct solvers).

We note that the CUBLAS only support the simultaneous execution of multiple small operations of different sizes through the use of multiple streams. However, tests showed that using the specialized kernels we developed is faster and more straightforward, particularly on the older Fermi hardware. CUBLAS are, however, used when only a single node is being operated on (e.g., at the root).

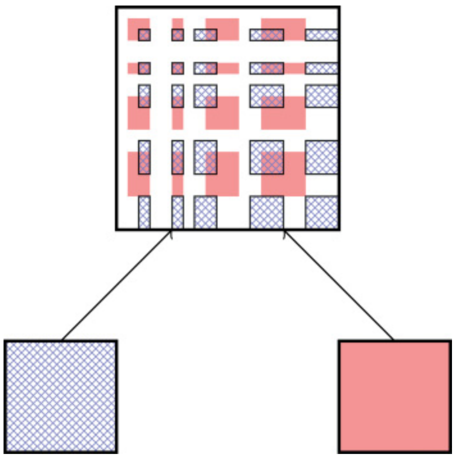


Fig. 3. Sparse assembly of the contribution blocks of two child nodes into the parent frontal matrix.

3.1. Sparse Assembly

At each node, entries from A, generated elements from child nodes, and any delayed pivots must be assembled. Separate kernels handle each of these.

3.1.1. Entries from A. During the analysis phase, a mapping array from positions in A to positions in L is constructed. This makes the assembly of A into the frontal matrix very straightforward, with the only complication being that, if there are delayed pivots inherited from its child nodes, the dimension of the front is greater than predicted in the analyze phase. In this case, the mapping array is used to calculate the row and column for the insertion, rather than providing the insert location directly.

3.1.2. Child Nodes. Figure 3 shows the typical situation for assembly (though the number of child nodes may vary). Entries from each child node's contribution block must be scattered in a sparse fashion into the parent frontal matrix. Some (but not all) entries of this parent matrix have contributions from multiple children. To ensure correctness, either some form of explicit synchronization is required or atomic operations must be used. There are two obvious designs for the assembly: one thread per entry of the parent (allowing ordering and synchronization to be inherent but complex addressing) or one thread per entry of each child node (requiring external synchronization but reducing memory traffic to the child).

The thread-per-parent-entry approach requires large data structures as we must either store a list of source pointers into the children (even if there is no contribution to a given entry) or spend considerable time calculating this list (a very nontrivial task if parallel efficiency is desired).

The thread-per-child-entry approach has the advantage that every thread is guaranteed to perform useful work, but the disadvantage that it requires some external synchronization to avoid write conflicts on entries shared by multiple children. Further, if bit compatibility is required, an assembly ordering must be enforced. Past experience [Hogg and Scott 2011] has shown that for the multifrontal algorithm, it is possible to ensure bit compatibility at relatively low cost. This is likely to be equally true on GPUs that have a limited number of atomic operation units available.

Table I explores the relative cost of the two approaches on a subset of the test matrices used in Section 4. The thread-per-child-entry approach potentially uses more bandwidth than the thread-per-parent-entry approach, as it may access the same

Table I. A Summary of the Number of Entries of the Parent Frontal Matrices That Have Contributions from 1 or More Child Nodes

Problem	Total Entries	$nchild = 0$ %	$nchild = 1$ %	$nchild > 1$ %	Potential False Dependencies %
2. Andrianov/mip1	1.65×10^8	11.2	88.8	1.0	1.1
7. GHS_indef/ncvxqp3	4.42×10^8	12.0	88.0	2.0	55.4
8. Oberwolfach/gas_sensor	1.61×10^8	29.3	70.7	4.3	51.8
14. Rothberg/cfd2	2.20×10^8	33.7	66.3	4.8	58.8
15. DNVS/thread	1.47×10^8	32.0	68.0	5.3	56.8
22. AMD/G3_circuit	7.18×10^8	41.3	58.7	3.3	46.2
23. GHS_psdef/bmwcra_1	4.29×10^8	32.4	67.6	5.3	52.8
29. ND/nd6k	1.19×10^9	6.6	93.4	2.0	12.6
30. Schenk_IBMNA/c-big	1.96×10^9	8.8	91.2	1.4	84.9
32. PARSEC/Si5H12	2.21×10^9	5.6	94.4	2.0	9.6

Statistics in columns 3 to 6 are given as a percentage of the total number of entries summed over all the parent frontal matrices reported in column 2. $nchild = 0$ ($nchild = 1$, respectively) is the percentage of the entries that 0 children (1 child, respectively) contribute to. Potential false dependencies is the percentage of entries that $0 < nchild < nc$ child nodes contribute to, where nc is the number of children of the parent.

parent entries multiple times. However, the table shows that this is relatively rare, typically occurring for less than 5% of entries; indeed, it may in fact use less bandwidth than the thread-per-parent-entry approach once the sparsity data structures are taken into account. The $nchild = 0$ column shows the number of excess threads that are launched but have no work to perform under the thread-per-parent-entry approach. One might imagine launching threads only for entries with $nchild \neq 0$, but this would further complicate data structures (the destination must now be stored) in addition to reducing the coherence of memory accesses. Given this analysis, we feel that the thread-per-parent-entry approach is weaker than the thread-per-child-entry approach (which is also simpler to implement).

Our implementation thus uses the thread-per-child-entry approach. Each generated element from a child is divided into a number of tiles, and a thread block is launched to assemble each tile into the parent frontal matrix using a simple mapping array to determine the destination row and column of each entry. The mapping array used is calculated during the analysis phase and takes the same amount of storage as a traditional row list would.

As already observed, to preserve bit compatibility, we must ensure that the child entries are always assembled in the same order. This could be achieved by using a separate kernel launch to handle each child; however, doing so reduces parallel efficiency and introduces additional kernel launch overheads. Instead, we use global memory to signal once a child has completed so the next child may begin (each block atomically increments a per-parent counter to indicate when it has finished).

This approach of using node-level synchronization can lead to inefficiencies due to false dependencies (i.e., unneeded synchronizations) on an entrywise basis. To examine this, Table I also includes a column labeled “potential false dependencies.” Suppose a node has nc child nodes. A *potential false dependency* is said to occur when the number of child nodes that contribute to a given entry of the parent frontal matrix is at least one but is fewer than nc (in Figure 3, these are the entries that come from one child node but not the other). We see that these potential false dependencies represent a significant proportion of entries (often more than half). However, these figures are an overestimate because of the way they are calculated. Under any given assembly order, some of the false dependencies will not occur as the child nodes that do contribute are ordered correctly (i.e., in Figure 3, those entries belonging to the left node only are not actual false dependencies if the left node is ordered first). Further, trying to avoid false

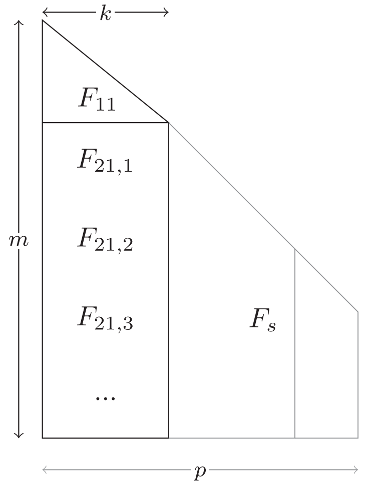


Fig. 4. $m \times k$ tile column within the fully summed part of a frontal matrix. Here, F_s denotes the fully summed columns lying to the right of the tile column.

dependencies is likely to offer only limited speedup as warps only run at the speed of their slowest member.

