Accelerating Sparse Cholesky Factorization on GPUs

Submitted to ***"IA^3 Workshop on Irregular Applications: Architectures & Algorithms"****,*
to be held ***Sunday, November 16, 2014****,*
Colorado Convention Center, New Orleans, LA USA
http://cass-mt.pnnl.gov/irregularworkshop.aspx

Steven C. Rennich
Sr. Engr. DevTech Compute
NVIDIA, Santa Clara, CA
Email: srennich@nvidia.com

Darko Stosic
DevTech Compute Intern
CIn, UFPE
Recife, Brazil

Timothy A. Davis
Professor, CSE
Texas A&M University
College Station, TX

***Abstract*—Sparse direct factorization is a fundamental tool in scientific computing. As the major component of a sparse direct solver, it represents the dominant computational cost for many analyses. While the substantial computational capability provided by GPUs (Graphics Processing Units) can help alleviate this cost, many aspects of sparse factorization and GPU computing, most particularly the prevalence of small/irregular dense math and slow PCIe communication, make it challenging to fully utilize this resource.**

In this paper we describe a supernodal Cholesky factorization algorithm which permits improved utilization of the GPU when factoring sparse matrices. The central idea is to stream branches of the elimination tree (subtrees which terminate in leaves) through the GPU and perform the factorization of each branch entirely on the GPU. This avoids the majority of the PCIe communication without the need for a complex task scheduler. Importantly, within these branches, many independent, small, dense operations are batched to minimize kernel launch overhead and several of these batched kernels are executed concurrently to maximize device utilization. Supernodes towards the root of the elimination tree (where a branch involving that supernode would exceed device memory) typically involve sufficient dense math such that PCIe communication can be effectively hidden, GPU utilization is high and hybrid computing can be easily leveraged.

Performance results for commonly studied matrices are presented along with suggested actions for further optimizations.

I. Introduction

Achieving high performance for sparse direct solvers in general, and sparse Cholesky factorization in particular, is a very well researched topic [1]. While the specific performance of such factorization depends strongly on the characteristics of the matrix being factored, highly efficient algorithms and implementations exist for the CPU which can achieve a large fraction of the CPU's theoretically available flops for a wide range of industrially important matrices.

Figure 1 shows modern GPUs are capable of dramatically higher computation rates than CPUs [2] and many researchers have investigated using GPUs to accelerate matrix factorization. In the case of *dense* factorization of large matrices, the results are impressive, with nearly the full computational capability of the GPU being fairly easily applied to the factorization operation [3]. However, for the case of *sparse* factorization, while the results can be considered encouraging, they are mixed [4] [5] [6]. That is, for some matrices, particularly large matrices with substantial fill, such that they contain a relatively large amount of dense math, significant speedup vs. the CPU can be achieved. However, for other matrices, particularly those that are smaller or have substantially lower fill ratios and consequently less dense math, relatively little or no speedup in the factorization is achieved using GPUs.

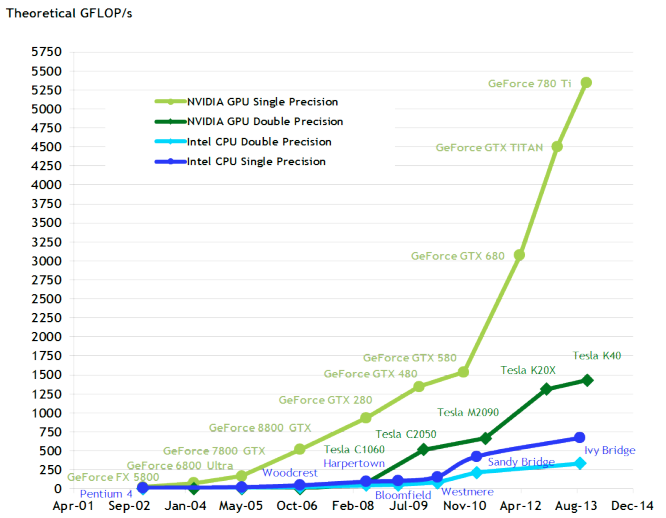

Fig. 1. Historical trends of growth in computational performance, floating point operations per second (flops), for CPUs and GPUs. [2]

Many problems of interest are in this last category: large matrices for which the factor's fill ratio is relatively low. A prominent example would be the stiffness matrix arising from a shell or beam Finite Element Analysis model in structural mechanics (*e.g.* the analysis of airplanes or automobiles).

Consequently, the objective of this work is to identify means

TABLE I
TEST MATRICES

Matrix	dimension	nnz in A	fill ratio
Fault_639	638,802	27,245,944	78.5
nd24k	72,000	28,715,634	22.6
inline_1	503,712	36,816,170	9.2
Emilia_923	923,136	40,373,538	82.7
bonesS10	914,898	40,878,708	9.7
ldoor	952,203	42,493,817	6.0
bone010	986,703	47,851,783	29.3
Hook_1498	1,498,023	59,374,451	51.6
Geo_1438	1,437,960	60,236,322	78.7
Serena	1,391,349	64,131,971	83.1
audikw_1	943,695	77,651,847	31.7
Flan_1565	1,564,794	114,165,372	24.9

by which sparse Cholesky factorization, defined as

$$A = LL^t$$

where the sparse, SPD (symmetric positive definite) matrix A is factored as the product of a sparse lower triangular matrix, L, and its transpose, can be further accelerated using GPUs. This investigation is carried out in the context of the left-looking, supernodal sparse Cholesky code CHOLMOD [7], part of the SuiteSparse [8] family of sparse solvers, using GPUs from NVIDIA. However, we expect many of the findings and algorithmic details will be applicable to GPU acceleration of many other sparse factorization algorithms.

