Sparse QR Factorization on GPU Architectures

Sencer Nuri Yeralan,
Timothy A. Davis, Fellow, SIAM,
and Sanjay Ranka, Fellow, IEEE and AAAS*

November 29, 2013

Abstract

Sparse matrix factorization involves a mix of regular and irregular computation, which is a particular challenge when trying to obtain high-performance on the highly parallel general-purpose computing cores available on graphics processing units (GPUs). We present a sparse multifrontal QR factorization method that meets this challenge, and is up to ten times faster than a highly optimized method on a multicore CPU. Our method is unique compared with prior methods, since it factorizes many frontal matrices in parallel, and keeps all the data transmitted between frontal matrices on the GPU. A novel *bucket scheduler* algorithm extends the communication-avoiding QR factorization for dense matrices, by exploiting more parallelism and by exploiting the staircase form present in the frontal matrices of a sparse multifrontal method.

*N. Yeralan is currently with Microsoft; this paper is part of his PhD work in the Department of Computer and Information Science and Engineering, University of Florida. Email: syeralan@microsoft.com. T. Davis and S. Ranka are in the Department of Computer and Information Science and Engineering, University of Florida. Email: davis@cise.ufl.edu and ranka@cise.ufl.edu.

1 Introduction

QR factorization is an essential kernel in many problems in computational science. It can be used to find solutions to sparse linear systems, sparse linear least squares problems, eigenvalue problems, rank and null-space determination, and many other mathematical problems in numerical linear algebra [14]. Although QR factorization and other sparse direct methods form the backbone of many applications in computational science, the methods are not keeping pace with advances in heterogeneous computing architectures, in which systems are built with multiple general-purpose cores in the CPU, coupled with one or more General Purpose Graphics Processing Units (GPGPUs) each with hundreds of simple yet fast computational cores. The challenge for computational science is for these algorithms to adapt to this changing landscape.

The computational workflow of sparse QR factorization [1,7] is structured as a tree, where each node is the factorization of a dense submatrix (a *frontal matrix* [13]). The edges represent an irregular data movement in which the results from a child node are assembled into the frontal matrix of the parent. Each child node can be computed independently. An assembly phase after the children are executed precedes the factorization of their parent. In this paper, we present a GPU-efficient algorithm for multifrontal sparse QR factorization that uses this tree structure and relies on a novel pipelined multifrontal algorithm for QR factorization that exploits the architectural features of a GPU. It leverages dense QR factorization at multiple levels of the tree to achieve high performance.

The main contributions of our paper are:

- A novel sparse QR factorization method that exploits the GPU by factorizing multiple frontal matrices at the same time, while keeping all the data on the GPU. The result of one frontal matrix (a contribution block) is assembled into the parent frontal matrix on the GPU, with no data transfer to/from the CPU.

- A novel scheduler algorithm that extends the Communication-Avoiding QR factorization [12], where multiple panels of the matrix can be factorized simultaneously, thereby increasing parallelism and reducing the number of kernel launches in the GPU. The algorithm is flexible in the number of threads/SMs used for concurrently executing multiple dense

QRs (of potentially different sizes). At or near the leaves of the tree, each SM works on its own frontal matrix. Further up the tree, multiple SMs collaborate to factorize a frontal matrix.

- The scheduling algorithm and software does not assume that the entire problem will fit in the memory of a single GPU. Rather, we move subtrees into the GPU, factorize them, and then move the resulting contribution block (of the root of the subtree) and the resulting factor (just R, since we discard Q) out of the GPU. This data movement between the CPU RAM and the GPU RAM is expensive, since it moves across the relatively slow PCI bus. We double-buffer this data movement, so that we can be moving data to/from the GPU for one subtree, while the GPU is working on another.

- For large sparse matrices, the GPU-accelerated algorithm offers up to 11x speedup over CPU-based QR factorization methods. It achieves up to 82 GFlops as compared to a peak of 32 GFlops for the same algorithm on a multicore CPU (two 12-core AMD Opteron$^{\text{TM}}$ 6168 processors and 64 GB of shared memory).

Section 2 presents the background of sparse QR factorization and the GPU computing model. The main components of the parallel QR factorization algorithm are given in Section 3. In Section 4, we compare the performance of our GPU-accelerated sparse QR to Davis' SuiteSparseQR package on a large set of problems from the UF Sparse Matrix Collection [10]. SuiteSparseQR is the sparse QR factorization in MATLAB [7]. It uses LAPACK [2] for panel factorization and block Householder updates, whereas our GPU-accelerated code uses our GPU compute kernels for this update step. Future work in this algorithm is discussed in Section 5. An overview of related work, and a summary of this work, are presented Sections 6 and Sections 7.

2 Preliminaries

An efficient sparse QR factorization is an essential kernel in many problems in computational science. Application areas that can exploit our GPU-enabled parallel sparse QR factorization are manifold. In our widely used and actively growing University of Florida Sparse Matrix Collection [10],

we have problems from structural engineering, computational fluid dynamics, model reduction, electromagnetics, semiconductor devices, thermodynamics, materials, acoustics, computer graphics/vision, robotics/kinematics, optimization, circuit simulation, economic and financial modeling, theoretical and quantum chemistry, chemical process simulation, mathematics and statistics, power networks, social networks, text/document networks, web-hyperlink networks, and many other discretizations, networks, and graphs. Although only some of these domains specifically require QR factorization, most require a sparse direct or iterative solver. We view our QR factorization method as the first of many sparse direct methods for the GPU, since QR factorization is representative of many other sparse direct methods with both irregular coarse-grain parallelism and regular fine-grain parallelism.

In the next section, we briefly describe the multifrontal sparse QR factorization method and explain why we have selected it as our target for a GPU-based method. We then give an overview of the GPU computing landscape, which provides a framework for understanding the challenges we addressed as we developed our algorithm.

2.1 Multifrontal Sparse QR Factorization

2.1.1 Ordering and Analysis phase

The first step in solving a sparse system of equations $Ax = b$ or solving a least squares problem is to permute the matrix A so that the resulting factors have fewer nonzeros than the factors of the unpermuted matrix. This step is NP-hard, but many efficient heuristics are available [6, 8, 9].

The second step is to analyze the matrix to set up the parallel multifrontal numerical factorization. This step finds the elimination tree, the multifrontal assembly tree, the nonzero pattern of the factors, and the sizes of each frontal matrix. In a multifrontal method, the data flows only from child to parent in the tree, which makes the tree suitable for exploiting coarse-grain parallelism, where independent subtrees are handled on widely separated processors. The analysis takes time that is no worse than (nearly) proportional to the number of integers required to represent the nonzero pattern of the factors, plus the number of nonzeros in A. This can be much less than the number of nonzeros in the factors themselves.

The ordering and analysis steps are based on our existing multifrontal sparse QR method (SuiteSparseQR) [7]. The ordering and analysis phase is

very irregular in its computation and is thus best suited to stay on the CPU.

Each node in the tree represents one or more nodes in the column elimination tree. The latter tree is defined purely by the nonzero pattern of R, where the parent of node i is j if $j > i$ is the smallest row index for which r_{ij} is nonzero. There is one node in column elimination tree for each column of A.

A multifrontal assembly tree is obtained by merging nodes in the column elimination tree. A parent j and child $j-1$ are merged if the two corresponding rows of R have identical nonzero pattern (excluding the diagonal entry in the child). In general, this requirement is relaxed, so that a parent and child can be merged if their patterns are similar but not necessarily identical (this is called *relaxed amalgamation* [4]). Figures 1 and 2 gives an example of both trees, and the related A and R matrices. In the figure, the rows of A are sorted according to the column index of the leftmost nonzero in each row, so as to clarify the next step, which is the assembly of rows of A into the frontal matrices. Each `x` is an nonzero in A, each dot is an entry that will become nonzero as the matrix is factorized. Each `r` is a nonzero in R. Each node of the tree is a column of A or row of R, and they are grouped together when adjacent rows of R have the same nonzero pattern.

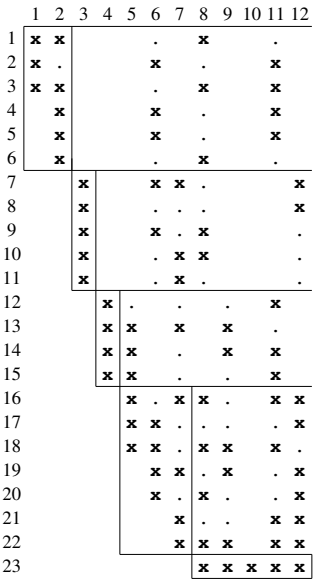

Figure 1: A sparse matrix A.

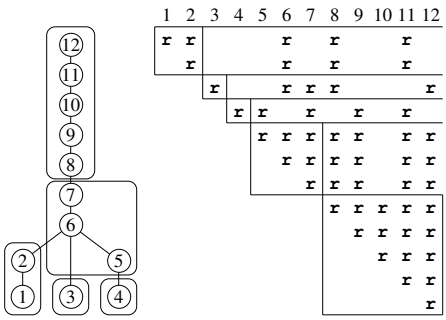

Figure 2: The factor R and its column elimination tree.