3.1.3. Delayed Pivots. A final kernel handles the assembly of entries corresponding to delayed pivots. Assuming the problem has been well scaled and well ordered, we expect relatively few delays (or at least relatively few children to contribute them) [Hogg and Scott 2014]. Thus, a single CUDA block is used to handle all delays contributing to a parent node from any of its child nodes. The same lookup arrays used in the assembly of the generated elements are reused to determine the correct insert locations for entries in delayed columns (delayed rows can be assembled without any permutation).

3.2. Dense Factorization of Fully Assembled Columns

We seek to factorize an $m \times p$ submatrix by splitting it into a number of tiles that are each handled by an associated block. Factorization proceeds one column of tiles at a time. We refer to a column of tiles as a *tile column*; it is illustrated in Figure 4 for a tile column composed of k columns ($k < p$). We note that while F_{11} is always square, the off-diagonal tiles of F_{21} are in fact rectangular. Ignoring pivoting, the factorization proceeds as follows:

1. Factor $F_{11} = L_{11} D_1 L_{11}^T$.
2. Apply factor L_{11} to the off-diagonal part of the tile column, $L_{21} = F_{21} L_{11}^{-T}$.
3. Update the remaining tile columns to the right, $F_s \Leftarrow F_s - L_{21} D_1 L_{21}^T$.

Numerical stability requires that we bound entries of L. As discussed in Section 2.3, traditionally this is done by considering the below diagonal entries of largest absolute value in the candidate column(s) (as in Equations (4) and (5)). This requires the whole of the tile column to be kept up-to-date as the factorization of F_{11} progresses. On the GPU, this is suboptimal as communication between the blocks handling each tile is slow. Further, it may not be possible at all as there is no guarantee that all blocks handling the tile column are run simultaneously (for large matrices there may be insufficient resources). Instead, we use a posteriori pivoting in which F_{11} is factored without reference to F_{21}, and the size of the entries in L_{21} are checked after the application of L_{11}. Suppose an entry in column $r \le k$ of L_{21} exceeds the threshold tolerance u^{-1} in absolute value. Columns $r, r+1, \ldots, k$ in the tile column are marked as failed and are restored to their state before application of the rth pivot. This is achieved by storing the factorization in a temporary buffer until it is confirmed as successful, and as this also reduces the complexity of the pivoting permutation, it comes at a low overhead cost. Our implementation uses an `atomicMin()` operation to determine the number of successful columns (we expect this in general to be equal to k). We emphasize that this strategy is to ensure backward stability and is very different from the approach taken by, for example, the PARDISO solver [Schenk and Gärtner 2006], in which F_{11} is factored without reference to F_{21} but **no** check is made subsequently on the size of the factor entries (and thus it offers no guarantee of stability).

Failed candidate pivot columns can be managed with varying degrees of sophistication. One option is to delay them to the parent node, as is done in the CPU code of Kim and Eijkhout [2012]. However, this may lead to significant increases in both the memory usage and the number of flops. Instead, we adopt a simple scheme whereby any column that has failed the pivot test is not retested until it has been updated by at least one other column that has passed the pivot test; once a failed column has been updated, it becomes once more a candidate pivot. Once all candidate columns in the front have been considered for pivoting, any that have failed are delayed to the parent node.

This scheme can break down at the root node(s), as it only allows partners for 2×2 pivots to be selected from a band of width k around the diagonal. In some (rare) cases, this will be insufficient. We cater to these numerically problematic matrices by implementing a special-case kernel that handles any failed pivots that remain at the end of the root node, but, as this will not normally be required, we have not invested significant efforts in its efficiency.

3.2.1. Implementation Details. We split the dense factorization of fully summed columns into three different kernels that are run repeatedly until all the fully summed columns have been either factorized or flagged as delays.

—`factor()`: Each block loads its assigned tile(s) $F_{21,j}$ and the diagonal block F_{11} into shared memory. Each block computes $F_{11} = L_{11}D_1L_{11}^T$ and applies L_{11} to its tiles(s) $L_{21,j} = F_{21,j}L_{11}^{-T}$. The threshold condition is checked for each column of $L_{21,j}$ and the number of successful columns is recorded using `atomicMin()`. L_{21} and $(L_{21}D_1)$ are stored in temporary global memory. The first thread block associated with each tile column also stores L_{11}, D_1, and the pivot sequence used in the factorization of F_{11}. Once this step is complete, the number of successful pivots is known.

—`reorder()`: A row permutation is applied to any computed columns of L in the frontal matrix that are to the left of the current tile column, and a column permutation is applied so that successful columns are moved to become the leading columns of the frontal matrix. Any failed columns are swapped as necessary with successful columns. This leaves the frontal data structure as shown in Figure 5, with the eliminated columns first, then any failed pivots, followed by any fully summed columns in tile columns that have not yet been pivoted on. The non-fully summed part is shown only for illustration purposes and is not touched by this kernel (it will form the generated element). The temporary array that contained L_{21} can now be discarded. The row and column permutations are applied to the temporary array containing $(L_{21}D_1)$.

—`update()`: The partial outer product $L_{21}D_1L_{21}^T$ that updates the remaining fully summed columns F_s is formed by performing a matrix multiplication between L_{21} (stored in the factor array) and $(L_{21}D_1)$ (stored in the temporary memory).

When factorizing F_{11}, we say that a column has been *processed* if it has been selected as a pivot column; otherwise, it is *nonprocessed*. A 2×2 pivot (i, j) is chosen as long as

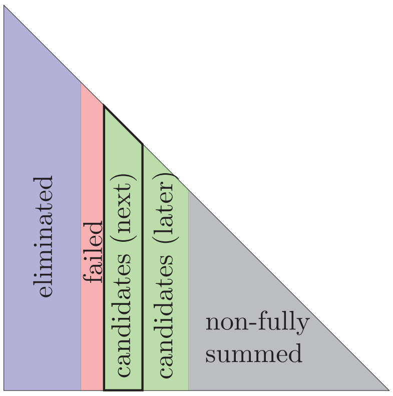


Fig. 5. Frontal matrix data structure after `reorder()`. In order from the left, the columns are (1) those that have been successfully eliminated; (2) those that have failed (but may be retested later); (3) the untested candidate pivots, of which the first k will be tested next in this example and the remainder will be tested later; and (4) the non-fully summed columns that cannot be eliminated at this node.

$$\Theta \frac{|f_{ii} f_{jj} - f_{ij} f_{ij}|}{|f_{ii}| + |f_{jj}| + |f_{ij}|} \geq \max\left(|f_{ii}|, |f_{jj}|\right), \tag{7}$$

where Θ is the 2×2 pivot bias. A bias toward 2×2 pivots is desirable as it prevents the need for a second pivot test and allows updates to be performed more efficiently. Through experiment, we have chosen a value $\Theta = 100$. The observant reader might be curious as to the approximation to Equation (6) that this test represents. We note that the expression $|f_{ii}| + |f_{jj}| + |f_{ij}|$ used as the denominator is within a factor 3 of the correct expression $\max(|f_{ii}|, |f_{jj}|, |f_{ij}|)$, and the error is hence small compared to Θ. If the test in Equation (7) is not met, then a 1×1 pivot is chosen as the larger (in magnitude) of f_{ii} or f_{jj}. The factorization of F_{11} is performed as follows:

```
Initialize the set of processed columns as empty: $\mathcal{P} = \emptyset$.
while there are nonprocessed columns do
  Let $i$ be the leftmost nonprocessed column. Find $j = \max\{|f_{ji}| : j \notin \mathcal{P}\}$ (break ties in favor
  of smallest $j$).
  Check if all entries in column are $<10^{-20}\alpha$, where $\alpha$ is a bound on the maximum entry in
  $F_{11}$ and $F_{22}$ before the start of the partial factorization. If so, record zero pivot.
  Choose $2 \times 2$ pivot $(i, j)$ if stable (test Equation (7)), else choose best $1 \times 1$ of $(i)$ or $(j)$.
  Update $\mathcal{P}$. Perform the eliminations with the selected pivot column(s).
end while
```

We note that the processed columns are not permuted to the leading columns of F_{11}. Instead, the array representing $\mathcal{P}$ is used to mask any operations performed.