The scope of this paper is to demonstrate how a novel algorithm can alleviate the performance issues observed in previous work, illustrate its advantages using model problems, and then demonstrate some of the algorithm's performance potential when implemented within a complete, mature, high-performance sparse solver code.

II. SETUP

For this work we emphasize testing on recent hardware, for both the CPU and GPU, and on a wide variety of industrially-relevant matrices. The test matrices were all taken from the Florida Sparse Matrix Collection [9]. As we will be considering only Cholesky factorization here, tests are performed using the 100 largest, real, SPD matrices available from the collection. More detailed performance results are presented for the 12 largest of these matrices listed in Table I, where the fill ratio is the ratio of number of non-zeros (nnz) in L divided by the number of non-zeros in A.

For all test results presented in this paper, the hardware used is:

CPU: Dual-socket Intel Xeon (Ivy Bridge) E5-2690 v2 @ 3.00 Ghz. using Intel Parallel Studio 2013 [10].

GPU: NVIDIA Tesla K40 with maximum boost clocks of 3004 Mhz. (memory) and 875 Mhz. (core) using CUDA 6.5 [11].

In this work, with the exception of some trends presented in Figure 1, we are only concerned with double-precision (dp) arithmetic.

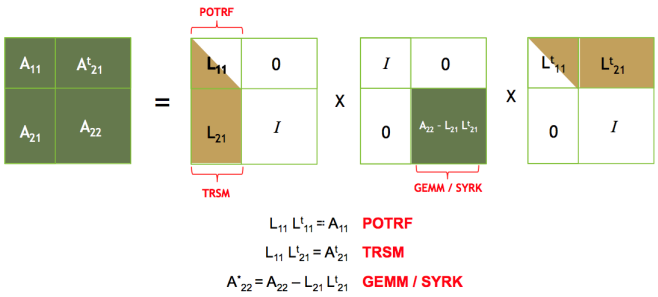

Fig. 2. Schematic showing BLAS/LAPACK routines used in dense block factorization. Brown denotes a factorized portion of the matrix.

Sparse matrix factorization typically makes extensive use of the "Basic Linear Algebra Subprograms" (BLAS) [12] and "Linear Algebra Package" (LAPACK) [13] libraries. For best performance, highly optimized versions of these libraries have been used on both the CPU (Intel MKL [10]) and GPU (NVIDIA cuBLAS [14]).

The version of CHOLMOD included in the 4.3.1 version of the publicly-available SuiteSparse package is used to provide CPU-only and CPU+GPU reference performance results. In all cases the Metis 4.0 fill-reducing reordering was used. [15]

III. BACKGROUND AND PREVIOUS WORK

Here we only briefly discuss the mechanics of sparse matrix factorization. Figure 2 is a schematic showing the partial Cholesky factorization of a symmetric dense matrix and the BLAS / LAPACK routines that are used. After the partial factorization, the tan portion of the matrix has been factored and the green portion, the Schur complement, remains to be factorized. Any dense SPD matrix can be factored by repeated application of this operation. Any sparse SPD matrix can also be factored by extracting dense sub-problems (fronts or supernodes) and applying exactly the same technique. In either case, the fact that factorizing of one portion of the matrix, A_{11} and A_{21}, updates the values in other portions, A_{22}, implies that this block factorization must be performed in some order, typically represented by an elimination tree [16].

The specific double-precision BLAS and LAPACK routines used in Cholesky factorization are:

DPOTRF : direct Cholesky factorization of a dense matrix (LAPACK)
DTRSM : triangular system solution (BLAS)
DGEMM/DSYRK : general/symmetric matrix-matrix multiplication (BLAS)

The important point is that most of the factorization occurs in BLAS operations involving dense matrices which can achieve very high performance on both the CPU and the GPU. Consequently, a simple approach to accelerating a sparse factorization is to just perform the BLAS routines on the GPU.

However, when the input matrices to these BLAS operations are 'small', there are some issues which prevent the routines from actually being accelerated on the GPU. (Illustrated here using the DGEMM operation - the most common BLAS

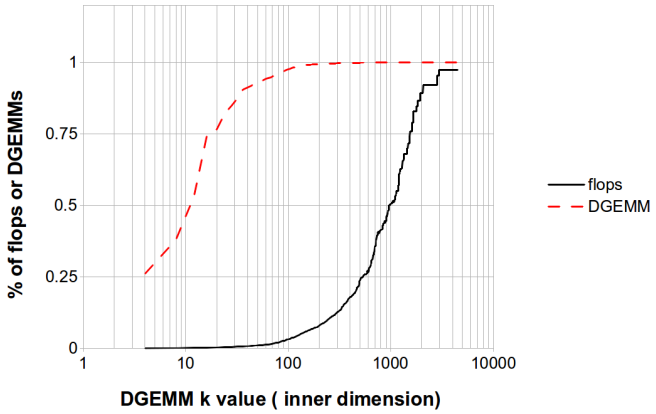

Fig. 3. Ratios of flops (black) and DGEMM calls (dashed red) with an inner dimension k below a specific value for the audikw_1 matrix.

operation used in sparse factorization. For clarity, note that DGEMM involves multiplying an $m \times k$ matrix by a $k \times n$ matrix to get an $m \times n$ matrix as a result.)

1) Computational intensity: When the inner dimension, k, of the DGEMM is small, the operation on the GPU is bound by device memory bandwidth which prevents the operation from achieving the GPU's peak performance.
2) Device utilization: When the DGEMM output matrix size is small ($m \times n$), there is insufficient parallelism to fully utilize the GPU.
3) PCIe bandwidth: The time required to communicate the input/output data between the CPU and GPU via PCIe will reduce the effective performance.
4) Launch latency: Every kernel launch requires some amount of launch overhead, typically 4-15 μsec. This will further reduce effective performance.