In the assembly process, the incoming data for a frontal matrix is concatenated together to construct the frontal matrix. No flops are performed. Each row of A is assembled into a single frontal matrix. If the leftmost nonzero of row i is in column j, then row i is assembled into the frontal matrix containing node j of the elimination tree. Figure 3 illustrates a leaf frontal matrix with no children in the assembly tree. It is the first frontal matrix, and contains nodes 1 and 2 of the column elimination tree. Six rows of A have leftmost nonzeros in columns 1 or 2. These are concatenated together to form a 6-by-5 frontal matrix, held as a 6-by-5 dense matrix. Note that the dimensions of a frontal matrix are typically much less than the dimensions of A itself. The frontal matrix does include some explicit zero entries, in the first column. This is due to the amalgamation of the two nodes into the front.

2.1.2 Factorization phase

This is where the bulk of the floating-point operations are performed (the remainder are done in the next step, the solve phase). All of the flops are computed within small dense frontal matrices (small relative to the dimensions of A, to be precise; the frontal matrices can be quite large). These computations are very regular, very compute-intensive (relative to the memory traffic requirements), and thus well-suited to be executed on one or more GPUs. Continuing the example in Figure 2, after the 6 rows of A are assembled into the front, we compute its QR factorization, reflected in the matrix on the right side of Figure 3. Each `r` is a nonzero in R, each `h` is a nonzero Householder coefficient, and each `c` is an entry in the contribution block.

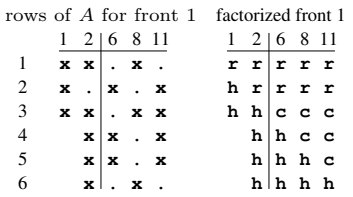

Figure 3: Assembly and factorization of a leaf.

Figure 4 illustrates what happens in the factorization of a frontal matrix that is not a leaf in the tree. Three prior contribution blocks are concatenated and interleaved together, along with all rows of A whose leftmost nonzero falls in columns 5, 6, or 7 (the fully-assembled columns of this front). The QR factorization of this rectangular matrix is then computed.

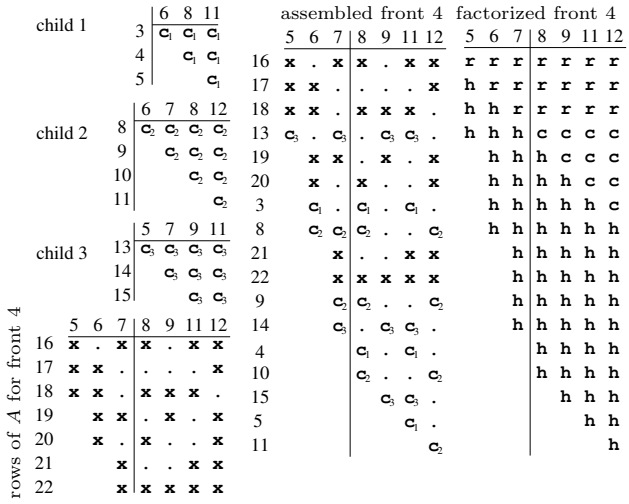

Figure 4: Assembly and factorization of a front with children.

Computation across multiple CPU cores and multiple GPUs can be obtained by splitting the tree in a coarse-grain fashion. At the root of each subtree, a single contribution block would need to be sent, or distributed, to the CPU/GPU cores that handle the parent node of the tree. Our current method exploits only a single GPU, but to handle very large problems, it splits the trees into subtrees that fit in the global memory of the GPU.

2.1.3 Solve phase

Once the system is factorized, the factors typically need to be used to solve a linear system or least-squares problem. These computations are regular in nature, but they perform a number of flops proportional to the number of nonzeros in the factors. The ratio of flop count per memory reference is quite low. Thus, we perform this computation on the CPU, and leave a GPU implementation for future work.

2.2 GPU architecture

A GPU consists of a set of SMs (Streaming Multiprocessors), each with a set of cores that operate in lock-step manner. The *shared memory* available on each SM can be accessed by all cores in the SM, but it is very limited (32K to 64K bytes) and must be shared among multiple blocks of threads. Minimizing the memory traffic between slow GPU global memory and very fast shared memory is crucial to achieve high performance. Dense QR factorization has a very good compute-to-data-movement ratio and can achieve high performance even under these limitations.

The GPU executes one or more *kernels*, which are launched by the CPU. When launching a kernel, the programmer specifies a number of *thread blocks* and the number of threads per block the GPU should commit to the kernel launch. These kernel launch parameters describe how the work is intended to be divided by the GPU among its SMs, and GPUs have varying upper bounds on the allowed parameter values. Regardless of the number of available SMs or cores per SM for a particular GPU, the GPU's scheduler assigns thread blocks to SMs and executes the kernel until the thread block completes its execution. The GPU scheduler organizes threads into collections of 32, called a *warp,* and threads constituting a warp execute code in a Single Program Multiple Data (SPMD) fashion within an SM [18].

Each thread on the GPU has access to a small number of registers, which cannot be shared with any other thread (new GPUs allow for some sharing of register data amongst the threads in a single warp, but our current algorithm does not exploit this feature). The GPU in our experiments, an NVIDIA Tesla C2070, provides up to 31 double-precision floating-point registers for each thread.

Each SM has a small amount of *shared memory* that can be accessed and shared by all threads on the SM, but which is not accessible to other SMs.

there is no cache coherency across multiple SMs. The shared memory It is arranged in banks, and bank conflicts occur if multiple threads attempt to access different entries in the same bank at the same time. Matrices are padded to avoid bank conflicts, so that a row-major matrix of size m-by-n is held in an m-by-$(n+1)$ array. Accessing of shared memory can be done in randomly-accessed order without penalty, so long as bank conflict is avoided.

Global memory on the GPU is large, but its bandwidth is much smaller than the bandwidth of shared memory, and the latency is higher. Global memory can be read by all SMs, but must be read with stride-one access for best performance (a *coalesced* memory transaction).

The GPU provides hardware support for fast warp-level context switching on an SM, and the GPU scheduler attempts to hide memory latency by overlaping global memory transactions with computation by switching between warps. While a memory transaction for one warp is pending, the SM executes another warp whose memory transactions are ready.

All three layers of memory (global, shared, and register) must be explictly managed for best performance, with multiple memory transactions between each layer "in flight" at the same time, with many warps, so that computation can proceed in one warp while another warp is waiting for its memory transaction to complete.

The NVIDIA Tesla C2070 (Fermi) has 448 double-precision floating point cores. It operates at up to 515 GFlops, and twice that speed in single-precision. The 448 cores are partitioned into 14 SMs. A single SM of 32 cores has access to 64KB of shared RAM, typically configured as 16K of L1 cache and 48K of addressable shared-memory for sharing data between threads in a block. The SM has 32K registers, partitioned amongst the threads. All 14 SMs share an L2 cache of 768KB. Sharing between SMs is done via the 6GB of global shared memory. Up to 16 kernels can be active concurrently on the Fermi architecture, on different SMs. The cost of a GPU is much less than a multicore CPU when considering its performance ($ per GFlop), and its power consumption is much less as well. This provides a strong rationale for the development of algorithms that exploit general-purpose GPU computing.

3 Parallel Algorithm

The computational workflow of QR factorization is structured as a tree, where each node is the factorization of a dense submatrix. The edges represent an

irregular data movement in which the results from a child node are assembled into the frontal matrix of the parent. Each child node can be computed independently. However, an assembly phase after the children are executed precedes the factorization of their parent. Our algorithm is flexible in the number of threads/SMs used for concurrently executing multiple dense QRs (of potentially different sizes). At or near the leaves of the tree, each SM in a GPU works on its own frontal matrix. Further up the tree, multiple SMs collaborate to factorize a frontal matrix.

The scheduling algorithm does not assume that the entire problem will fit in the memory of a single GPU. Rather the algorithm moves subtrees to the GPU, factorizes them, and moves the resulting contribution blocks and their resulting R factors off of the GPU. This data movement between CPU RAM and GPU RAM is expensive, since it moves across the relatively slow PCI bus. We double-buffer this data movement, so that we can be moving data to/from the GPU for one subtree, while the GPU is working on another.

Although hardware instructions are provided for atomic operations and intra-SM thread synchronization, GPU devices offer poor support for inter-SM synchronization. Our execution model uses a master-slave paradigm where the CPU is responsible for building a list of tasks, sending the list to the GPU, synchronizing the device, and launching the kernel. Since a GPU has poor inter-SM synchronization, we construct the set of tasks so that they have no dependencies between them at all. The GPU receives the list of tasks from the CPU for each kernel launch, performing operations described by each task. The kernel implementation is monolithic, inspecting the task descriptor to execute the appropriate device function. In this manner, a single kernel launch simultaneously computes the results for tasks in many different stages of the factorization pipeline. The CPU arranges tasks such that there are no data dependencies within the task list for a particular kernel launch. This software design pattern is called the überkernel [20].