3.3. Formation of the Generated Element

The majority of the floating-point operations typically derive from the formation of the generated element $C \Leftarrow C - L_2 D_1 L_2^T$ (recall Equation (2)). We observe that this operation cannot be handled by any one standard BLAS operation: instead, it is implemented using a custom kernel. The matrices L_2 and $(L_2 D_1)$ are available from the factorization of the fully summed columns. C is divided into a number of tiles and a thread block launched for each. The entries of C are stored in registers for the duration of the computation. For a given $k_1 \times k_1$ tile of C, the relevant parts of L_2 and $(L_2 D_1)$ are

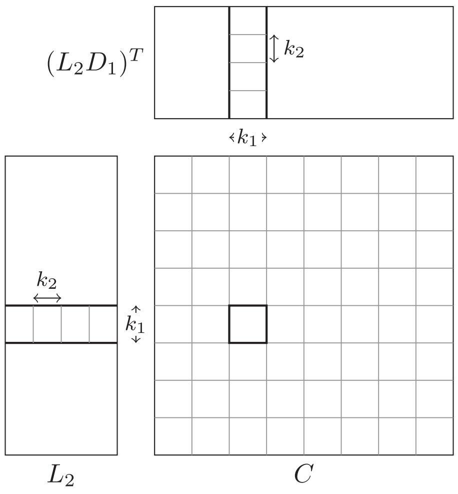


Fig. 6. Calculating the outer product for a tile of the contribution block.

divided into a number of $k_1 \times k_2$ tiles that must be looped over, as shown in Figure 6. Each tile is loaded into shared memory before each thread calculates the portion of matrix-matrix multiplication relevant to the entries of C it is responsible for.

Double buffering is used for performance optimization, allowing the next $k_1 \times k_2$ tiles of L_2 and (L_2D_1) to load while the current values are used for the matrix-matrix multiplication.

3.4. The Positive-Definite Case

In addition to the pivoted kernel for indefinite systems, our code offers an unpivoted Cholesky (LL^T) factorization suitable for positive-definite systems. This is implemented using a drop-in replacement for the dense factorization kernels and a modification to the generated element calculation to allow for the absence of D.

This is provided for convenience but does not fully exploit properties of the Cholesky factorization that allow considerably more scheduling flexibility and avoid the need to cater to the possibility of pivoting permutations. As such, it is unlikely to be as efficient as a dedicated Cholesky solver.

3.5. Implementation of the Solve Phase

The implementation of the solve phase mirrors that of the factorize phase. Tree parallelism is exploited through the allocation of nodes to levels, and all nodes in a level are addressed by the same kernel launch.

Two different variants of the solve phase have been implemented for different circumstances. The first follows the traditional triangular solve methodology. The second adds additional work to the factorize phase where the explicit inverse of the diagonal blocks is calculated (this can be done stably as they are triangular [Higham 1995]). Then, in the solve phase, at each node the dense triangular solve is replaced with a matrix-vector multiply. The latter, which we refer to as the *presolve* approach, is more efficient if the additional factorize cost can be amortized across a number of right-hand sides. This is discussed further in Section 4.4.

4. NUMERICAL EXPERIMENTS

Results are given for the hardware and are summarized in Table II. We note that CPU results are given for a two-socket system, so the performance statistics for the machines

Table II. Hardware Summary

	CPU		GPU	
	E5620	E5-2687W	C2050	K20
	Westmere-EP	Sandy Bridge-EP	Fermi	Kepler
Clock speed	2.4GHz	3.1GHz	1.15GHz	0.7GHz
Cores	4	8	448	2,496
Theoretical DP peak	38.4GFlop/s	199GFlop/s	515GFlop/s	1,170GFlop/s
Achieved `dgemm` peak	34.1GFlop/s	181GFlop/s	298GFlop/s	1,046GFlop/s
Max TDP	80W	150W	238W	225W
Vector length	128 bit	256 bit	n/a	n/a
Memory bandwidth	25.6GB/s	51.2GB/s	144GB/s	208GB/s
GPU memory	n/a	n/a	3GB	5GB
Launch date	Q1 2010	Q1 2012	Q2 2010	Q4 2012
Launch RRP	$391	$1,885	$2,499	$3,199

used are double those stated in the table for a single chip. While we compare CPU and GPU systems from the same eras, we note that this may be somewhat misleading due to the disparity in the systems in terms of both cost and power envelope.

Table III introduces the problems we use for our experiments. The number of entries in L and number of flops are for a factorization without any numerical pivoting using an ordering returned by the nested dissection ordering package METIS v4 [Karypis and Kumar 1998]. All problems are drawn from the UFL Sparse Matrix Collection [Davis and Hu 2011]. In each test, the right-hand side is computed so that the exact solution is the vector of all ones.

Unless otherwise indicated, problems are run with the default settings of `HSL_MA97` (CPU, using hyperthreading) and SSIDS (GPU), except we set the supernode amalgamation parameter `nemin` to 64 for `HSL_MA97` for better performance on our test set; the SSIDS equivalent remains at its default of 32 for best performance. All times are in seconds. Timings on the CPU are taken as the average of 10 runs due to variability (the GPU did not show any measurable variation so a single run was performed).

4.1. Headline Results

Figures 7 and 8 show the time and speed of the factorize phase, respectively, on both CPU and GPU platforms. For these results, all problems are treated as indefinite with a scaling used if it improves the factorization time. In each figure, the first graph shows the performance on the 2010/11 hardware, and the latter on the 2012/13 hardware. Problems 25, 31, and 35 could not be run on the C2050 due to memory constraints and so are omitted where appropriate. The CPU execution times for problem 35 are not clearly shown on the graphs as they are large in comparison with the times for the other examples: they were 21.5s and 6.9s, respectively, on the Westmere-EP and Sandy Bridge-EP platforms. The results show that significant performance benefits are gained through the use of the GPU hardware. The large disparities between the performance behavior of the CPU and GPU codes on particular problems are due to the significantly different parallel designs of the two codes.

The maximum speedup between the Westmere-EP and Fermi hardware was 4.6$\times$ on problem 11, with most problems achieving at least a 2$\times$ speedup. The maximum speedup between the Sandy Bridge-EP and Kepler hardware was 3.1$\times$ on problem 32, with nine problems achieving at least a 2$\times$ speedup.