For these reasons, a large or even dominant portion of the dense math encountered when factorizing a matrix is too small to be easily accelerated on the GPU. To illustrate this, figure 3 plots the ratio of a) flops, and b) DGEMM calls, performed with an inner dimension k below a specific value vs. the total number of flops or DGEMM calls for the audikw_1 matrix, extracted from the CPU version of CHOLMOD. Most of the flops are in DGEMM calls with large k, but most of the DGEMM calls have small k. Specifically, for audikw_1, 50% of the flops occur in DGEMM calls with an inner dimension, $k > 1024$ and and will achieve performance of more than 1 Tflops on the GPU. However, 50% of the calls to DGEMM are made with an inner dimension $k < 128$ and their performance on the GPU will be much worse.

Distributions for DTRSM and DSYRK behave somewhat differently (but DGEMM is the dominant operation) and the distributions will be different for different matrices. The audik_1 matrix is in the middle of the spectrum. Others, like Serena, with a high fill-in ratio, will have a greater portion of the flops in operations with $k > 1024$ and fewer calls to DGEMM with $k < 128$. While ldoor, with a very low fill-in ratio, will be the opposite.

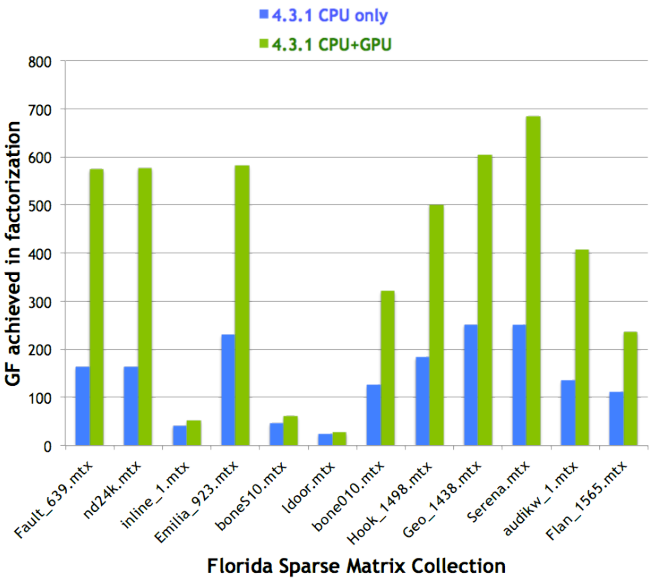

Fig. 4. Previous results showing comparison of factorization rates using CHOLMOD from SuiteSparse 4.3.1 using the CPU (blue) vs. the CPU+GPU (green) from [18]. This work made extensive effort to overlap all of a) device computation, b) host computation and c) PCIe communication.

The point is that not all BLAS operations can be accelerated by simply moving them to the GPU. So researchers have constructed a variety of schemes to move only those BLAS operations that can be accelerated to the GPU and to try to overlap PCIe communication with GPU computation and CPU computation as much as possible [17] [6] [5]. Some relevant results from this type of effort are shown in figure 4 from [18] which focused on hybrid computing and hiding (aka shadowing) PCIe communications behind GPU and CPU computation as much as possible. Here it can clearly be seen that matrices with large fill ratios, which consequently tend to have the majority of the flops occurring in BLAS calls with large input matrices and with large inner dimensions k, (high computational intensity, and plenty of parallelism to fully utilize the GPU) show good acceleration on the GPU. Conversely, those matrices with the lowest fill ratios show very modest levels of acceleration on the GPU since the factorization of those matrices is dominated by small BLAS operations which remain on the CPU. This is the primary issue we seek to remedy in the present work.

IV. Algorithm

To construct an algorithm which minimizes the detrimental effects of 3 of the 4 "small dense BLAS" issues itemized in the previous section, we observe that:

1) 'batching' (using a single kernel to perform many BLAS operations) can be used to minimize the effect of launch latency
2) concurrent kernels (i.e. simultaneous execution of multiple kernels on the GPU - possible when an individual kernel isn't utilizing all of the GPU's resources) can be used to maximize GPU utilization

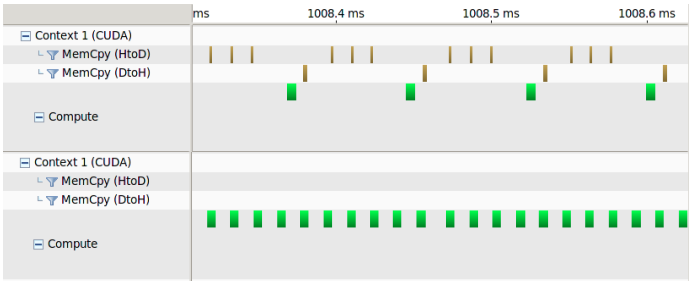

Fig. 5. Comparison of the performance of a series of host-interface DGEMMs, vs. a series of device-interface DGEMMs for $m, n, k = 16$ showing the effect of PCIe communication and kernel launch latency. All kernels are launched in succession - there are no intervening computations. (Note that the two cases were not run concurrently, but time scales are identical and profiles are stacked vertically for comparison purposes.