Thus factorization of the matrix may take several kernel launches to complete. We launch each kernel asynchronously using the NVIDIA CUDA events and streams model. While one kernel is executing within a CUDA stream, the CPU builds the list of tasks for the next kernel launch. We use another stream to send the next list of tasks asynchronously. The CPU is responsible for synchronizing the device prior to launching the next kernel in order to ensure that the task data has arrived and that the previous kernel launch has completed.

The details of the algorithm are presented in the next several subsections.

3.1 Dense QR Scheduler (the Bucket Scheduler)

Our CPU-based Dense QR Scheduler is comprised of a data structure representing the factorization state of the matrix together with an algorithm for scheduling tasks to be performed by the GPU. We call this algorithm and its data structure the *bucket scheduler.*

The algorithm partitions the input matrix into 32-by-32 submatrices called *tiles*. The choice of a tile size reflects the thread geometry of the GPU and the amount of shared memory that each SM can access.

All tiles in a single row are called *row tiles*, so that a single row tile in a 256-by-160 matrix consists of a submatrix of size 32-by-160. In our scheduler, we refer to a row tile by a single integer, its row tile index. In contrast, a *column tile* is just a single tile, so one row tile in a 256-by-160 matrix consists of a 32-by-160 submatrix, containing 5 column tiles. The *leftmost column tile* in a row tile refers to the nonzero column tile with the least column tile index. In a row tile, all column tiles to the left of the leftmost column tile are all zero. Each row tile has a flag indicating whether or not its leftmost column tile is in upper triangular form. The goal of the QR factorization is to reduce the matrix so that the kth row tile has a leftmost column tile k in upper triangular form.

All 32 rows in a row tile are contiguous. A set of two or more row tiles with the same leftmost column tile can be placed in a *bundle*, where the row tiles in a bundle need not be contiguous.

We place row tiles into *column buckets,* where row tile i with leftmost column tile j is placed into column bucket j. During factorization, row tiles move from their initial positions in the column buckets to the right until each column bucket contains exactly one row tile with its flag set to indicate that it is upper triangular. Figure 5 shows a 256-by-160 matrix and its corresponding buckets after initialization. The matrix has two row tiles that are all zero (tiles (7,1) and (8,1)).

The CPU is responsible for manipulating row tiles within bundles, filling a queue of work for the GPU to perform, and advancing row tiles across column buckets until exactly one row tile remains in each column bucket. Each round of factorization builds a set of tasks by iterating over the column buckets and symbolically manipulating their constituent row tiles.

All row tiles in a bundle have the same leftmost column tile, prior to factorization of the bundle. After factorization, the leftmost column tile of the topmost row tile is placed into upper triangular form, and the leftmost

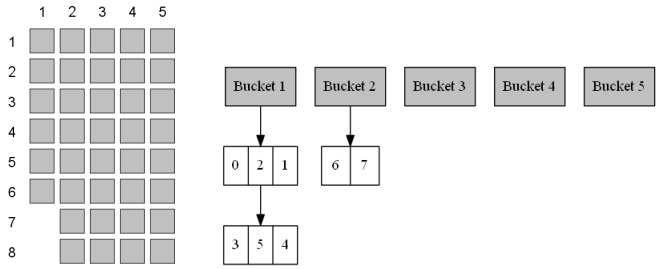

Figure 5: A 256-by-160 matrix and its bucket scheduler.

column tile of the remaining row tiles are all zeroed out. The bundle now represents a set of row tiles to which a block Householder update must be allied, to all column tiles to the right of the leftmost column tile of the bundle.

We iterate over the column buckets and for each column bucket, we perform the operations described below, building a set of tasks to be executed by the GPU at each kernel launch, illustrated in Figure 6. Each image in the figure represents the tasks at each kernel launch and is color-coded by bundle. Gray tiles are unmodified by their respective kernel launches. White tiles are finished.

1. Generate bundles and build block Householder update tasks on the CPU: Row tiles that are unassociated with a bundle become new bundles ready for factorization. Figure 6b illustrates this by grouping tiles into bundles of size 3 (red), 3 (magenta), and 2 (green). *Factorize* tasks are created for such bundles.

2. Launch the kernel with its current set of non-uniform tasks: Launch the GPU kernel and perform any queued Factorize and Apply tasks. Factorize tasks factorize the leftmost column tile in each bundle, and Apply tasks apply a block Householder update from a prior Factorize task in the previous kernel launch.

3. Advance the bundles on the CPU: Next we advance the bundles, leaving the topmost row tile in upper triangular form, as shown in Figure 6c for column bucket 1. These advancing bundles move to the next column bucket and represent a pending block Householder update from the previous factorization step.

We continue performing the operations described above until only one upper triangular row tile appears in each column bucket.

The bucket scheduler can further exploit parallelism and decrease the number of idle tiles (shown as gray in Figure 6), by following a block Householder update with an immediate factorization of the same bundle. In a modification of the procedure described above, we regard advancing bundles to be candidates for *pipelining*. This pipelining approach occurs in two different scenarios.

1. The first scenario involves adding idle tiles to preexisting bundles. A row tile may become idle if its bundle has just been factorized and it is the only member of its bundle following bundle advancement. The kernel launch between Figure 6b and Figure 6c leaves tile (7,2) upper triangular and idle following bundle advancement (green bundle). Instead of leaving the tile idle, a new bundle could be formed, consisting of tiles (5,2), (6,2), and (7,2). This new bundle could be factorized immediately after the block Householder update of the magenta bundle (row tiles 4, 5, and 6 in Figure 6b. We call this strategy *bundle growth*, and we call the set of newly added idle row tiles the bundle's *delta*. Although the bundle delta does not participate in its host bundle's block Householder update, it does participate in a subsequent factorization of the host bundle. In the example discussed above, tile (7,2) would be the new bundle's delta. This blue bundle can then be be pipelined with the preexisiting magenta bundle, where the magenta task that applies a Householder update to tiles (4,2),(5,2) and (6,2) can immediately follow this with the factorizations of the blue bundle, instead of waiting until the next kernel launch. Comparing Figure 6c and Figure 7c illustrates the addition of this tile into the blue bundle in Figure 7c. With pipelining, the blue bundle is factorized in the second kernel launch (Figure 6c), rather than waiting until the third kernel launch (Figure 7d) when pipelining is not exploited.

2. The second scenario involves bundles that do not undergo bundle growth and are scheduled only for block Householder updates (the red bundle in Figure 6c, for example). When such bundles perform their block Householder applies, we can pipeline the factorization of the bundle's leftmost column tile. Comparing Figure 6c to Figure 7c demonstrates that a new yellow bundle has been created, representing the pipelined task of performing the red bundle's block Householder update followed by a fresh

factorization of the tiles contained therein. We call tasks representing this pipelined approach, *Apply/Factorize* tasks. With pipelining, the yellow bundle with tiles (2,2) and (3,2) is factorized in the second kernel launch (Figure 6c), rather than waiting until the third kernel launch (Figure 7d) when pipelining is not exploited. Without pipelining, the bundle performs its update and factorize on the next kernel launch, resulting in a portion of the matrix remaining idle during the next kernel launch. For example, in Figure 6d, row tiles 2, 3, 5, 6, and 7 are mostly gray, except for a factorization of their leftmost column tiles. By comparison, these same row tiles are active in Fiugre 7d.

3.2 Computational Kernels on the GPU

In this section, we provide details of the key computational kernels required for the simultaneous dense QR factorization of multiple frontal matrices on the GPU.

The Factorize task factorizes the leading tiles of a bundle, producing a block Householder update (the V and T matrices) and an upper triangular factor R in the top row tile. The Apply task uses the V and T matrices to apply the block Householder update to the remaining column tiles, to the right in this bundle.

Our tile size is selected to be a 32-by-32 submatrix, so that the row and column dimensions match the size of a warp (32 threads). Six tiles can fit exactly into the 48K of shared RAM available each SM, but with padding this drops to five tiles. The V matrix is lower trapezoidal and T is a single upper triangular tile, so three tiles are set aside for V and T (plus one row). Three tiles are used for V, and the upper triangular part of its topmost tile (where V is lower triangular) holds the matrix T. Two tiles hold a temporary matrix C.

3.2.1 Factorize kernel

The Factorize task requires a description of the bundle, the column tile, and a memory address to store T. The GPU factorizes the leftmost column tile of the bundle into upper triangular form (called A in this task) and overwrites it with the Householder vectors, V, and the upper triangular T matrix, which are then written back into GPU global memory. Saving V and T is necessary because on the next GPU kernel launch, they are involved

in the block Householder application across the remaining columns in this bundle. The lower triangular portion of the topmost tile of V is stored together with T. Because both V and T contain diagonal values, the resulting memory space is a 33-by-32 tile with V offset by 1. We call this combined structure a *VT tile.* This in turn leaves the first tile in the bundle upper triangular, and the other tiles in the bundle contain V.