4.2. Cost of Pivoting

To give an indication of the cost of incorporating pivoting into our multifrontal algorithm, in Table IV, we compare running the positive-definite test matrices using both

Table III. Test Problems: n Is the Dimension of A, $nz(A)$ Is the Number of Nonzero Entries in A, $nz(L)$, and Flops

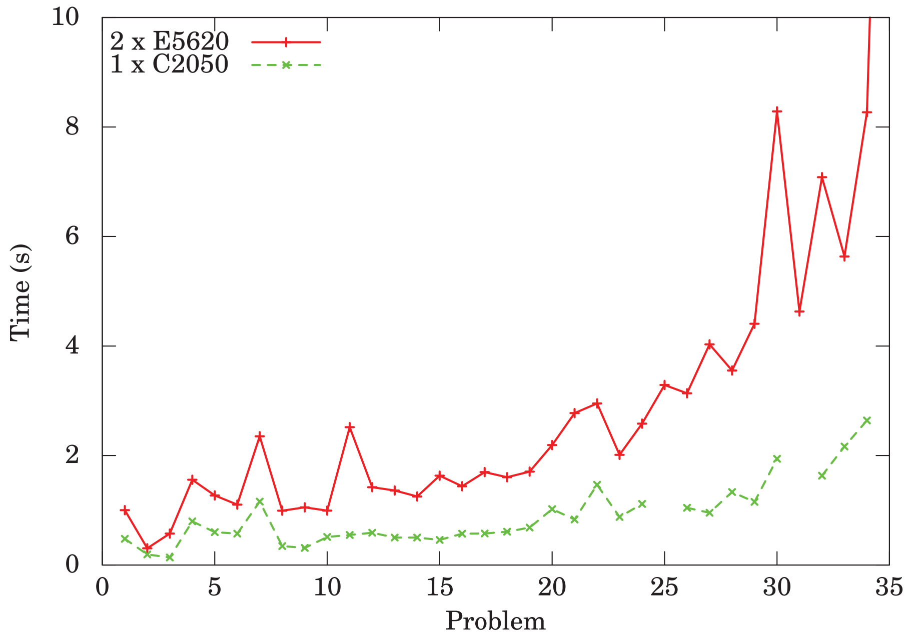
	Problem	n $\times 10^3$	$nz(A)$ $\times 10^6$	$nz(L)$ $\times 10^6$	Flops $\times 10^9$	Description
1.	Newman/astro-ph	16.7	0.121	2.65	2.98	Collaboration network
2.	Andrianov/mip1	66.5	5.21	10.4	5.07	Optimization
3.	PARSEC/SiNa	5.74	0.102	4.89	6.52	Density functional theory
4.	Schmid/thermal2	1230	4.90	51.6	14.6	Unstructured thermal FEM
5.	McRae/ecology1	1000	3.00	37.3	15.2	Electrical network theory
6.	INPRO/msdoor	416.	10.3	52.9	17.6	Structural problem: medium door
7.	GHS_indef/ncvxqp3	75.0	0.275	15.5	18.5	Nonconvex QP problem
8.	Oberwolfach/gas_sensor	66.9	0.885	23.8	21.2	Thermal model single gas sensor device
9.	ND/nd3k	9.00	1.64	12.9	22.1	3D mesh problem
10.	Boeing/pwtk	218.	5.93	48.6	22.4	Pressurized wind tunnel
11.	GHS_indef/c-71	76.6	0.468	13.5	21.7	Nonlinear optimization
12.	BenElechi/BenElechi1	246.	6.70	53.8	26.8	Unknown
13.	GHS_psdef/crankseg_1 †	52.8	5.33	33.4	32.3	Linear static analysis
14.	Rothberg/cfd2 †	123.	1.61	38.3	32.7	CFD pressure matrix
15.	DNVS/thread †	29.7	2.25	24.1	34.9	Threaded connector
16.	DNVS/shipsec1 †	141.	3.98	39.4	38.1	Ship section
17.	DNVS/shipsec8 †	115.	3.38	35.9	38.1	Ship section
18.	Oberwolfach/boneS01 †	127.	3.42	40.2	46.7	Bone micro-FEM
19.	GHS_psdef/crankseg_2 †	63.8	7.11	43.8	46.7	Linear static analysis
20.	Schenk_AFE/af_shell7 †	505.	9.05	93.6	52.2	Sheet metal forming
21.	DNVS/shipsec5 †	180	5.15	53.5	57.3	Ship section
22.	AMD/G3_circuit †	1590	4.62	97.8	57.0	Circuit simulation
23.	GHS_psdef/bmwcra_1 †	149.	5.40	69.8	60.8	Automotive crankshaft
24.	Schenk_AFE/af_0_k101 †	504.	9.03	98.2	60.9	Sheet metal forming
25.	GHS_psdef/ldoor †	952.	23.7	145.	78.3	Structural problem: large door
26.	DNVS/ship_003 †	122.	4.10	60.2	81.0	Ship structure
27.	PARSEC/Si10H16	17.1	0.447	30.6	86.5	Density functional theory
28.	Um/offshore †	260	2.25	84.5	106.	Electromagnetics
29.	ND/nd6k †	18.0	3.46	39.8	111.	3D mesh problem
30.	Schenk_IBMNA/c-big	345.	1.34	40.3	98.2	Nonlinear optimization
31.	GHS_psdef/inline_1 †	504.	18.7	173.	144.	Inline skate
32.	PARSEC/Si5H12	19.9	0.379	44.1	153.	Density functional theory
33.	GHS_psdef/apache2 †	715.	2.77	135.	174.	3D structural problem
34.	Lin/Lin	256.	1.01	108.	277.	Eigenvalue problem
35.	ND/nd12k †	36.0	7.13	117.	505.	3D mesh problem

Give the number of entries in and flops to compute the factors, respectively, without supernode amalgamation and assuming no pivoting occurs. A † indicates that a problem is positive definite.

the positive-definite and indefinite modes of SSIDS. Here, the reported percent cost is the overhead of using the indefinite mode. The results only provide an indication because, as well as the extra cost of pivoting, there are some additional costs associated with using an LDL^T rather than LL^T factorization in the indefinite runs. Further, more attention has been paid to the indefinite implementation than to the positive-definite one (as Cholesky factorization can be implemented in a less constrained and hence faster parallel framework).

The typical overhead on the GPU is around 10–20%. This is comparable to the overheads on the CPU and helps confirm that the pivoting strategy has been implemented efficiently on the GPU.