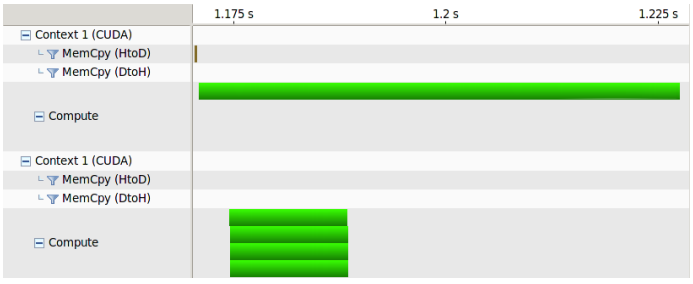

Fig. 6. Comparison of the performance of a single batch of 8192 DGEMMs with $m, n, k = 16$ vs. the same set of DGEMMS split into four batches and run concurrently on the GPU. (Note that the two cases are not run concurrently, but time scales are identical and timelines are stacked vertically for comparison purposes.

3) by placing a large amount of matrix data on the GPU and performing all of the factorization steps (DPOTRF, DTRSM, DGEMM, DSYRK) on the GPU, communication across PCIe can be almost completely avoided.

To illustrate this, figure 5 shows two timelines (generated using the Nvidia Visual Profiler [19]) performing repeated DGEMMs with $m, n, k = 16$, using cuBLAS [14]. The first shows the profile of host-based DGEMMs where the A, B and C matrices need to be first copied to the GPU and then the result needs to be copied back to the CPU. Each DGEMM kernel takes approximately 6 μsec and achieves 1.2 Gflops (dp). However, the time required for PCIe communications, and the approximately 10 μsec of launch latency between kernels, results in an effective performance of only 100 Mflops. The second timeline in the same figure shows repeated DGEMMs using only data on the GPU and avoiding PCIe communication. Launch latency is still present, but by avoiding all PCIe communications (and the launch latency for the memory copies) effective performance is accelerated by 5x and rises to 500 Mflops.

Launch latency can be almost entirely eliminated by 'batching' small BLAS or LAPACK operations. That is, using a modified kernel, which takes $lists$ of all input arguments (such as m, n, k dimensions, pointers to A, B and C matrices, etc. for DGEMM) and loops through the list performing many independent dense computations with a single kernel launch. Figure 6 shows two timelines, the first being the case of a single 'batched' DGEMM kernel computing 8192 individual DGEMMs with $m, n, k = 16$. For this batched case, elimination of launch latency improves aggregate performance to 1.2 Gflops. The performance of such small DGEMM kernels is limited since they can utilize only a small fraction of the GPU's resources. But this permits multiple instances of such a batched DGEMM kernel to be executed simultaneously (concurrent kernels using CUDA streams). The lower timeline in figure 6 shows the case of four batched DGEMM kernels, each computing 2048 DGEMMs for which aggregate performance rises to 4.8 Gflops. Note however, the case of four concurrent kernels is used only for visualization purposes. The optimal number of concurrent kernels for this case is 64 (each kernel computing a batch of 128 DGEMMs) which gives a performance of 34 Gflops - a 340-fold improvement over the initial implementation.

While most effective for DGEMM, this improved performance of BLAS routines obtained through batching and concurrent kernels applies to DTRSM, DSYRK and DPOTRF as well and this forms a fundamental component of the proposed algorithm.

However, to take advantage of such batching and concurrency, large collections of independent dense BLAS operations are needed. As well, since many matrices of interest have factors which are too large to fit in GPU memory, any batching and concurrency scheme must also permit streaming the matrix data through the device so that matrices of any size can be accommodated.

Figure 7 shows a simple schematic of a supernode elimination tree as might be seen in a Cholesky factorization. Circles represent supernodes, lines represent parent/child relationships and the root node is at the top of the figure. All of the independent supernodes are grouped in 'levels' shown as black boxes. The factorization of any supernode is only dependent on the factored supernodes in previous levels. Consequently, within a level all supernodes are independent and can be factored simultaneously. High computational performance will be achieved by computing all of the independent BLAS and LAPACK operations within a level using concurrent, batched kernels.

Figure 8 shows the number of supernodes and the sum of DGEMM and DSYRK operations in each independent level for the audikw_1 model. Clearly the lowest level, with 24,700 supernodes and 36,068 DGEMM or DSYRK operations, has more than enough to fill many batched, streamed kernels. For higher levels, the number of supernodes per level drops off rapidly with the top 10 levels having only one supernode. While the number of DGEMM or DSYRK operations falls for the higher levels as well, it never drops below 920. For the higher levels the average size of the supernodes is larger which means both a) the individual BLAS operations run longer so it takes fewer operations to amortize the kernel launch latency and b) the BLAS operations are larger and fill the device more quickly so fewer concurrent kernels are needed to completely

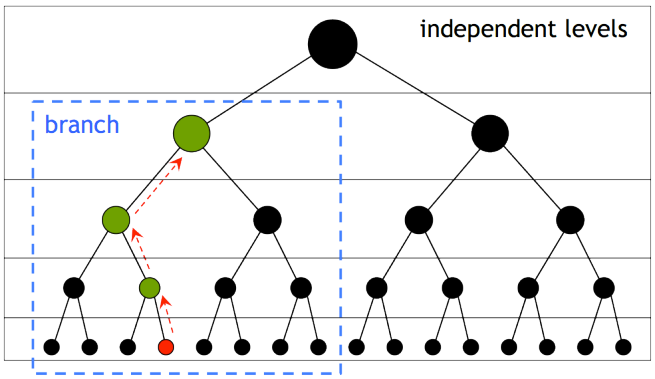

Fig. 7. Simple diagram of an elimination tree for Cholesky factorization. Circles represent supernodes and lines represent parent/child dependencies. Red arrows show how the factored red supernode, a descendant of each of the green supernodes, would need to be assembled. The blue box shows a possible branch of the elimination tree.