This description assumes a bundle of three tiles and a kernel launch with 384 threads per task, but we have other kernels for different bundle sizes.

1. **Load A from global memory**. All 384 threads in the task cooperate to load the bundle's tiles from global memory (the A matrix of size m-by-n) into a single shared memory array of size m-by-$(n+1)$, where $m = 96$ and $n = 32$. No computation is performed while A is being loaded. After A is loaded, each thread loads into register 8 entries of A along a single column for which it is responsible. It keeps these entries in register for the entire factorization. We call this 8-by-1 submatrix operated on by a single thread a *bitty block.*

2. **Compute σ for the first column**, where $\sigma = \sum(A_{2:m,1})^2$ (where $2:m$ denotes $2...m$). This computation is composed of two reduction operations. First, the 12 threads responsible for the first column of A compute the sum of squares using an 8-way fused multiply-add reduction in register memory, saving the final result into shared memory. Threads in the thread block synchronize to ensure they have all finished the first phase before entering the second phase of the reduction. We designate the first thread in the thread block to be the *master thread.* The master thread completes the computation with a 12-way summation reduction reading from shared memory into register memory. The master thread retains σ in register during the factorization loop.

3. **The main factorization loop** iterates over the columns of A, performing the following operations (a) through (f) in sequence.

   (a) **Write the k^{th} column of A back into shared memory.** The threads responsible for the k^{th} column write their A values from register back into shared memory. The SM then synchronizes all 384 threads before proceeding, maintaining memory consistency. Note that once the diagonal value is computed, the k^{th} column

becomes the k^{th} Householder vector. Additionally, values above the diagonal are the k^{th} column of R.

(b) **Compute the k^{th} diagonal.** The master thread is responsible for computing the k^{th} diagonal entry of R, the k^{th} diagonal entry of V, and τ: $s = \sqrt{{a_{kk}}^2 + \sigma}$, $v_{kk} = a_{kk} - s$ if $a_{kk} \leq 0$ and $-\sigma/a_{kk} + s$ otherwise, and $\tau = -1/sv_{kk}$. If σ is very small, the square root is skipped, and v_{kk} along with τ are set to 0. Because all threads will need v_{kk} and τ, the remaining threads synchronize with the master thread.

(c) **Compute an intermediate z vector.** All threads cooperate to perform a matrix-vector multiply $z = -\tau v^{\mathsf{T}} A_{k:m,1:n}$, which is used to compute the V and T matrices, where v is the Householder vector, held in $A_{k:m,k}$.

To compute z, each thread loads the entries of the v vector it requires from shared memory into register memory, from $A_{k:m,k}$. The thread's bitty block of A is already in register (the 8 entries of A that the thread operates on).

The calculation is done in two parts. First, each thread performs an 8-way partial dot product reduction using fused multiply-adds in register memory. The threads store the partial result in a 12 by 32 region of shared memory. The threads synchronize to guarantee that they have completed the operation before proceeding with the second phase of the calculation. The second phase of the calculation involves only a single warp. This warp performs the final 12-way summation reduction into shared memory, completing the calculation of z.

(d) **Update A in register memory.** All threads responsible for columns to the right of the k^{th} column of A participate in updating A by summing values with the outer product $A_{k:m,k+1:n} = A_{k:m,k+1:n} + vz_{k+1:n}$. Threads involved in the outer product computation already have v in register memory, and they need only load z from shared memory once to update their values of A. Each thread needs to load a single value of z, since each bitty block is 8-by-1, in a single column of A.

(e) **Compute the next σ value.** Some threads participating in updating A in register memory may also begin to compute the

next σ value if they are responsible for the $(k+1)^{st}$ column of A. These threads participate in computing σ, and the process is the same as the computation of σ for the first column.

(f) **Construct the k^{th} column of T,** with $T_{1:k-1,k} = T_{1:k-1,1:k-1} z_{1:k-1}{}^{\mathsf{T}}$, a matrix-vector multiply. Threads 1 to $k-1$ are assigned to compute the kth column of T, where the ith thread performs the inner product to compute t_{ik}. Threads load values of T and z from shared memory, accumulating the result in register memory. Finally, the participating threads each write their scalar result t_{ik} from register memory into shared memory. The master thread writes $t_{kk} = \tau$.

4. **Store A, V, and T back into global memory**. All 384 threads in the task cooperate to store the bundle's tiles back into global memory. Since the first tile of V and T are held in a single VT tile in global memory, they are stored together to maintain coalesced global memory transactions.

   Once the VT tile is stored, the three tiles that hold A are stored back into global memory in the frontal matrix being factorized. The first tile of A is the upper triangular matrix R, and the remaining tiles are the second and third tiles of the Householder vectors, V.

Following a Factorize task, the corresponding bundle's topmost tile contains R. The remaining leftmost column tiles contain V, which is used in the subsequent block Householder applies. The first tile of V (which is lower triangular) is stored together with the upper triangular T matrix in a separate 33-by-32 global memory space (the VT tile), since the top left tile in the frontal matrix now holds R. The VT tile remains only until the next kernel launch, when the block Householder update is applied to the column tiles to the left, in this bundle. At that point, the space is freed to hold another VT tile, from another bundle in this frontal matrix or in another one being factorized at the same time.

3.2.2 Apply kernel

Each Apply task involves a bundle, an originating column tile, a column tile range, and the location of the VT tile. The GPU loads the VT tile and

iterates over the column tile range performing the block Householder update, (1) $C = V^\mathsf{T} A$, (2) $C = T^\mathsf{T} C$, and (3) $A = A - VC$.

Since the V and T matrices are used repeatedly, and since V is accessed both by row and column order (V and V^T), they are loaded from global memory by the SM and held in shared memory until the Apply task completes. The temporary C matrix is also held in shared memory or register. The A matrix remains only in global memory, and is staged into a shared memory buffer and then into register, one chunk at a time. The algorithm is as follows:

- **Load V and T.** All 384 threads in the task cooperate to load the V and T matrices from global memory into a single shared memory array of size 97-by-32. Since the first tile of V and T are held in a single VT tile in global memory, they are loaded together to maintain coalesced global memory accesses. No computation is performed while V and T are being loaded.

- **Apply the block Householder:** $A = A - VT^\mathsf{T} V^\mathsf{T} A$. The A matrix is 3-by-t tiles in size in global memory, and represents a portion of the frontal matrix being factorized. Since V and T take up three tiles of shared memory, two tiles remain for a temporary matrix (C) required to apply the block Householder update. Registers also limit the size of C and the submatrix of A that can be operated on. If held in register, each thread can operate on at most a 4-by-4 submatrix of A or C (its bitty block). With 384 threads, this results in a submatrix of A of size 96-by-64, or 3-by-2 tiles. The same column dimension governs the size of C, which is 2-by-1 tiles in size.

  Thus, the t column tiles of A are updated two at a time. Henceforth, to simplify the discussion, A refers to the 96-by-64 submatrix (3-by-2 tiles) updated in each iteration across the t column tiles. The block update is computed in three phases as (1) $C = V^\mathsf{T} A$, (2) $C = T^\mathsf{T} C$ and (3) $A = A - VC$, as follows:

  1. **Load A and compute $C = V^\mathsf{T} A$.** This work is done in steps of 16 rows each (a halftile), in a pipelined manner, where data for the next halftile is loaded from global memory into shared, while the current halftile is being computed. This enables the memory / computation overlap required for best performance.

The C matrix is held in register, so the 2 tiles of shared memory (for C) are used to buffer the A matrix. This 32-by-64 matrix is split into two buffers B_0 and B_1, each of size 16-by-64 (two halftiles).

All threads prefetch the first halftile ($p = 0$) into register, which is the topmost 16-by-64 submatrix of A. This starts the pipeline going. Next, $C = V^\mathsf{T}A$ is computed across six halftiles, one halftile (p) at a time:

for $p = 1$ **to** 6
    a. Write this halftile (p) of A from register
        into shared buffer $B_{p \bmod 2}$.
    b. syncthreads.
    c. Prefetch the next halftile ($p + 1$) of A
        from global to register.
    d. Compute $C = V^\mathsf{T}A$, where A is in
        buffer $B_{p \bmod 2}$.
end for

In step (b), all threads must wait until all threads reach this step, since there is a dependency between steps (a) and (d). However, steps (c) and (d) can occur simultaneously since they operate on different halftiles. In step (d), each thread computes a 4-by-2 bitty block of C, held in register for phases 1 and 2 (only the first 256 threads do step (d); the other 128 threads remain idle and are only used for memory transactions in this phase).