To assess the efficacy of the a posteriori pivoting strategy, Figure 9 shows the percentages of pivots accepted in each column. We see that for the majority of matrices, around 85–95% of pivot columns succeed entirely, with the remainder partially succeeding. For those matrices with a large number of delayed pivots (1, 7, 11, and 30),



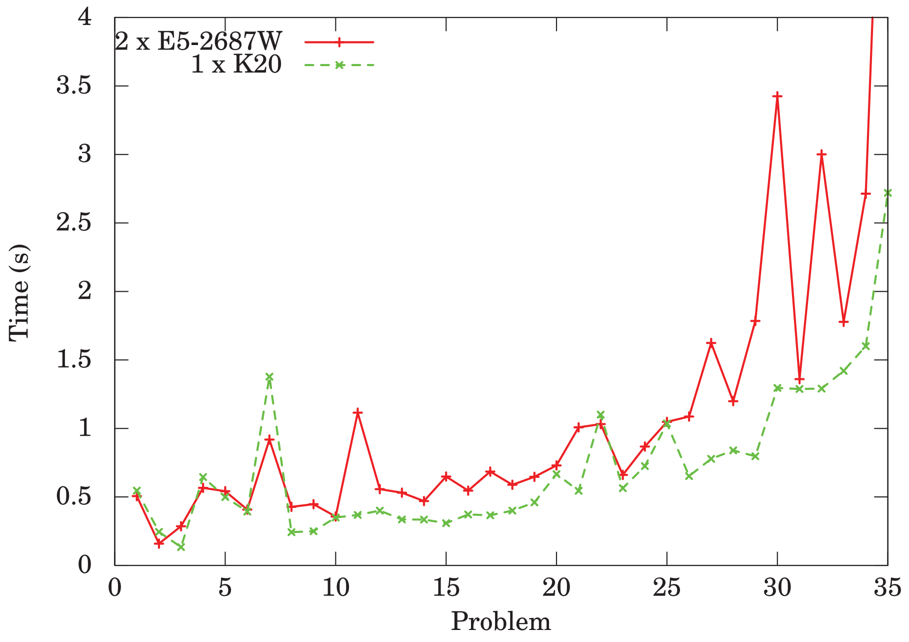
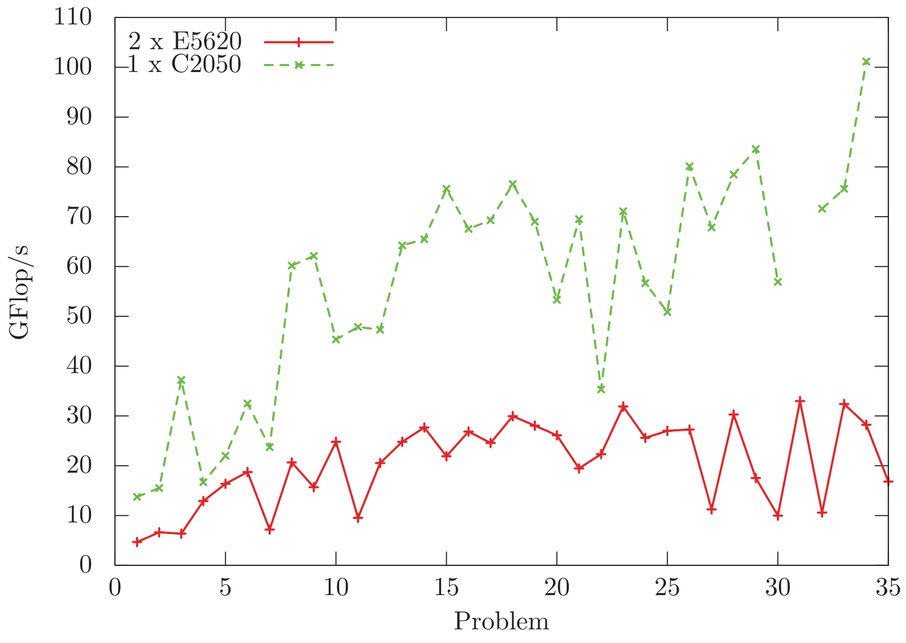


Fig. 7. Factorize time.


2 x E5620
1 x C2050
GFlop/s
Problem
0
10
20
30
40
50
60
70
80
90
100
110
0
5
10
15
20
25
30
35

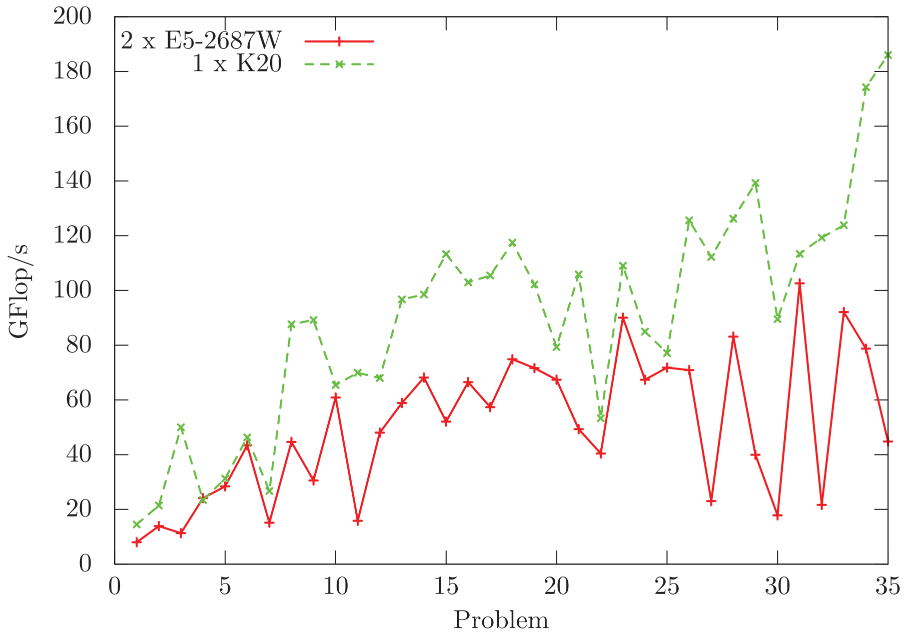

2 x E5-2687W
1 x K20
GFlop/s
Problem
0
20
40
60
80
100
120
140
160
180
200
0
5
10
15
20
25
30
35

Fig. 8. GFlop/s.

Table IV. Factorize Phase Times When Run with the Positive Definite and the Indefinite Kernels

Problem		CPU (E5-2678W)			GPU (K20)		
		indef	posdef	% cost	indef	posdef	% cost
13.	GHS_psdef/crankseg_1	0.527	0.466	13.1	0.352	0.312	12.8
14.	Rothberg/cfd2	0.468	0.413	13.4	0.352	0.271	29.9
15.	DNVS/thread	0.641	0.564	13.8	0.317	0.275	15.3
16.	DNVS/shipsec1	0.555	0.482	15.0	0.392	0.335	17.0
17.	DNVS/shipsec8	0.676	0.616	9.8	0.395	0.344	14.8
18.	Oberwolfach/boneS01	0.608	0.549	10.7	0.413	0.325	27.1
19.	GHS_psdef/crankseg_2	0.662	0.558	18.7	0.482	0.429	12.4
20.	Schenk_AFE/af_shell7	0.744	0.634	17.3	0.741	0.625	18.6
21.	DNVS/shipsec5	1.017	0.910	11.7	0.572	0.449	27.4
22.	AMD/G3_circuit	1.007	0.965	4.4	1.055	0.851	24.0
23.	GHS_psdef/bmwcra_1	0.666	0.585	14.0	0.606	0.535	13.3
24.	Schenk_AFE/af_0_k101	0.867	0.746	16.3	0.799	0.670	19.3
25.	GHS_psdef/ldoor	1.036	0.919	12.7	1.158	0.855	35.4
26.	DNVS/ship_003	1.091	0.965	13.0	0.694	0.617	12.5
28.	Um/offshore	1.224	1.142	7.2	0.877	0.772	13.6
29.	ND/nd6k	1.785	1.922	−7.1	0.702	0.645	8.8
31.	GHS_psdef/inline_1	1.379	1.249	10.4	1.384	1.210	14.4
33.	GHS_psdef/apache2	1.788	1.575	13.4	1.399	1.216	15.0
35.	ND/nd12k	6.836	7.301	−6.4	2.452	2.325	5.5

we see a significant departure from this behavior, as one might reasonably expect. For problems 1 and 7, only about 20% of columns completely succeed; this is in part skewed by the retesting of the same pivots until they have been delayed sufficiently that they can be safely eliminated. The need to fall back on the special-case kernel that handles breakdown at root nodes is rare: in our test set, it was not required if scaling was used (and only one problem required it if scaling was not used).