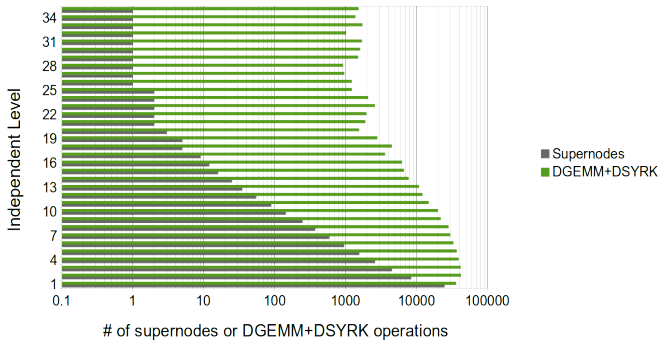

Fig. 8. Number of independent supernodes (grey) and number of DGEMM+DSYRK operations (green) in each independent level of the elimination tree for audikw_1.

utilize the device. The result is that for the matrices studied, there are plenty of BLAS and LAPACK operations at all levels to support effective batching and concurrency.

The remaining issue to resolve is the case where the factor is too large to fit on the GPU. We solve this by only copying a 'branch' of the original matrix to the GPU, such that the factored branch fits on the GPU. (Where a branch is a subtree with terminates in leaves.) This is shown in Figure 7 by the dashed blue box. Assembling any supernode in the branch might require the factored supernodes in the levels below. By placing a branch of the elimination tree on the GPU, thus ensuring that any lower-level result is already present on the GPU, and performing all the factorization operations on the GPU, this entire branch can be factored without requiring any communication with the CPU across the PCIe bus.

For the case of large matrices, multiple branches are simply streamed through the GPU. For those nodes of the elimination tree which are above any branches (that is, the factor of a branch containing these supernodes would be too large to fit on the GPU) existing methods, particularly those from [18], are used for factorization. While these supernodes might still have many small supernodes as descendants (requiring small BLAS and LAPACK operations) they very likely have

```
branches ← scan elimination tree for A to construct
  appropriately sized branches
for all branches do
  levels ← arrange branch supernodes into levels
  copy unfactored A data in branch to GPU
  for all levels do
    using concurrent, batched kernels:
      initialize buffers and construct all scatter maps
      assemble supernodes (DSYRK)
      assemble lower panels (DGEMM)
      factor supernodes (DPOTRF)
      factor lower panels (DTRSM)
    copy factored supernodes back to CPU
  end for
end for
for all remaining supernodes do
  simultaneously:
    assemble small descendants on CPU
    assemble large descendants on GPU
  block factorization of supernode diagonal block
    on CPU and, depending on size, GPU
  block factorization of lower panel on CPU or
    GPU depending on size
end for
```

Fig. 9. Sparse supernodal Cholesky factorization algorithm.

sufficient large operations, such that both CPU computation and PCIe communication can be executed concurrently with (hidden behind) computation on the GPU.

Summarizing the algorithm, if there are GPU versions of the required BLAS and LAPACK routines, and a few additional GPU kernels for mapping and scattering, once the A data pertaining to a branch of the elimination tree has been copied to the GPU, the entire branch can be factored without any need for PCIe communication, thus completely eliminating the biggest impediment previously encountered when accelerating sparse Cholesky factorization on the GPU. To achieve high computational performance, BLAS/LAPACK operations within an independent level of the elimination tree can be batched to minimize kernel launch overhead and these batched kernels can be executed concurrently to fully utilize the GPU's computational resources. Large matrices are decomposed into multiple branches which are streamed through the GPU. The factorization of supernodes which do not fit into a branch due to limited GPU memory are expected to involve sufficient dense math such that existing GPU accelerated factorization methods perform well.

The suggested supernodal Cholesky factorization algorithm is presented in figure 9.

V. Implementation

A private version of CHOLMOD has been modified to develop and test the performance this algorithm. The benefit of starting with a complete, industrial-strength, direct sparse

factorization library is that we can be confident the performance observed is representative of what an end-user might see - no simplifying assumptions or limitations. However, it also means that the complete implementation is both quite involved and, as yet, by no means optimal. Only the most important implementation aspects are covered here.

The first implementation issues are the code and data structure modifications required to manage dividing the existing elimination tree into branches and ordering the supernodes into levels. That process is straightforward using the existing data structures describing the elimination tree.

GPU-specific work is required to construct the batched versions of DPOTRF, DTRSM, DSYRK and DGEMM which accept $lists$ of pointers as inputs and which loop sequentially over the lists to perform all the necessary computations. In the present case this was done by just creating minimally modified versions of the cuBLAS code for DTRSM, DSYRK and DGEMM. An NVIDIA-internal version of DPOTRF has been modified in a similar manner. Such modifications are not difficult, but it is recognized that other developers might not have access to the necessary GPU source code. An aspect of this research is to suggest considering the creation of a library of such batched routines.

While the branches of the elimination tree are being factored on the GPU, the sole task of the CPU is to package the BLAS/LAPACK operation into batches and submit the batched kernels to the GPU. This can be done in a large number of different ways. In the present case the levels in the elimination tree have been divided into three categories, with each category being batched in a different manner.

- Lowest level: The supernodes in the lowest level of the elimination tree have no descendants. Hence they do not require assembly and do not call DSYRK or DGEMM. In this case only DPOTRF and DTRSM (and supporting mapping and initialization) operations are batched. Operations are grouped in batches of 16.
- Mid levels: In the mid levels the supernodes have a few descendants, but not a sufficient number to effectively fill a batch. So batches are created by aggregating operations required for 16 supernodes. That is, all of the operations to assemble the descendants for 16 supernodes form one batch.
- Upper levels: Levels with 64 or fewer supernodes, using batches of 16 supernodes at a time, would not permit sufficient concurrency. So in these levels the batches are formed by the descendants of just each individual supernode. Supernodes in the higher levels typically have many descendants. When there are fewer than 4 supernodes, multiple batches are created from the descendants of each supernode.