The global memory transactions for a warp are scheduled in step (c), but the warp does not need to wait for them to be completed before computing step (d) (they are not needed until step (d) of iteration $p + 1$). Likewise, no synchronization is required between step (d) of iteration p and step (a) of the next iteration $p + 1$.

Since steps (c) and (d) (for iteration p) can overlap with step (a) (for iteration $p + 1$), this algorithm keeps all parts of the SM busy at the same time: computation (step (d)), global memory (step (c)), and shared memory (steps (a) and (d)).

2. **Compute $C = T^\mathsf{T}C$.** All matrices are now in shared memory. Each thread operates on the same 4-by-2 bitty block of C it op-

erated on in phase 1, above, and now writes its bitty block into the two tiles of shared memory. These are no longer needed for the buffer B, but now hold C instead. Only the first 256 threads take part in this computation.

3. **Compute $A = A - VC$, where V and C are in shared memory but A remains in global**. The A matrix had already been loaded in from global memory once, in phase 1, but it was discarded since the limited shared memory is already exhausted by holding V, T and the C/B buffer. Each of the 384 threads updates a 4-by-4 bitty block of A.

The layout of the bitty blocks of A and C is an essential component to the algorithm. Proper design of the bitty blocks avoids bank conflicts and ensures that A is accessed with coalesced global memory accesses. Both A and C bitty blocks are spread across the matrices. They are not contiguous submatrices of A and C. The C matrix is 32-by-64 and is operated on by threads 0 to 255. Using 0-based notation, the 4-by-2 bitty block for thread i is defined as

$$C_{[i]} = \begin{bmatrix} c_{(i \bmod 8),(\lfloor i/8 \rfloor)} & c_{(i \bmod 8),(32+\lfloor i/8 \rfloor)} \\ c_{(8+i \bmod 8),(\lfloor i/8 \rfloor)} & c_{(8+i \bmod 8),(32+\lfloor i/8 \rfloor)} \\ c_{(16+i \bmod 8),(\lfloor i/8 \rfloor)} & c_{(16+i \bmod 8),(32+\lfloor i/8 \rfloor)} \\ c_{(24+i \bmod 8),(\lfloor i/8 \rfloor)} & c_{(24+i \bmod 8),(32+\lfloor i/8 \rfloor)} \end{bmatrix}$$

where $c_{0,0}$ is the top left entry of C. For example, the bitty blocks of threads 0 and 1 are, respectively:

$$C_{[0]} = \begin{bmatrix} c_{0,0} & c_{0,32} \\ c_{8,0} & c_{8,32} \\ c_{16,0} & c_{16,32} \\ c_{24,0} & c_{24,32} \end{bmatrix}, \quad C_{[1]} = \begin{bmatrix} c_{1,0} & c_{1,32} \\ c_{9,0} & c_{9,32} \\ c_{17,0} & c_{17,32} \\ c_{25,0} & c_{25,32} \end{bmatrix}$$

The 4-by-4 bitty block of A for thread i is defined very differently than the C bitty block, where $A_{[i]} =$

$$\begin{bmatrix} a_{(\lfloor i/16 \rfloor),(i \bmod 16)} & \cdots & a_{(\lfloor i/16 \rfloor),(48+i \bmod 16)} \\ a_{(24+\lfloor i/16 \rfloor),(i \bmod 16)} & \cdots & a_{(24+\lfloor i/16 \rfloor),(48+i \bmod 16)} \\ a_{(48+\lfloor i/16 \rfloor),(i \bmod 16)} & \cdots & a_{(48+\lfloor i/16 \rfloor),(48+i \bmod 16)} \\ a_{(72+\lfloor i/16 \rfloor),(i \bmod 16)} & \cdots & a_{(72+\lfloor i/16 \rfloor),(48+i \bmod 16)} \end{bmatrix}$$

so that thread 0 owns $a_{0,0}$ and thread 1 owns $a_{0,1}$. When used in our algorithm, these layouts of the C and A bitty blocks ensure that all global memory accesses are coalesced, that no memory bank conflicts occur, and that no significant register spilling occurs in our kernels.

With a 4-by-4 bitty block for A, each thread loads in 8 values from shared memory (a 4-by-1 column vector of V and a 1-by-4 row vector of C), and then performs 32 floating point operations (a rank 1 outer product update of its 4-by-4 bitty block). This gives a flops per memory transfer ratio of 4, which is essential because the floating point units for this particular GPU are 4 times faster than register bandwidth.

The 4-by-2 bitty block for C requires 6 loads for 16 operations, a ratio of $16/6 = 2.67$. Since this is less than 4, it is sub-optimal, but unavoidable in the context of the entire block Householder update.

3.2.3 Apply/Factorize kernel

In an effort to reduce global memory traffic on the GPU, the Apply/Factorize pipelined task attempts to avoid superfluous global memory loads and stores for tiles of the matrix modified in both the Apply segment and the Factorize segment. For example, the last step of the Apply task performs a read-modify-write operation of the A matrix in global memory. However, for the Apply/Factorize task, A may instead be read from global memory, modified, and stored into shared memory, priming the immediate Factorize. This modification saves two global memory operations.

Because the completion of a kernel launch synchronizes the device, we regard it as an expensive barrier synchronization. The completion of a kernel launch wipes data from shared memory, and the execution of our dense QR kernels ceases. When the bucket scheduler adds Apply/Factorize tasks to the GPU work list, it reduces the number of kernel launches (i.e. barriers) with the goal that reducing the number of kernel launches increases GPU occupancy and throughput. We discuss the performance impact of the Apply/Factorize pipelined tasks in Section 4.

3.3 Sparse QR Scheduler

The CPU-based Sparse QR Scheduler represents the factorization state for each dense front using a finite state machine, and it uses the Bucket Sched-

uler for the simultaneous factorization of each those dense fronts. In other words, we have many bucket schedulers active at the same time. The Sparse QR Scheduler manages both assembly and factorization kernel launches, coalescing the schedules of tasks from many assembly operations and many dense QR bucket schedulers into a single kernel launch.

Fronts that are leaves in the assembly tree have no children and are activated for factorization first, as illustrated in Figure 8. In the figure, arrows point in the direction of contribution block data flow from child to parent. Fronts with no children have been activated and performing *S-Assembly,* as identified in light blue leaves. The size of each node reflects the size of the corresponding frontal matrix.

The scheduler builds *S-Assembly* tasks for each front. Once values from the input problem are in place within the dense front, the front must now wait for contribution blocks from its children to be assembled into it. Once every child of a frontal matrix completes, the scheduler advances that front into factorization. The Bucket Scheduler is invoked to factorize the dense matrix. Once dense factorization of the front completes, its rows of the result, the R factor, are ready to be transferred off the GPU. Further, its contribution block rows are ready to be assembled into its parent front. The scheduler builds *Pack Assembly* tasks to perform this operation. A front is finished when its rows of R are transferred off the GPU and its contribution block rows have been assembled into its parent.

We build tasks and execute kernels using a strategy similar to the Bucket Scheduler. Using CUDA events and streams, the Sparse QR Scheduler builds a list of tasks to be completed by a kernel while the previous kernel executes on the GPU. This strategy affords us additional benefits. We are able to hide the latency of memory traffic between the GPU device and the CPU host. We perform a transfer of the R factor in a non-blocking fashion by initiating an asynchronous memory transfer on a CUDA stream and marking an event to record when the transfer completes. Furthermore, the R factor may become available before factorization completes. This occurs when the remaining factorization tasks involve only contribution block rows.

3.3.1 Assembly Kernels

In addition to the compute kernels used in the dense QR factorization, sparse QR factorization employs two kernels responsible for data movement:

- **S-Assembly** refers to scattering values from the permuted sparse input matrix, S, into the dense frontal matrices residing on the GPU. The CPU packs all S entries for fronts within a stage into a list of index-value tuples, and describes to the GPU where each front can find its S entries. The value is copied within global memory to a frontal matrix at the location referred to by the index field of the tuple. The data movement is embarrassingly parallel since multifrontal QR factorization relies on concatenation of the children contribution blocks. This is in contrast to multifrontal LU or Cholesky factorization, where the contribution blocks of multiple children must be summed, not concatenated.

  We select a granularity with which to build S-Assembly tasks. In our implementation, each thread is responsible for moving 4 values into position. S-Assembly may occur concurrently with children pushing their contribution blocks into the front.

- **Pack Assembly** refers to scattering values from a front's contribution block into its parent. The CPU builds and sends two maps to the GPU that describe the correspondence between a front's row and column indices to its parent's row and column indices. We call these two maps *Rimap* and *Rjmap,* respectively. When a front completes its factorization step, the values in its contribution block are copied into its parent front. The CPU describes to the GPU where the front's contribution block begins, where its parent resides in GPU memory, the number of values to copy, and the location of Rimap and Rjmap. The GPU reads Rimap and Rjmap into shared memory and uses shared memory as a cache for fast index translations. The data movement is embarrassingly parallel as with S-Assembly, and we select a granularity that best suits GPU shared memory limits per streaming multiprocessor. We select a maximum Pack Assembly tile size of 2048 entries of Rimap and Rjmap.