4.3. Factorization Performance

Table V gives a detailed breakdown of times and performance metrics between different parts of the factorize phase on the C2050. Results on Kepler class hardware are broadly similar, with small changes reflecting the different balance between computation and memory resources on the hardware. The results were captured with the profiling tool `nvprof` and only measure time spent in kernels. The “o” column represents the difference between the sum of these parts and the measured time for the entire factorize phase. This includes such things as memory management, transfer, and synchronization. As such, it is largely attributable to the dense factorization where information is bounced between the CPU and GPU.

The performance of the contribution and assembly kernels is clearly a major factor in dominating the CPU times. By capturing the node sizes involved, we validated the performance of our contribution kernel by comparing it against a simulation using the CUBLAS `dsyrk` routine to perform similar operations, both with and without the use of multiple streams to run multiple matrices in parallel. In all cases, our code was faster, and in the majority of cases, by at least 50%. The assembly phase data rate is calculated based on the number of entries in the child matrices on the assumption that each entry requires two floating-point loads, one integer load, and a floating-point store. The achieved data transfer rates are still respectable, especially due to the sparse nature of some loads (where the entire cache line is loaded but only a single value is used).

In contrast, the factorization kernel is slow, largely because it is bound on memory and host-device latencies. If this work could be effectively overlapped with the contribution block calculation (which would require synchronization between the two

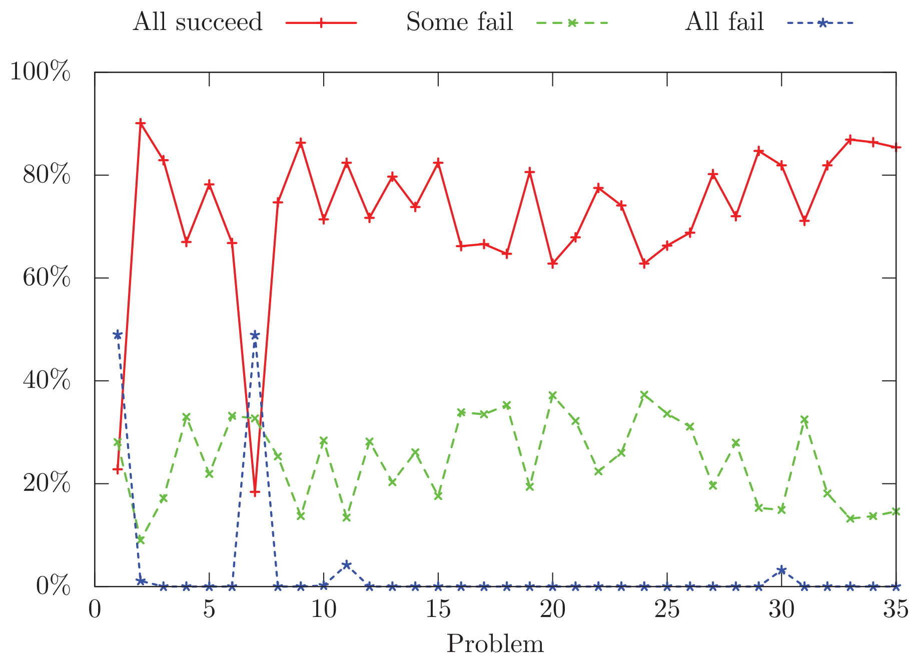


Fig. 9. Percentage of candidate pivot columns accepting a given number of pivots (candidate pivot columns can have at most eight pivots).

kernels), then it would not be unreasonable to expect a speedup of 2× over the current implementation for some matrices. However, as this will be difficult to achieve in an effective fashion, it is left as a topic for future work.

4.4. Solve

Figures 10 and 11 show (for a single right-hand side) the performance of the solve phase with and without using the presolve, compared with the CPU solve phase of HSL_MA97 with control%solve_mf=true (enabling parallel forward solve). Note that these figures show only the time for the solve phase, and not the overhead on the factorize phase required by the presolve.

These show that the GPU is able to deliver a speedup of about 2× on average over the CPU code, and significantly more using the presolve option. Those problems where the presolve significantly outperforms the traditional solve (4, 5, 20, 22, 25, 30, 33) are characterized by a large number of small nodes. As the substantial difference between the standard and presolve algorithms is the replacement of the triangular solve (trsv) with a matrix-vector multiply (gemv), it is unsurprising that profiling shows that the triangular solve kernel is the dominant cost for these problems. Both the trsv and gemv kernels are memory bound and capable of achieving similar peak bandwidth on large problems. Hence, it is only on problems with a significant number of small operations that we see the benefits of using the communication-avoiding presolve technique.

The additional cost of applying the presolve in the factorize phase can add as much as 44% (DNVS/thread) or as little as 11% (Newman/astro-ph) to the factorization time. On the best problems, the extra cost is amortized over only three solves, but on average it requires 20 to 30 solves. On some problems, there is no measurable advantage to using the presolve algorithm. Due to these performance overheads, the presolve algorithm must be explicitly enabled by the user.

Table V. Split of the Operations and Time Between Different Routines Within the Factorize Phase on the C2050, with Performance Metrics