For all levels, concurrency is achieved by simultaneously executing batches/operations in 16 CUDA streams.

When performing these batched BLAS operations, temporary GPU memory buffers are required to store intermediate output. For example, the Schur complement update from a descendant before it gets assembled. In the current implementation, these temporary buffers are just whatever GPU memory is not occupied by the factor of the branch or the supernode maps. So the size of the branches are adjusted to be as large as possible such that the factor, plus the required temporary memory, fills the available GPU memory.

To present just a snapshot of how this works, figure 10 shows a timeline of supernodes from the lowest-level of the elimination tree ($i.e.$ leaves) of the inline_1 matrix being factored without the benefit of batching (but using 4 CUDA streams for concurrency). This shows the full factorization (DPOTRF and DTRSM) of 32 supernodes (plus the beginning operations for 3 more supernodes). Note the large amount of idle time (whitespace) and limited concurrency (lack of overlap between streams) due to launch latency. In the figures grey and white backgrounds represent different CUDA streams used achieve concurrency. Colored blocks represent compute kernels. Gold blocks represent PCIe communication - all but the very leftmost gold blocks are GPU to CPU communication copying the factored supernodes back to the CPU.

By contrast, figure 11 shows the timeline generated from the factorization of 128 leaf supernodes of the inline_1 matrix when performed in 8 batches of 16 supernodes. The horizontal scale (time) is not the same for the two timelines (figures 10 and 11). The point is that there is relatively much less idle time and relatively much greater concurrency as compared to the non-batched case shown in figure 10. The performance for the batched case shown is 4.8x that of the non-batched case. However, for performance benchmarking, much larger batches and many more streams are used - the simpler cases are shown here only for clarity. Timelines for the mid and upper levels are not shown, but the effect is the same.

It must be emphasized that this version of CHOLMOD, modified to leverage the current branches algorithm, did not exploit CPU/GPU hybrid processing when the branches of the elimination tree were being factored on the GPU. Only those supernodes which were not included in branches leveraged hybrid computing. That is, the CPU was used to sort BLAS/LAPACK operations, assemble them into batches and launch kernels on the GPU to factor a branch. However, while doing this, which was not computationally intensive, the CPU was not being used to perform any actual factorization computations. Implementing hybrid computing to better leverage this idle CPU resource should provide a straightforward way to achieve significant further performance improvements.

VI. Results

Figure 12. shows the performance (Gflops) achieved for the sparse factorization, using the described algorithm implemented in CHOLMOD vs. the previous (SuiteSparse 4.3.1) GPU-accelerated version of CHOLMOD and vs. the CPU version of CHOLMOD for the 12 largest real SPD matrices from the Florida Sparse Matrix collection. For the matrices whose factors had the lowest fill ratios, boneS10, inline_1 and ldoor, for which better GPU acceleration was the primary motivation for this work, the GPU performance has risen

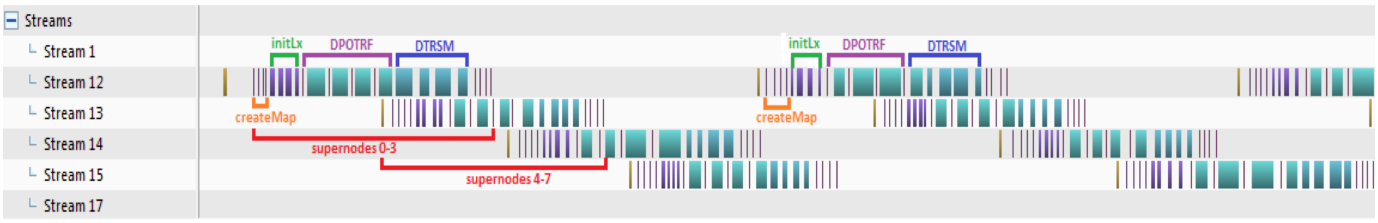

Fig. 10. NVIDIA Visual Profiler output showing the factorization of 32 leaf supernodes of the inline_1 matrix for the case no batching. Alternating grey and white backgrounds indicate separate CUDA streams. Colored blocks represent compute kernels (as labeled). Gold blocks represent GPU↔CPU communication.

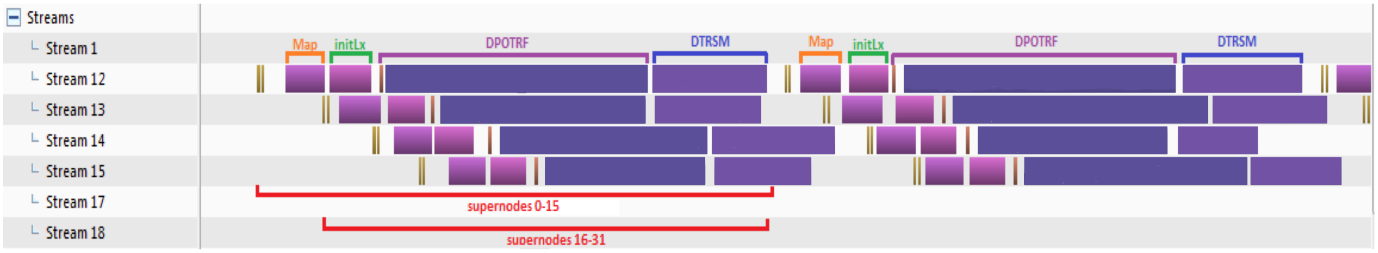

Fig. 11. NVIDIA Visual Profiler output for 128 leaf supernodes of the inline_1 matrix in batches of 16. Alternative grey and white backgrounds indicate separate CUDA streams. Colored blocks represent compute kernels (as labeled). Gold blocks represent GPU↔CPU communication.

dramatically, from an average of 1.2x vs. the CPU to an average of 2.2x vs the CPU. This clearly demonstrates that the algorithm is achieving its goal of accelerating a much greater portion of the factorization computation on the GPU.