3.4 Staging for Large Trees

During symbolic analysis, the CPU may discover that the amount of memory required to store the frontal matrices and assembly data on the GPU exceeds the total amount of memory available to the device. When this occurs, we switch to a strategy where we divide the assembly tree and perform the factorization in stages.

During symbolic analysis we compute a postordering of the fronts. We keep a list of stages to be executed by the GPU. Each entry in the staging list is an index into the postordered list. As we iterate over the postordering, we keep a running summation of the memory required by each front. The memory required by each front in a stage is the summation of the number of entries in the front, the entries of its children, and number of entries in the original sparse input matrix that are to be assembled into the front. As we traverse the fronts in this postordered manner, a new stage is created when the next front would exceed the memory limitation of the GPU.

Executing a staged sparse factorization uses the CPU-based Sparse QR Scheduler for each front in the stage. We transfer relevant values from the original input problem and assembly mappings and allocate space on the GPU for each front participating in the stage. We then invoke the Sparse QR Scheduler, and we flag fronts whose parents are in subsequent stages, signalling to the Sparse QR Scheduler to bypass the Pack Assembly phase. Such fronts are roots of the subtrees in Figure 9.

When crossing staging boundaries, the contribution block must be marshalled into the next stage. We perform this marshalling at the end of a stage when pulling rows of R from the GPU. In addition to the rows of R, we also pull the contribution block rows into a temporary location in CPU memory. As we build the data for the next stage, we send the contribution block back to the GPU. When invoking the Sparse QR Scheduler for the next stage, we flag fronts whose only data is contribution blocks, and those fronts begin factorization at the Pack Assembly phase, as illustrated in Figure 10. In the figure, fronts with no children have been activated and performing either Pack Assembly if the front was in stage 1 or S-Assembly, if it is new to this stage. Children performing pack assembly are identified as yellow leaves, and children performing S assembly are identified as light blue leaves.

4 Experimental Results

Experimental results were obtained on a single shared-memory system equipped with two 12-core AMD Opteron^{TM} 6168 processors, 64 GB of shared memory, and an NVIDIA Tesla C2070 with 14 SMs each with 32 cores. The Tesla C2070 has a total of 6 GB memory, 4 GB of which is available as global device memory, 2 GB as texture memory.

We measured the performance of each of our compute kernels individually.

Apply tasks are able to achieve up to 183.3 GFlops. Factorize tasks are able to achieve up to 23.62 GFlops. When a frontal matrix is small enough that it can be factorized by a single task, the VT tile need not be computed. In this case, the factorize tasks are able to achieve up to 34.80 GFlops on a 72-by-64 problem. Factorize tasks suffer from a hefty initial serial fraction computing σ for the first column.

We also measured the performance of QR factorization for dense matrices, presented in Tables 1 and 2. In the tables, *Canonical GFlops* reflects the Golub and Van Loan flop count for factoring dense matrices [14], and *GPU GFlops* is based on the number of flops actually performed by the GPU device. The algorithm is able to achieve up to 31.83% of the Tesla C2070's peak theoretical double-precision performance.

Rows	Cols	Canonical GFlops	GPU GFlops
128	2048	88.90	89.60
256	4096	110.30	150.09
384	6144	118.97	159.17

Table 1: Performance of 1x16 "short and fat" dense matrices.

Rows	Cols	Canonical GFlops	GPU GFlops
2048	128	29.42	45.47
4096	256	47.40	69.80
6144	384	60.31	87.03

Table 2: Performance of 16x1 "tall and skinny" dense matrices.

We compared our GPU-accelerated sparse QR with Davis' CPU-only SuiteSparseQR on 624 problems from the UF Sparse Matrix Collection [10]. SuiteSparseQR uses LAPACK for panel factorization and block Householder applies while our GPU-accelerated code uses our GPU compute kernels to accomplish the same.

In Table 3, we describe a sample problem set representing a variety of domains. Table 4 shows the results for these seven matrices on our CPU and our GPU, and the relative speedup obtained on the GPU. Intensity refers to arithmetic intensity, the number of floating point operations required to factorize the matrix divided by the amount of memory (in bytes) required to represent the matrix. The flopcount (fl.) is the number of floating point

problem	type	rows	cols	nz
circuit_2	circuit sim.	4,510	4,510	21,199
lp_cre_d	linear prog.	73,948	8,926	246,614
EternityII_A	optimization	150,638	7,362	782,087
olesnik0	2D/3D	88,263	88,263	744,216
lp_nug20	linear prog.	72,600	15,240	304,800
ch7-8-b3	combinatorial	58,800	11,760	235,200

Table 3: Seven matrices from the UF Sparse Matrix Collection [10].

problem	CPU (Gflop)	GPU (Gflop)	intens.	fl.	speedup
circuit_2	0.54	0.14	7.0	0.02	0.26
lp_cre_d	1.75	12.46	203.1	7.14	7.12
EternityII_A	2.53	28.70	425.2	35.35	11.36
olesnik0	4.13	36.94	192.3	173.33	8.95
lp_nug20	23.90	74.31	2110.2	3988.03	3.11
ch7-8-b3	24.12	82.36	1662.9	3458.26	3.41

Table 4: Results for matrices in Table 3.

operations needed to factorize the matrix, in billions (a canonical count, not what the GPU actually performs).

Figure 11 shows the speedup of our GPU-accelerated code over the SuiteSparseQR code as a function of arithmetic intensity for all test matrices, in a logarithmic scale. Many problems experience significant speedup of up to 11x over the CPU-based method. Speedup is limited by two factors:

1. **Available parallel flops**: Dense QR factorization offers $O(n^3)$ flops for $O(n^2)$ memory storage. As arithmetic intensity increases, the algorithm is able to exploit more parallelism than the CPU-based method. However, for small problems such as *circuit_2* in Table 3, the algorithm is unable to exploit enough parallelism. As a result, the time to factorize for small problems is dominated by memory transfer costs.

2. **Hardware resources on the GPU**: Current GPU devices offer several cores arranged into SMs along with small amounts of fast shared memory per SM. Our algorithm is designed to flood the GPU device with many parallel tasks. However, as problem size grows with arithmetic intensity, we reach a performance asymptote as the amount of

available GPU hardware resources begins to limit the performance of our algorithm.

We examined the impact of the pipelined factorization method described in Section 3.1 in which a bundle may be factorized immediately following a block Householder update.

Pipelining both reduces the number of kernel launches required to factorize the problem by a nearly factor of 2, and increases the amount of parallel work sent to the GPU per kernel launch. Pipelining also ensures that nearly every tile of the matrix is modified at each kernel launch, and it also leads to a significantly more uniform amount of workload per kernel launch. However, in context of our current GPU, the pipelined strategy requires 5% more time to factorize large problems. We anticipate that as GPU devices continue to add more SMs, or as we move to a multiple-GPU algorithm with many GPUs, the pipelined factorization will eventually outperform the non-pipelined strategy.

5 Future Work

QR factorization is representative of many other sparse direct methods, with both irregular coarse-grain parallelism and regular fine-grain parallelism, and these methodologies will be very relevant for other methods.

Further parallelism is possible by extending our staging strategy to split the fronts for a stage across multiple GPU devices. The CPU could be used as an additional compute device to factorize some of the fronts in parallel with the GPU. Finally, for distributed memory systems, extending the CPU-based scheduler using MPI would allow for multiple GPU-accelerated systems to participate in the factorization.

6 Related Work

Anderson et al. [3] and Demmel et al. [11, 12] consider how to exploit the orthogonal properties of QR factorization to reduce communication costs in parallel methods for dense matrices. Our *bucket scheduler* is an extension of this idea. They do not consider the sparse case, nor the staircase-form of our frontal matrices. They do not consider multiple factorizations and multiple

assembly operations simultaneously active on the same GPU, as we must do in our sparse QR.

NVIDIA provides a CUDA BLAS for dense matrix operations such as matrix-matrix multiply; see http://www.culatools.com. It includes a dense QR factorization that can achieve 370 GFlops (single precision) on one C2070 Fermi GPU (130 GFlops in double precision). Recently, we have developed matrix multiplication algorithms for the same architecture that provide up to 3% improvement over the CUDA BLAS [16].

NVIDIA has also developed an efficient sparse-matrix-vector multiplication algorithm [5], which achieves 36 GFlops on a GeForce GTX 280 (with a peak performance of 933 GFlops). The performance of sparse-matrix-vector multiplication is limited by the GPU memory bandwidth, since it computes only 2 floating-point operations per nonzero in A. Sparse matrix multiplication is similar to the irregular assembly step in sparse QR.