		% Flops			% Time				GFlop/s		GB/s
	Problem	f	a	c	f	a	c	o	f	c	a
1.	Newman/astro-ph	33.9	0.7	65.4	41.8	15.2	6.2	36.8	14.1	184.1	20.9
2.	Andrianov/mip1	11.2	1.4	87.4	35.2	19.7	16.3	28.8	8.5	143.8	52.7
3.	PARSEC/SiNa	30.5	0.7	68.9	35.4	11.4	21.0	32.2	40.4	153.8	80.1
4.	Schmid/thermal2	42.0	0.3	57.6	53.1	8.5	9.9	28.5	16.8	123.3	23.6
5.	McRae/ecology1	36.4	0.3	63.3	58.1	9.3	13.5	19.0	18.1	134.8	24.6
6.	INPRO/msdoor	30.8	0.4	68.8	51.6	10.9	17.4	20.1	21.0	139.7	36.6
7.	GHS_indef/ncvxqp3	24.1	0.6	75.3	48.4	20.3	11.6	19.7	15.7	203.7	26.1
8.	Oberwolfach/gas_sensor	29.5	0.3	70.2	47.3	8.6	26.2	17.9	37.6	161.0	53.8
9.	ND/nd3k	24.2	0.5	75.3	34.3	13.2	31.4	21.0	48.3	163.9	68.6
10.	Boeing/pwtk	29.7	0.3	69.9	52.8	9.8	21.5	15.8	26.6	153.5	43.7
11.	GHS_indef/c-71	9.7	0.7	89.6	29.1	18.2	32.2	20.5	19.2	160.4	63.6
12.	BenElechi/BenElechi1	35.6	0.3	64.1	53.7	9.2	20.3	16.8	32.0	152.5	43.6
13.	GHS_psdef/crankseg_1	32.2	0.2	67.6	49.1	8.7	26.7	15.5	42.6	163.8	46.8
14.	Rothberg/cfd2	31.1	0.2	68.6	49.6	8.6	27.7	14.1	40.9	161.7	48.2
15.	DNVS/thread	49.3	0.2	50.6	56.6	6.2	22.4	14.8	63.8	165.9	49.7
16.	DNVS/shipsec1	30.9	0.2	68.9	47.8	7.6	27.9	16.7	43.2	164.5	52.5
17.	DNVS/shipsec8	32.8	0.2	67.0	47.7	7.8	28.0	16.5	47.4	164.9	55.4
18.	Oberwolfach/boneS01	27.9	0.2	71.9	45.9	7.7	33.4	13.0	46.3	164.0	51.4
19.	GHS_psdef/crankseg_2	35.0	0.2	64.8	50.9	8.2	26.6	14.3	46.2	163.8	46.7
20.	Schenk_AFE/af_shell7	34.8	0.3	64.9	54.7	9.4	23.2	12.8	34.4	151.7	43.7
21.	DNVS/shipsec5	36.1	0.2	63.7	51.9	7.7	27.3	13.1	48.7	163.2	55.1
22.	AMD/G3_circuit	26.2	0.3	73.5	53.7	10.4	21.9	14.1	21.6	149.3	39.3
23.	GHS_psdef/bmwcra_1	29.9	0.2	69.8	50.2	9.0	30.5	10.3	42.5	163.0	53.5
24.	Schenk_AFE/af_0_k101	34.9	0.3	64.8	54.3	9.1	24.2	12.4	36.7	153.1	44.5
25.	GHS_psdef/ldoor	29.8	0.3	69.9	-	-	-	-	-	-	-
26.	DNVS/ship_003	36.4	0.2	63.4	50.0	8.0	30.2	11.8	58.1	167.7	65.1
27.	PARSEC/Si10H16	21.3	0.6	78.1	28.9	14.3	44.1	12.7	67.3	161.1	103.1
28.	Um/offshore	27.8	0.2	71.9	45.9	9.0	34.6	10.5	48.2	164.9	61.3
29.	ND/nd6k	15.3	0.5	84.2	26.2	14.2	48.8	10.8	55.9	165.1	94.2
30.	Schenk_IBMNA/c-big	13.1	0.6	86.3	29.6	17.0	38.8	14.6	32.1	160.8	74.6
31.	GHS_psdef/inline_1	27.5	0.2	72.3	-	-	-	-	-	-	-
32.	PARSEC/Si5H12	21.4	0.7	78.0	26.3	16.1	47.4	10.2	77.6	157.0	109.7
33.	GHS_psdef/apache2	29.3	0.2	70.5	47.9	8.3	35.2	8.5	50.2	164.8	53.6
34.	Lin/Lin	18.6	0.3	81.1	32.6	9.4	50.8	7.2	60.2	167.8	88.1
35.	ND/nd12k	12.4	0.4	87.2	-	-	-	-	-	-	-

Data captured using `nvprof`. ‘f’ represents the dense matrix factorization, ‘a’ the assembly, ‘c’ the calculation of the contribution block, and ‘o’ other operations not reported in the profiling data, including data transfer, memory management, and synchronization. A - indicates missing data (insufficient memory).

It may be that a significant redesign of the triangular solve could dispatch large numbers of small operations more efficiently (see, e.g., the dedicated small matrix work in Hogg [2013]). However, the cost of the presolve in the factorize phase could be similarly reduced by limiting the inversion to a smaller band around the diagonal, rather than the entire diagonal block of each node.

5. CONCLUDING REMARKS

In this article, we have reported on our work to address the challenging problem of designing and developing an efficient and robust symmetric indefinite sparse (LDL^T) direct solver for use on NVIDIA GPUs. The new library-quality open-source solver is called SSIDS and is available from http://www.numerical.rl.ac.uk/spral/. SSIDS implements a multifrontal algorithm and one of its key features is that it allows all the frontal matrices to be factorized on the GPU. Furthermore, the code produces

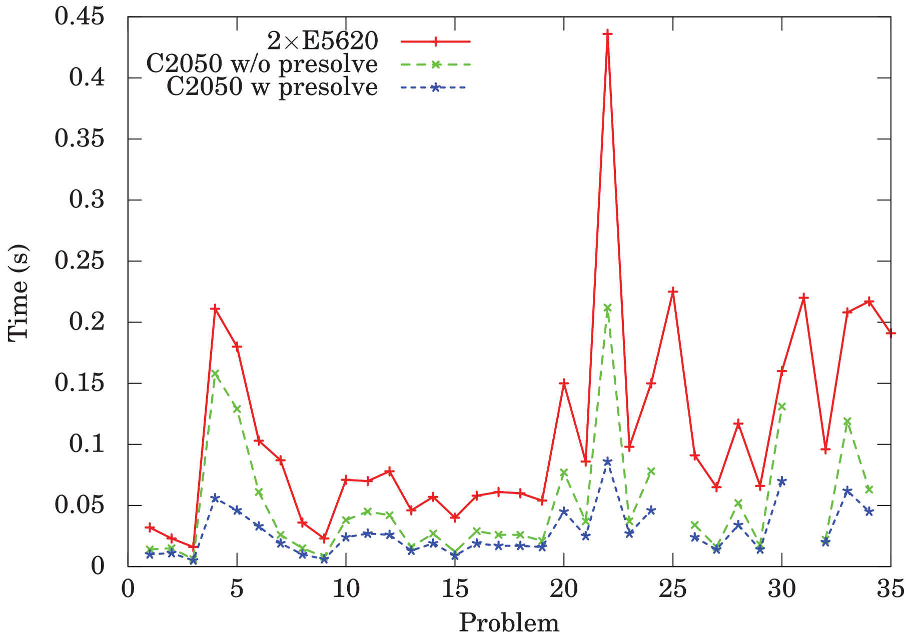


Fig. 10. Times for the solve phase in seconds, E5620/C2050.

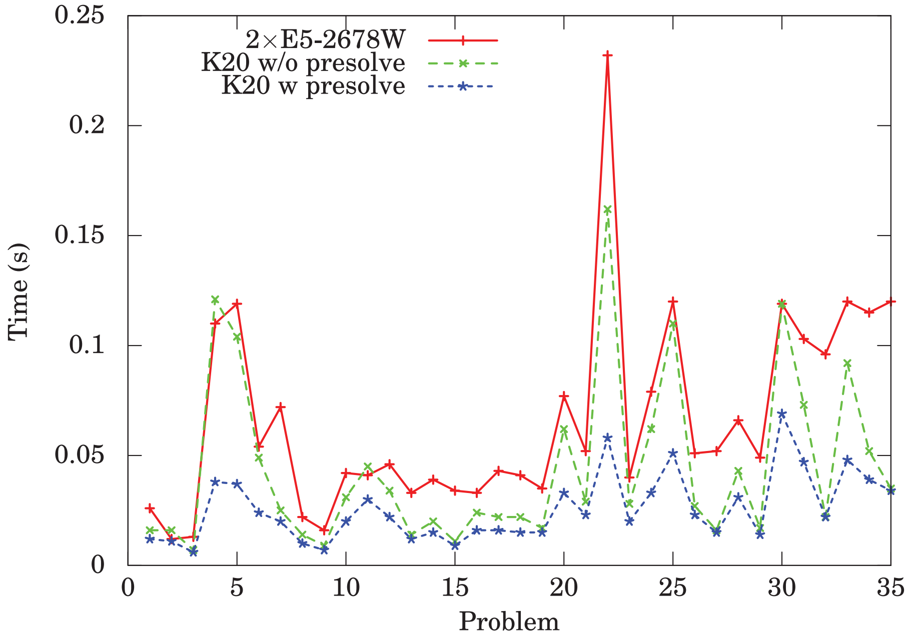


Fig. 11. Times for the solve phase in seconds, E5-2687W/K20.

bit-compatible results and incorporates threshold partial pivoting to maintain numerical stability: this has not been done in other GPU sparse solvers. Another novel feature is that SSIDS implements the solve phase on the GPU. Both the factorize and solve phases have been shown to achieve performance improvement over our recent multicore CPU code HSL_MA97.