Compared to the previous version of GPU accelerated results, the peak performance (Serena) rises from 684 Gflops to 783 Gflops and demonstrates that, even for large matrices with substantial fill, use of the current algorithm results in a significant performance benefit. The peak speedup for the GPU accelerated factorization rises from 3.5x to 4.1x vs. the CPU-only version.

Speedups over the CPU-only case exist for many matrices beyond what was presented in Figure 12. Figure 13 shows the current GPU+CPU speedup vs. the CPU-only performance for the 100 largest SPD matrices in the Florida Collection, ordered by size of A. (Results for 6 of the 100 matrices are missing due to errors in the test version of CHOLMOD.) The current method uses the GPU to successfully accelerate the majority of the 100 matrices studied and, in fact, works well even for matrices that are quite small. As a point of reference, matrix number 30, Dubcova2, which is roughly on the boundary separating poorly accelerated matrices from well accelerated matrices, has a dimension of only 65,025 and a fill ratio of only 5.4. The average speedup vs. the CPU for all 94 tested matrices is 1.75x and the geometric mean is 1.6x. For matrices number 31 and higher, the average and geometric means of the speedups are both approximately 2x.

Figure 14 shows the speedup of the algorithm presented here vs. the previous best CPU+GPU accelerated algorithm (SuiteSparse 4.3.1). Clearly there are some performance regressions but, except for the smallest 30 matrices, the general trend is again good speedup. Average speedup for the 94 tested matrices is 1.2x. Average speedup for matrices 31 and greater is 1.3x.

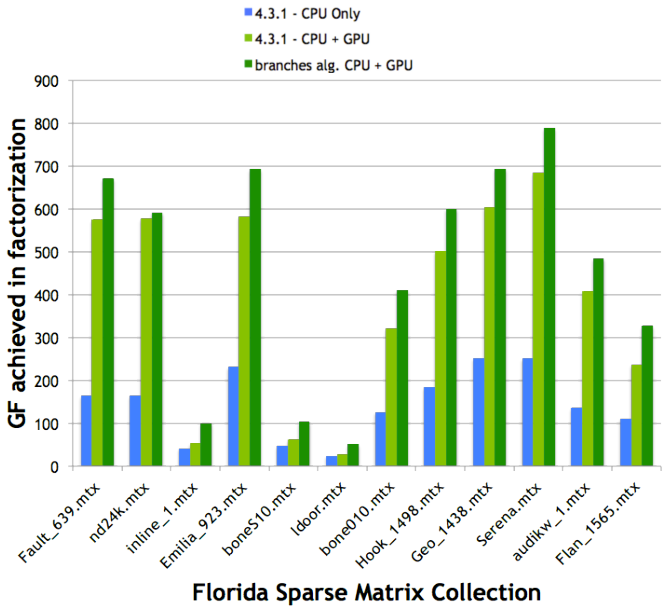

Fig. 12. Comparison of Gflops achieved for the sparse factorization between CHOLMOD 4.3.1 CPU-only (blue), CHOLMOD 4.3.1 CPU+GPU (light green) and CPU+GPU for the current algorithm for the 12 largest real SPD matrices in the Florida Sparse Matrix collection.

VII. Conclusions and Future Work

Previous work has shown that, while it is relatively easy to achieve GPU acceleration for large matrices whose factors contain substantial fill, basic methods are insufficient to achieve compelling speedups when factoring matrices with smaller fill ratios.

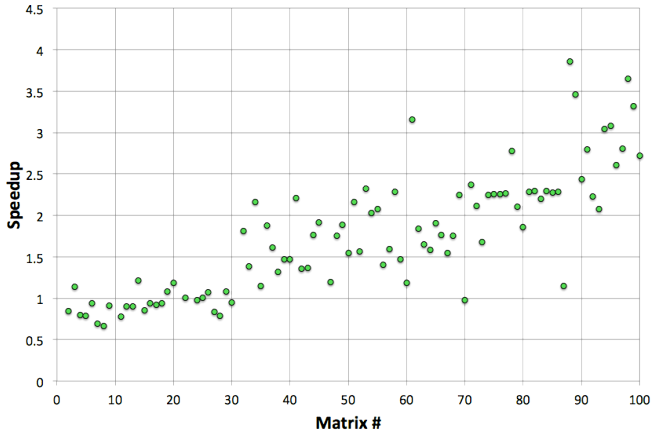

Fig. 13. Speedup achieved using the current algorithm vs. the CPU-only version for the 100 largest real SPD matrices from the Florida Sparse Matrix collection.

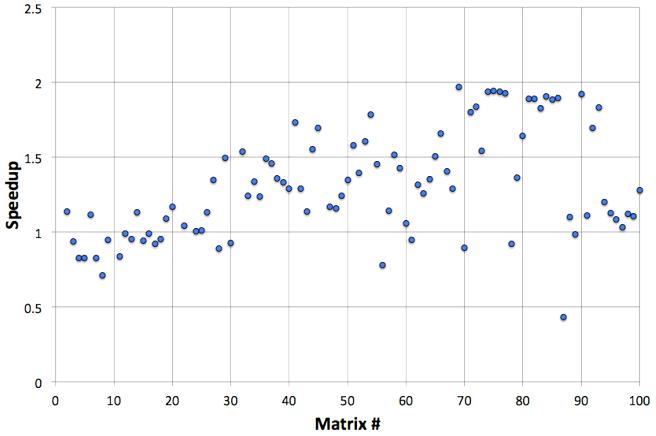

Fig. 14. Speedup achieved using the current algorithm vs. the previous CPU+GPU algorithm (SuiteSparse 4.3.1) for the 100 largest real SPD matrices from the Florida Sparse Matrix collection.

