Communication-Avoiding QR Decomposition for GPUs

Michael Anderson, Grey Ballard, James Demmel and Kurt Keutzer
UC Berkeley: Department of Electrical Engineering and Computer Science
Berkeley, CA USA
{*mjanders,ballard,demmel,keutzer*}*@cs.berkeley.edu*

***Abstract*—We describe an implementation of the Communication-Avoiding QR (CAQR) factorization that runs entirely on a single graphics processor (GPU). We show that the reduction in memory traffic provided by CAQR allows us to outperform existing parallel GPU implementations of QR for a large class of tall-skinny matrices. Other GPU implementations of QR handle panel factorizations by either sending the work to a general-purpose processor or using entirely bandwidth-bound operations, incurring data transfer overheads. In contrast, our QR is done entirely on the GPU using compute-bound kernels, meaning performance is good regardless of the width of the matrix. As a result, we outperform CULA, a parallel linear algebra library for GPUs by up to 17x for tall-skinny matrices and Intel's Math Kernel Library (MKL) by up to 12x.**

We also discuss stationary video background subtraction as a motivating application. We apply a recent statistical approach, which requires many iterations of computing the singular value decomposition of a tall-skinny matrix. Using CAQR as a first step to getting the singular value decomposition, we are able to get the answer 3x faster than if we use a traditional bandwidth-bound GPU QR factorization tuned specifically for that matrix size, and 30x faster than if we use Intel's Math Kernel Library (MKL) singular value decomposition routine on a multicore CPU.

***Keywords*-GPGPU; Linear Algebra; QR Decomposition; Robust PCA;**

I. Introduction

One of the fundamental problems in linear algebra is the QR decomposition, in which a matrix A is factored into a product of two matrices Q and R, where Q is orthogonal and R is upper triangular. The QR decomposition is most well known as a method for solving linear least squares problems, and is used commonly across all of dense linear algebra.

In terms of getting good performance, a particularly challenging case of the QR decomposition is tall-skinny matrices. These are matrices where the number of rows is much larger than the number of columns. For example the ratio of rows to columns can be 3 to 1, 1,000 to 1, or even 100,000 to 1 in some cases. QR decompositions of this shape matrix, more than any other, require a large amount of communication between processors in a parallel setting. This means that most libraries, when faced with this problem, employ approaches that are bandwidth-bound and cannot saturate the floating-point capabilities of processors.

Matrices with these extreme aspect ratios would seem like a rare case, however they actually occur frequently in applications of the QR decomposition. The most common example is linear least squares, which is ubiquitous in nearly all branches of science and engineering and can be solved using QR. Least squares matrices may have thousands of rows representing observations, and only a few tens or hundreds of columns representing the number of parameters. Another example is stationary video background subtraction. This problem can be solved using many QR decompositions of matrices on the order of 100,000 rows by 100 columns [1]. An even more extreme case of tall-skinny matrices are found in *s*-step Krylov methods [2]. These are methods for solving a linear equation $Ax = b$ by generating an orthogonal basis for the Krylov sequence $\{v, Av, A^2v, ...A^nv\}$ for a starting vector v. In *s*-step methods, multiple basis vectors are generated at once and can be orthogonalized using a QR factorization. The dimensions of this QR factorization can be millions of rows by less than ten columns.

These applications demand a high-performance QR routine. Extracting the foreground from a 10-second surveillance video, for example, can require over a teraflop of computation [1]. Unfortunately, existing GPU libraries do not provide good performance for these applications. While several parallel implementations of QR factorization for GPUs are currently available [3], [4], [5], they all use generally the same approach, tuned for large square matrices, and thus have up to an order of magnitude performace degredation for tall-skinny problems. The loss in performance is largely due to the communication demands of the tall-skinny case. We aim to supplement the existing libraries with a QR solution that performs well over *all* matrix sizes.

Communication-Avoiding QR (CAQR) is a recent algorithm for solving QR decomposition which is optimal with regard to the amount of communication performed [6]. This means that the algorithm minimizes the amount of data that must be sent between processors in the parallel setting, or alternatively the amount of data transmitted to and from global memory. As a result, the CAQR algorithm is a natural fit for the tall-skinny case where communication is usually the bottleneck.

In this work we discuss the implementation and performance of CAQR for a single graphics processor (GPU). It

is a distinguishing characteristic of our work that the entire decomposition is performed on the GPU using compute-bound kernels. Despite their increasing general-purpose capabilities, it is still a very challenging task to map the entirety of this particular algorithm to the GPU. In doing so, however, we are able to leverege the enormous compute capibility of GPUs while avoiding potentially costly CPU-GPU transfers in the inner loop of the algorithm. The benefits can most clearly be seen for very skinny matrices where communication demands are large relative to the amount of floating-point work. As a result, we can outperform existing libraries for a large class of tall-skinny matrices. In the more extreme ratios of rows to columns, such as 1 million by 192, we saw speedups of up to 17x over linear algebra libraries parallelized for GPUs. It is important to note that everything we compare to is parallel. The speedups we show are a result of using the parallel hardware more efficiently.