The current implementation does not support working across multiple GPUs and is hence limited in the size of problems that can be solved to those that fit in the GPU memory (the most recent K40 cards have 12GB memory). This limitation can be overcome through the exploitation of tree parallelism and is a feature we plan to add to the code in the future.

ACKNOWLEDGMENTS

We wish to thank Mike Giles of the University of Oxford for his useful comments on a draft of this article. Our thanks also to the three anonymous referees whose recommendations helped improved our work, and also to Eric Boman for his work on reproducing our results and the improvements this led to.

We would also like to thank the University of Oxford Advanced Research Computing (ARC) facility for providing access to the K20 GPUs used for benchmarking. http://dx.doi.org/10.5281/zenodo.22558.

REFERENCES

P. R. Amestoy, T. A. Davis, and I. S. Duff. 1996. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl.* 17 (1996), 886–905.

P. R. Amestoy, T. A. Davis, and I. S. Duff. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Software* 30, 3 (2004), 381–388.

C. Ashcraft, R. G. Grimes, and J. G. Lewis. 1999. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl.* 20, 2 (1999), 513–561.

J. R. Bunch and L. Kaufman. 1977. Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comp.* 31 (1977), 1634–179.

T. A. Davis. 2006. *Direct Methods for Sparse Linear Systems*. SIAM.

T. A. Davis and Y. Hu. 2011. The University of Florida sparse matrix collection. *ACM Trans. Math. Software* 38, 1 (2011), 1:1–1:25.

I. S. Duff, A. M. Erisman, and J. K. Reid. 1989. *Direct Methods for Sparse Matrices*. Oxford University Press.

I. S. Duff, N. I. M. Gould, J. K. Reid, J. A. Scott, and K. Turner. 1991. Factorization of sparse symmetric indefinite matrices. *IMA J. Numer. Anal.* 11 (1991), 181–2044.

I. S. Duff and J. K. Reid. 1983. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Math. Software* 9, 3 (1983), 302–325.

A. George. 1973. Nested dissection of a regular finite-element mesh. *SIAM J. Numer. Anal.* 10 (1973), 345–363.

T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury. 2011. Multifrontal factorization of sparse SPD matrices on GPUs. In *Parallel and Distributed Processing Symposium (IPDPS'11)*. 372–383.

G. H. Golub and C. F. van Loan. 1996. *Matrix Computations* (3rd ed.). Johns Hopkins University Press.

A. Gupta and H. Avron. 2013. *WSMP: Watson Sparse Matrix Package. Part I - Direct Solution of Symmetric Systems. Version 13.06*. Technical Report RC 21886. IBM T. J. Watson Research Center, Yorktown Heights, NY. Retrieved from http://www.research.ibm.com/projects/wsmp.

N. J. Higham. 1995. Stability of parallel triangular system solvers. *SIAM J. Sci. Comput.* 16, 2 (1995), 400–413.

J. D. Hogg. 2013. A fast dense triangular solve in CUDA. *SIAM J. Sci. Comput.* 35, 3 (2013), C303–C322.

J. D. Hogg and J. A. Scott. 2010. *A Note on the Solve Phase of a Multicore Solver*. Technical Report RAL-TR-2010-007. STFC Rutherford Appleton Laboratory.

J. D. Hogg and J. A. Scott. 2011. HSL_MA97*: A Bit-Compatible Multifrontal Code for Sparse Symmetric Systems*. Technical Report RAL-TR-2011-024. STFC Rutherford Appleton Laboratory.

J. D. Hogg and J. A. Scott. 2012. *Achieving Bit Compatibility in Sparse Direct Solvers*. Technical Report RAL-P-2012-005. STFC Rutherford Appleton Laboratory.

J. D. Hogg and J. A. Scott. 2013. New parallel sparse direct solvers for multicore architectures. *Algorithms* 6 (2013), 702–725.

J. D. Hogg and J. A. Scott. 2014. Pivoting strategies for tough sparse indefinite systems. *ACM Trans. Math. Software* 40, 1 (Sept. 2013), 4:1–4:19.

HSL 2013. HSL. A collection of Fortran codes for large-scale scientific computation. (2013). http://www.hsl.rl.ac.uk.

G. Karypis and V. Kumar. 1998. METIS: A software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing orderings of sparse matrices - Version 4.0. (1998). http://www-users.cs.umn.edu/ karypis/metis/.

G. Karypis and V. Kumar. 1999. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.* 20 (1999), 359–392.

K. Kim and V. Eijkhout. 2012. *A Parallel Sparse Direct Solver via Hierarchical DAG Scheduling*. Technical Report TR-12-04. Texas Advanced Computing Center (TACC).

K. Kim and V. Eijkhout. 2013. Scheduling a parallel sparse direct solver to multiple GPUs. In *Proceedings 14th IEEE Workshop on Parallel and Distributed Scientific and Engineering Computing*.

G. P. Krawezik and G. Poole. 2010. Accelerating the ANSYS direct sparse solver with GPUs. (2010). In *Symposium on Application Accelerators in High Performance Computing* (SAAHPC'10).

X. Lacoste, P. Ramet, M. Faverge, Y. Ichitaro, and J. Dongarra. 2013. *Sparse Direct Solvers with Accelerators Over DAG Runtimes*. Technical Report No. 7972. INRIA, University of Bordeaux, France.

J. W. H. Liu. 1985. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Software* 11, 2 (1985), 141–153.

R. F. Lucas, G. Wagenbreth, J. J. Tran, and D. M. Davis. 2012. Multifrontal Sparse Matrix Factorization on Graphics Processing Units. (2012). Unpublished manuscript, available at ftp://ftp.mosis.com/isi-pubs/tr-677.pdf.

MUMPS. 2013. MUMPS: A MUltifrontal Massively Parallel Sparse Direct Solver. Retrieved from http://graal.ens-lyon.fr/MUMPS/.

J. K. Reid and J. A. Scott. 2011. Partial factorization of a dense symmetric indefinite matrix. *ACM Trans. Math. Software* 38, 2 (2011), 10:1–19.

O. Schenk and K. Gärtner. 2006. On fast factorization pivoting methods for symmetric indefinite systems. *Electronic Trans. Numer. Anal.* 23 (2006), 158–179.

W. F. Tinney and J. W. Walker. 1967. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE* 55 (1967), 1801–1809.

C. D. Yu, W. Wang, and D. L. Pierce. 2011. A CPU-GPU hybrid approach for the unsymmetric multifrontal method. *Parallel Comput.* 37, 12 (2011), 759–770.

Received March 2014; revised October 2014; accepted February 2015