We have proposed that better performance can be achieved by a) streaming branches of the elimination tree through the GPU to accommodate matrices of any size, b) performing the entire factorization of that branch on the GPU to greatly reduce PCIe communication, c) batching collections of small BLAS and LAPACK operations to minimize launch latency and by d) executing multiple such batches concurrently to maximize GPU utilization. A version of CHOLMOD which been modified with the resulting algorithm has been used to demonstrate the performance benefits. For those matrices where the previous CPU+GPU algorithm performed the worst, the current algorithm performs as much as 1.9x better, and 2.2x better than the CPU alone. Even large matrices with relatively large fill ratios showed significant improvements of 1.15x vs. the previous GPU algorithm and 4.1x vs. the CPU alone. An average performance gain of 1.75x vs. the CPU alone was observed for the 94 matrices tested.

This algorithm has been tailored for and performance benchmarked using a left-looking supernodal Cholesky factorization code. However, it is expected that that fundamental optimizations could likely be applied to other factorization methods as well (right-looking/multi-frontal, LU, pivoting, etc.) since they all still involve many small dense BLAS and LAPACK operations. Reducing PCIe communications overhead and improving effective BLAS and LAPACK operation performance should benefit any GPU accelerated sparse direct factorization.

While the performance improvement achieved so far is significant, there is substantially more that can be done. Most particularly, hybrid computing could be enabled to leverage the CPU's computing resources while a branch is being factored on the GPU. That is, the CPU could factor one branch while the GPU is factoring another. In that case the current GPU performance would be almost completely additive to the CPU-only performance meaning that, for matrices with low fill ratios, the speedup vs. the CPU should be closer to 3x (rather than the current 2x).

The other straightforward improvement is to leverage multi-GPU computing. The mechanism of splitting the elimination tree into branches naturally lends itself to sending different branches to different GPUs such that multiple branches are factored simultaneously. For large matrices with many branches, where each branch fills the GPU, we expect this would work well. For smaller matrices, care would need to be taken when creating the branches to trade-off load-balancing (easier with smaller branches) with GPU efficiency (larger branches will lead to larger batches of BLAS operations and greater efficiency on the GPU).

References

[1] N. I. M. Gould, J. A. Scott, and Y. Hu, "A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations," *ACM Trans. Math. Softw.*, vol. 33, no. 2, Jun. 2007. [Online]. Available: http://doi.acm.org/10.1145/1236463.1236465

[2] NVIDIA, "CUDA C programming guide," http://docs.nvidia.com/cuda/-pdf/CUDA_C_Programming_Guide.pdf, 2014.

[3] M. Fatica, "Accelerating linpack with cuda on heterogenous clusters," in *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 46–51. [Online]. Available: http://doi.acm.org/10.1145/1513895.1513901

[4] R. Vuduc, A. Chandramowlishwaran, J. Choi, M. Guney, and A. Shringarpure, "On the limits of gpu acceleration," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, ser. HotPar'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 13–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863086.1863099

[5] T. George, V. Saxena, A. Gupta, A. Singh, and A. R. Choudhury, "Multifrontal Factorization of Sparse SPD Matrices on GPUs," in *IEEE International Parallel & Distributed Processing Symposium*. IEEE, January 2011.

[6] R. Lucas, G. Wagenbreth, J. Tran, and D. Davis, "Multifrontal sparse matrix factorization on graphics processing units," 2012.

[7] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, "Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate," *ACM Trans. Math. Softw.*, vol. 35, no. 3, pp. 22:1–22:14, Oct. 2008. [Online]. Available: http://doi.acm.org/10.1145/1391989.1391995

[8] T. A. Davis, "SuiteSparse," http://www.suitesparse.com, 2014.

[9] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663

[10] Intel, "Intel Parallel Studio XE," https://software.intel.com/en-us/intel-parallel-studio-xe/, 2013.

[11] NVIDIA, "CUDA Toolkit," https://developer.nvidia.com/cuda-toolkit, 2014.
[12] Netlib, "BLAS (Basic Linear Algebra Subprograms)," http://www.-netlib.org/blas.
[13] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "Lapack: A portable linear algebra library for high-performance computers," in *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '90. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 2–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=110382.110385
[14] NVIDIA, "CUBLAS LIBRARY," http://docs.nvidia.com/cuda/pdf/-CUBLAS_Library.pdf, 2014.
[15] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998. [Online]. Available: http://dx.doi.org/10.1137/S1064827595287997
[16] T. A. Davis, *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2006.
[17] G. Krawezik and G. Poole, "Accelerating the ansys direct sparse solver with gpus," 2009.
[18] S. C. Rennich, T. A. Davis, and P. Vandermersch, "GPU Acceleration of Sparse Matrix Factorization in CHOLMOD," in *GPU Technology Conference 2014*. Nvidia Corp., March 2014. [Online]. Available: http://on-demand.gputechconf.com/gtc/2014/presentations/S4201-gpu-acceleration-sparse-matrix-factorization-cholmod.pdf
[19] NVIDIA, "PROFILER USER'S GUIDE," http://docs.nvidia.com/cuda/-pdf/CUDA_Profiler_Users_Guide.pdf, 2014.