We do not directly use available Basic Linear Algebra Subroutine (BLAS) libraries as a building block. The algorithm requires many hundreds or thousands of small QR decompositions and other small BLAS and LAPACK [7] operations to be performed in parallel. This is not currently supported in the vendor's BLAS library [8]. Consequently, we had to do significant low-level tuning of these very small operations to achieve sufficient performance. We will discuss and quantify the effect of specific low-level optimizations that were critical to improving performance. We also highlight new tradeoffs the GPU introduces, which differ from previous CAQR work on clusters and multicore architectures [9] [10].

Finally, we discuss the application of CAQR to stationary video background subtraction, which was a motivating application for this work. Stationary video background subtraction can be solved using Robust Principal Component Analysis (PCA), by formulating the problem as a regularized nuclear norm minimization [1]. In the Robust PCA algorithm, the video is transformed into a tall-skinny matrix where each column contains all pixels in a frame, and the number of columns is equal to the number of frames. In an iterative process, the matrix is updated by taking its singular value decomposition (SVD) and thresholding its singular values. The SVD can be solved using the QR decomposition as a first step. We will show our CAQR implementation gives us a significant runtime improvement for this problem.

Several implementations CAQR have been done, as well as Tall-Skinny QR (TSQR), a building block of CAQR that deals only with extremely tall-skinny matrices. TSQR has been applied in distributed memory machines [9] [11] and grid environments [12] where communication is exceptionally expensive. However, previous work in large-scale distributed environments focuses on different scales and types of problems than ours. More recently, CAQR was also applied to multicore machines [10], and resulted in speedups of up to 12x over Intel's Math Kernel Library (MKL) at the time. Note, however, that implementing CAQR on GPUs is a much different problem than on multi-core and it is likely that both will be needed in future libraries and applications.

The paper is organized as follows. In Section I we survey previous algorithms and implementations of the QR decomposition. Next we discuss the high-level problem of mapping CAQR to heterogenous systems in Section III. Section IV describes the specifics of our GPU implementation and tuning process. Section V contains our performance compared to currently available software. In Section VI we outline our motivating video processing application and the performance benefits of using CAQR. Section VII draws conclusions.

II. Background on QR Approaches

There are several algorithms to find the QR decomposition of a matrix. For example, one can use Cholesky QR, the Gram-Schmidt process, Givens rotations, or Householder reflectors [13]. It is well-known however that Cholesky QR and the Gram-Schmidt process are not as numerically stable, so most general-purpose software for QR uses either Givens rotations or Householder reflectors. CAQR is a form of the Householder approach, where the Householder vectors are broken up in such a way that communication is minimized.

A. Householder QR

Most existing implementations of QR for GPUs have used the Householder approach, as does LAPACK. One favorable property of the Householder algorithm is that it can be organized in such a way that it makes use of BLAS3 (matrix-matrix) operations. Specifically, the trailing matrix updates for several Householder vectors can be delayed and done all at once using matrix-multiply. This allows for higher arithmetic intensity on machines with a memory hierarchy. Higher arithmetic intensity then leads to better performance. This is called blocked Householder QR, because it allows the updates to the trailing matrix to be blocked in cache. Several implementations of the blocked Householder approach for GPUs are currently available [5] [4], or will soon become available [14]. These are generally all very fast due to the heavy use of well-optimized matrix-multiply routines [15] [3].

Figure 1 shows a sketch of one step in the blocked Householder algorithm. Here, a panel of some width less than the total number of columns is factored using the Householder algorithm with BLAS2 (matrix-vector) operations. Next, a triangular matrix T is formed from the inner products of the columns in the panel. Finally, the trailing submatrix is updated using a matrix-matrix multiply of the panel's Householder vectors, T, and the trailing matrix itself. After the update, the next panel is factored, and so on.

From Figure 1 we can intuitively understand a shortcoming of the blocked Householder algorithm. For very wide matrices, a significant portion of the runtime is spent

Figure 1: Blocked Householder QR

doing matrix-matrix multiply, which is a very efficient use of hardware given the available high-performance dense matrix-multiply routines for GPUs [16]. However, if the matrix is skinny, a greater portion of the runtime is being spent in the BLAS2 panel factorization, which is not an efficent use of the hardware because matrix-vector routines are generally bandwidth-bound. Clever implementations of blocked Householder for heterogeneus CPU+GPU envoronments can hide this cost by sending the panel factorization to the CPU and overlapping it with the previous trailing matrix update, which is performed on the GPU [3]. While this greatly improves the performance for wide matrices, it does not eliminate the latency problem for the skinny case.