Krawezik and Poole [15], Lucas et al. [17], Pierce et al. [19], and Vuduc et al. [21] have worked on multifrontal factorization methods for GPUs. All four methods exploit the GPU by transferring one frontal matrix at a time to the GPU and then retrieving the results. The assembly operations are done in the CPU. As far as we know, no one has yet considered a GPU-based method for multifrontal sparse QR factorization, and no one has considered a GPU-based multifrontal method (LU, QR, or Cholesky) where an entire subtree is transferred to the GPU, as is done in the work reported here.

7 Summary

In this paper, we presented a novel sparse QR factorization method tailored for use on GPU-accelerated systems. The algorithm is able to factorize multiple frontal matrices simultaneously, while limiting costly memory transfers between CPU and GPU.

The algorithm uses the master-slave paradigm where the CPU serves as the master and the GPU as the slave. We extend the Communication-Avoiding QR factorization [12] strategy using our bucket scheduler, exploiting a large degree of parallelism and reducing the overall number of GPU kernel launches required to factorize the problem.

The algorithm uses the überkernel design pattern, allowing many different tasks for many different fronts to be computed simultaneously in a single kernel launch. Additionally, the algorithm schedules two flavors of assembly

tasks that move data between memory spaces on the GPU. These assembly tasks are responsible for transfering data from a packed input into frontal matrices prior to factorization as well as transfering data from child fronts to parent fronts. As fronts are factorized, their rows of R are asynchronously transfered off the GPU using the CUDA events and streams model.

For large sparse problems whose frontal matrices cannot simultaneously fit on the GPU, our algorithm examines the frontal matrix assembly tree and divides the fronts into stages of execution. The algorithm then moves data in stages to the GPU, factorizes the fronts within the stage, and transfers the results off the GPU. Contribution blocks are then passed back to the GPU, ready for push assembly.

For large sparse matrices, the GPU-accelerated code offers up to 11x speedup over CPU-based QR factorization methods, and achieves up to 82 GFlops as compared to a peak of 32 GFlops for the same algorithm on a multicore CPU (with 24 cores).

Our code is available at http://www.suitesparse.com.

8 Acknowledgements

We would like to thank NVIDIA for providing support via their Academic Partner program, and the National Science Foundation through grant 1115297.

References

[1] P. R. Amestoy, I. S. Duff, and C. Puglisi. Multifrontal QR factorization in a multiprocessor environment. *Numer. Linear Algebra Appl.*, 3(4):275–300, 1996.

[2] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 3rd edition, 1999.

[3] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-avoiding QR decomposition for GPUs. Technical Report UCB/EECS-2010-131, EECS Dept., UC Berkeley, Oct 2010.

[4] C. C. Ashcraft and R. Grimes. The influence of relaxed supernode partitions on the multifrontal method. *ACM Trans. Math. Softw.*, 15(4):291–309, 1989.

[5] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. Technical Report NVR-2008-004, NVIDIA, Santa Clara, CA, 2008. http://www.nvidia.com/object/nvidia_research_pub_001.html.

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

[7] T. A. Davis. Algorithm 915: SuiteSparseQR, a multifrontal multithreaded sparse QR factorization package. *ACM Trans. Math. Softw.*, 38(1), 2011.

[8] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):377–380, 2004.

[9] T. A. Davis, J. R. Gilbert, S. I. Larimore, and E. G. Ng. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30(3):353–376, 2004.

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

[11] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-avoiding parallel and sequential QR factorizations. Technical report, Univ. of Berkeley, EECS, 2008. http://techreports.lib.berkeley.edu/accessPages/EECS-2008-74.html.

[12] J. W. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comput.*, 34(1):206–239, 2012.

[13] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 9:302–325, 1983.

[14] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. The Johns Hopkins University Press, Baltimore, London, 4th edition, 2012.

[15] G. Krawezik and G. Poole. Accelerating the ANSYS direct sparse solver with GPUs. In *Proc. Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, Urbana-Champaign, IL, 2009. NCSA. http://saahpc.ncsa.illinois.edu/09.

[16] J. Li, S. Ranka, and S. Sahni. GPU matrix multiplication. In S. Rajasekaran, editor, *Handbook on Multicore Computing*. Chapman Hill, 2011 (to appear).

[17] R. Lucas, G. Wagenbreth, D. Davis, and R. Grimes. Multifrontal computations on GPUs and their multi-core hosts. In *VECPAR'10: Proc. 9th Intl. Meeting for High Performance Computing for Computational Science*, 2010. http://vecpar.fe.up.pt/2010/papers/5.php.

[18] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, June 2011.

[19] D. Pierce, Y. Hung, C.-C. Liu, Y.-H. Tsai, W. Wang, and D. Yu. Sparse multifrontal performance gains via NVIDIA GPU. In *Workshop on GPU Supercomputing*, Taipei, Jan. 2009. National Taiwan University. http://cqse.ntu.edu.tw/cqse/gpu2009.html.

[20] A. Tatarinov and A. Kharlamov. Alternative rendering pipelines using NVIDIA CUDA. In *SIGGRAPH 2009*, Aug. 2009.

[21] 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*, HotPar'10, pages 13–13, Berkeley, CA, USA, 2010. USENIX Association.

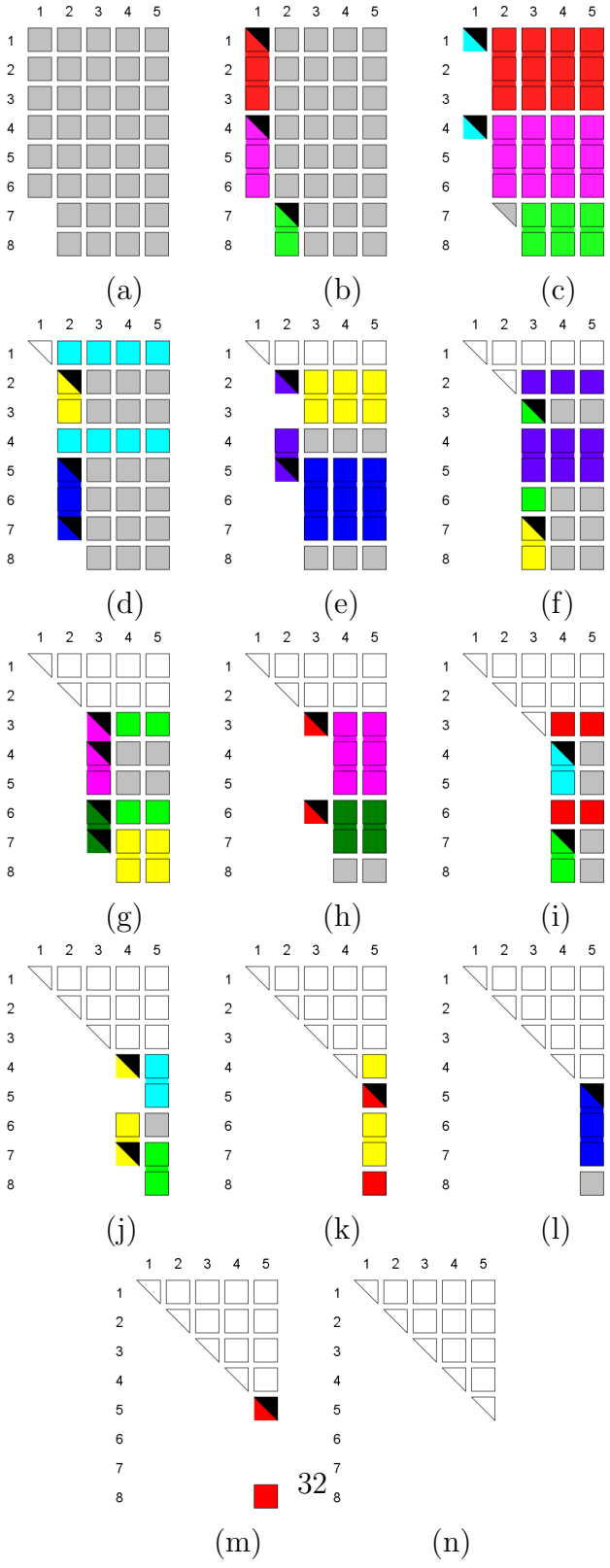

Figure 6: Factorization in 12 kernel launches.