B. Tall-Skinny QR (TSQR)

The TSQR algorithm reorganizes the factorization of a tall-skinny matrix (such as a column panel) to minimize memory accesses [6]. Instead of computing one Householder vector for each column directly, we divide the tall-skinny matrix vertically into small blocks, as shown in Figure 2(a). Next, we factor each block independently using Householder reflectors. This creates a small Householder representation of Q, which we call U, and an upper-triangular R for each block. This is shown in Figure 2(b). We would like to eliminate all the Rs below the top-most diagonal, so we can group sets together in a stack and apply the Householder algorithm to each (possibly exploiting the sparsity pattern), which is done in Figure 2(c). We can continue to reduce the Rs with another level as in Figure 2(d). This leaves us with our final upper triangular matrix R, and a series of small Us which, if needed, can be used to generate the explicit orthogonal matrix Q of the factorization.

The TSQR algorithm exposes parallelism. Each block in the panel can be processed independently by a different processor. TSQR also allows us to divide the problem into chunks with a more managable size. If we choose block sizes that fit in cache, we can achieve significant bandwidth savings.

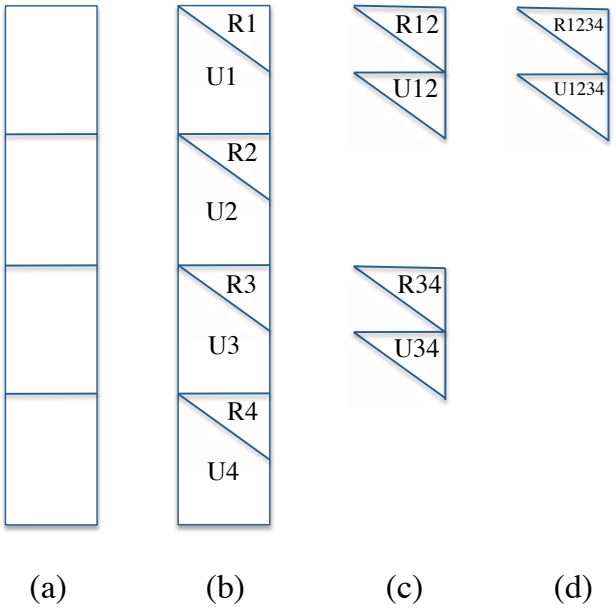

Figure 2: Stages of Tall-Skinny QR

In the figure, the Rs were eliminated in a binary tree. However, this can be done using any tree shape. The optimal shape can differ depending on the characteristics of the architecture. For example, on multi-core machines a binomial tree reduction was used [10], whereas our GPU approach employs a quad-tree reduction. The motivation behind this choice will be expanded in Section IV.

C. Communication-Avoiding QR (CAQR)

CAQR is an extension of TSQR for arbitrarily sized matrices [6]. This time we divide the matrix into a grid of small blocks. Like blocked Householder, CAQR involves a panel factorization and a trailing matrix update. The panel factorization is done using TSQR, shown in Figure 3(a). We must then do the trailing matrix update, which means applying the Q^T of the panel to the trailing matrix. Note that because TSQR works on blocks in the column panel, the trailing matrix update can begin before the entire panel is factored. This removes the synchronization in the standard blocked Householder approach and exposes more parallelism. Unfortunately, we cannot just use a large matrix-matrix multiply as we did in blocked Householder. This is due to the distributed format in which TSQR produces its Q. Instead, we carry out small updates in each small block of the trailing submatrix.

There are two types of trailing matrix updates: horizontal updates and tree updates. Horizontal updates are the easier case. Here we take the Householder vectors generated from the first stage of TSQR and apply them horizontally across