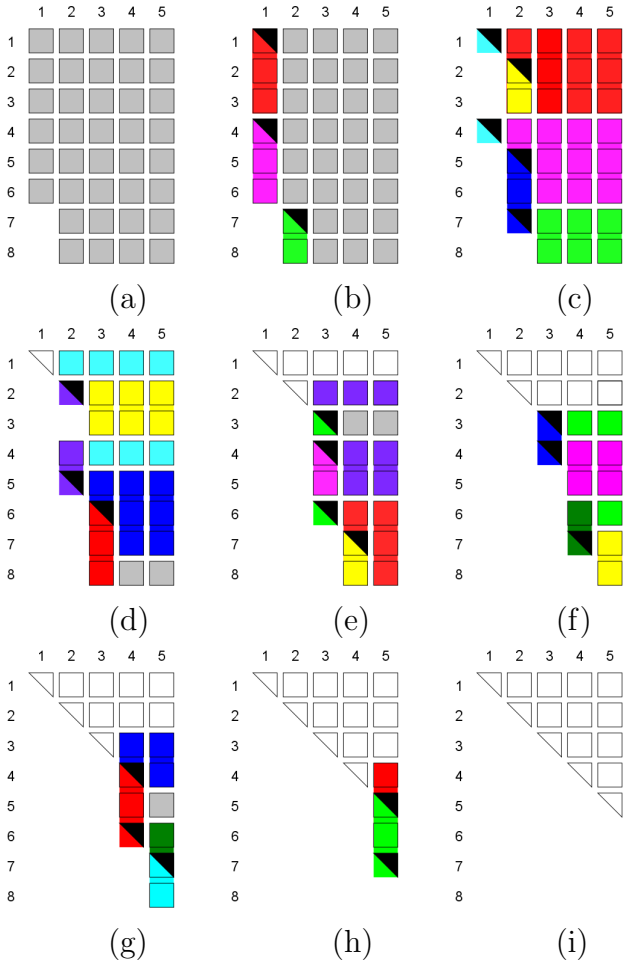

Figure 7: Pipelined factorization in 7 kernel launches.

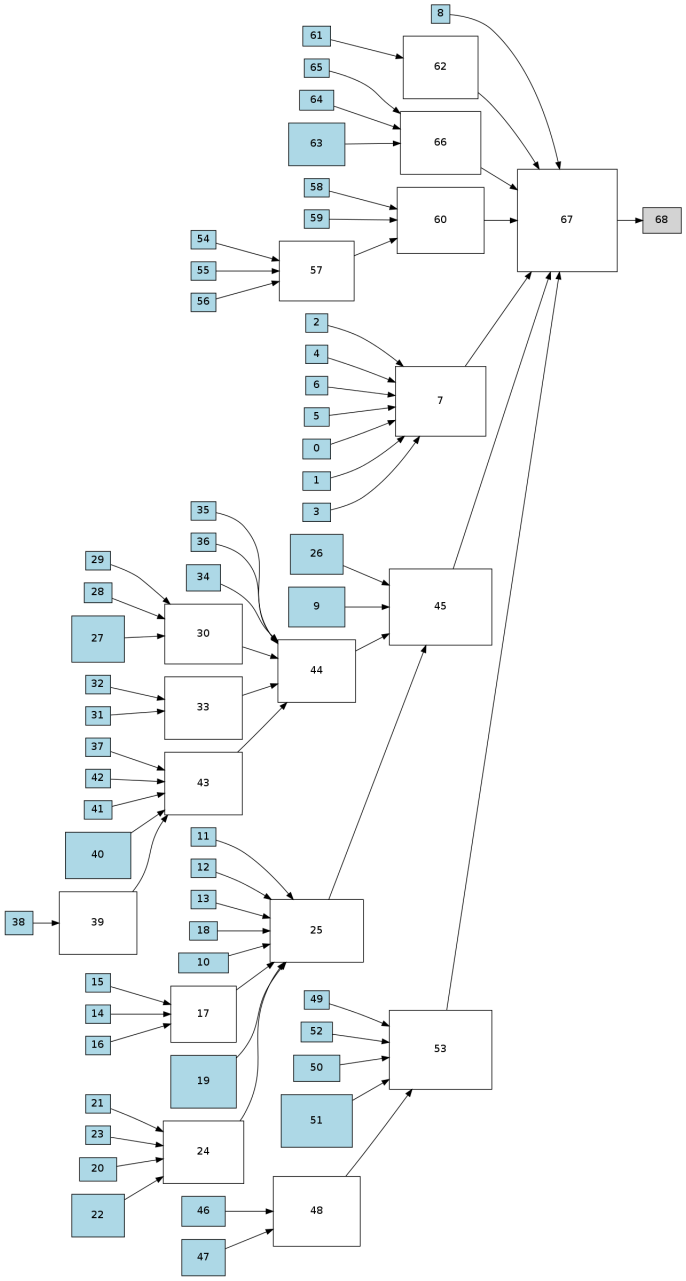

Figure 8: Assembly tree for a sparse matrix with 68 fronts.

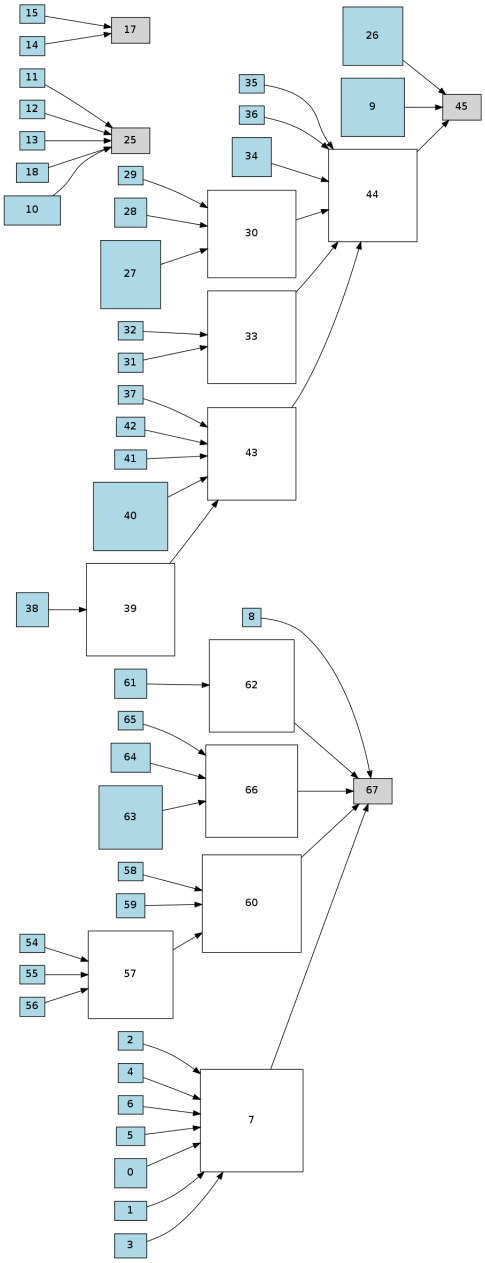

Figure 9: Stage 1 of an assembly tree with 68 fronts.

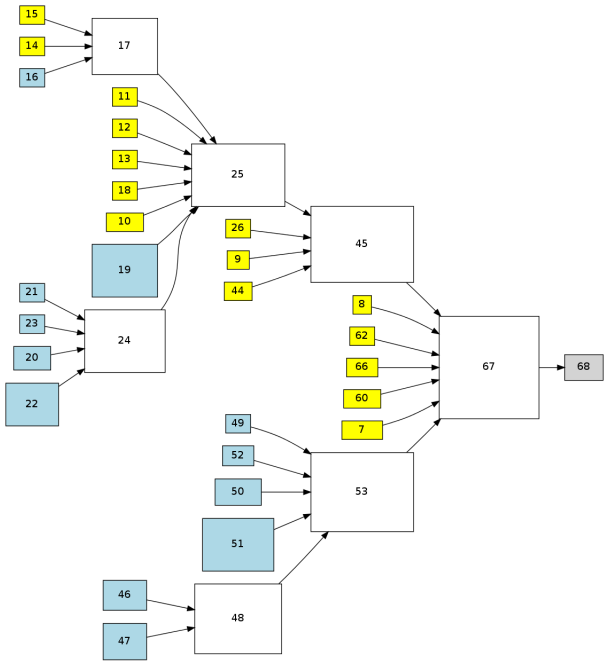

Figure 10: Stage 2 of an assembly tree with 68 fronts.

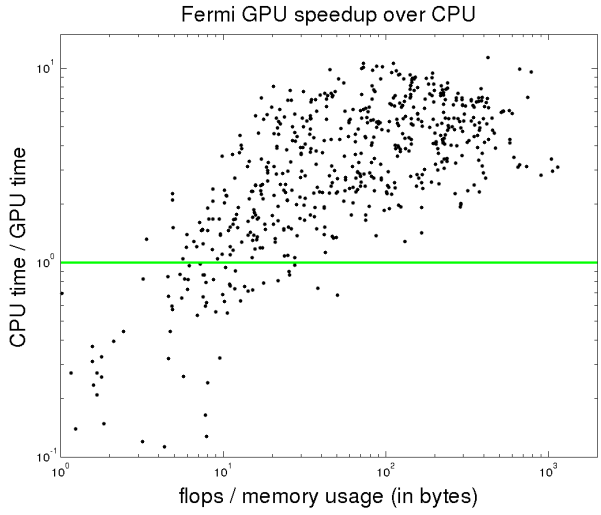

Figure 11: GPU-accelerated speedup over the CPU-only algorithm versus arithmetic intensity on a logarithmic scale.