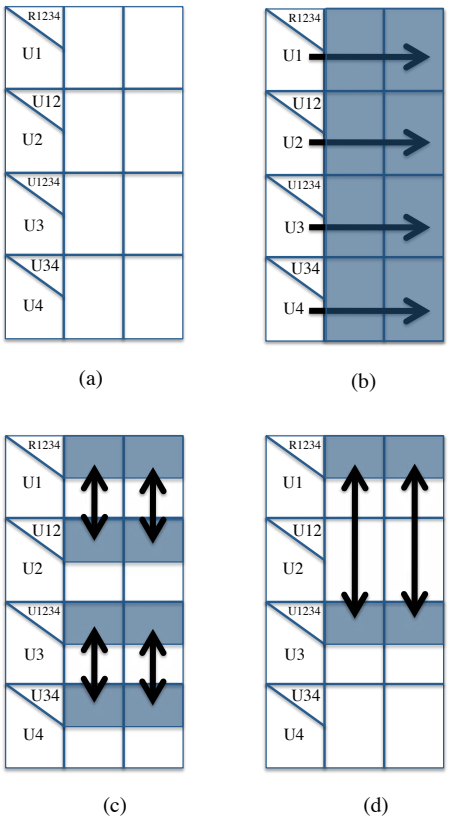

Figure 3: Communication-Avoiding QR

the matrix, shown in Figure 3(b). This operation is very uniform, and the update of each block in the trailing matrix is independent. The more challenging update is the tree update. Here we take the Householder vectors generated during each level of TSQR's tree reduction and apply them across the matrix. This involves mixing small pieces of the trailing matrix, as shown in Figure 3(c) and (d). The rows of the trailing matrix that get updated vary with each level of the reduction tree. This can be challenging on the GPU because the accesses to the matrix are more irregular and somewhat sparse.

After the trailing matrix is updated we can move to the next panel. We must take care to redraw the grid lower by a number of rows equal to the panel width, reflecting the fact that the trailing matrix becomes both shorter and narrower after each step.

III. Mapping CAQR to Heterogenous (CPU+GPU) Systems

Here we briefly discuss two different options for mapping CAQR to current heterogeneous systems. We consider a heterogeneous system containing one or more multi-core host CPUs with DRAM, a GPU with DRAM, and a physical link between the two memories. The GPU has more compute and bandwidth capability generally than the CPUs, whereas CPUs generally have a larger cache, more ability to exploit instruction-level parallelism, and are better equipped to handle irregular computation and data accesses.

The two questions we want to answer with regard to CAQR are: where we should do each step of the computation? Where should we store the data?

A. First option: CPU panel factorization and GPU trailing matrix update

With this approach, the algorithm proceeds as follows. A thin panel is sent to the CPU, if necessary, and the CPU factors the panel using TSQR. The result of the factorization is sent back to the GPU and used for the trailing matrix update. Potentially, the CPU could begin factoring the next panel while the GPU is still busy applying the previous panel to the trailing matrix.

The main advantage of this approach is that offloading work to the CPU makes it possible to overlap GPU and CPU work. This allows you to use the entire system. The TSQR panel factorization can be a good fit for the CPU because of the irregular nature of the reduction tree. The trailing matrix update is regular and can be done efficiently on the GPU.

One disadvantage of this approach is that in order to offload work to the CPU we must transfer the data between CPU and GPU memories. On current systems, this involves latency that can hurt performance for skinny problems. Unless we successfully overlap CPU and GPU computation, sending the panel factorization to the CPU means we cannot use the superior compute and bandwidth capabilities of the GPU for these parts of the computation.

B. Second Option: Entire factorization the GPU

With this approach, the entire factorization is done on the GPU. This includes the TSQR panel factorization and the trailing matrix updates.

Assuming the matrix is entirely in GPU memory, this approach eliminates transfer latency. This means that we can get good performance even on skinny problems. We also can benefit from the higher compute and bandwidth capability of the GPU for the panel factorizations.

Unfortunately, this approach is much more difficult to program. This is first because we cannot reuse existing tuned CPU libraries. Also, certain parts of the QR algorithm involve more irregular computations and are therefore more challenging and less efficient to carry out in the GPU's programming and execution models. The pseudocode in Figure 4, described in the next section, illustrates some of the irregular operations necessary for a GPU-only approach.

In this work we choose the second option, performing the entire factorization on the GPU, for the following reason. Our motivating application is Robust PCA for stationary

video background subtraction. The dimensions of the video matrices we deal with in this application are on the order of 100,000 tall by 100 wide. For this size problem, the latency of transferring data to the CPU will have high adverse impact on performance. During the course of the application, we are doing many QR decompositions and the video matrix is able to stay on the GPU. So the cost of initially transferring the video matrix to GPU memory is easily amortized.

IV. GPU Implementation

In this section we describe implementation details of our QR decomposition for the GPU. We start with an overview of the hardware, then give a high-level description of how the computation is organized. This is followed by an examination of the key individual kernels and low-level tradeoffs. Everything here is done using single-precision, which is adequate for our video application.

A. Hardware Overview

Our target is an NVIDIA C2050 GPU, which is the version of NVIDIA's recent Fermi architecture intended for high-performance computing. This choice of architecture is motivated by GPUs track record for good performance on high-throughput numerical computing [16], [17]. Though we use the C2050, our code can be run on any CUDA-capable GPU.

The 1.15 GHz C2050 has 14 multiprocessors, each of which has 32 floating-point units capable of executing one single-precision FLOP per cycle. In total, the chip is capable of 1.3 single precision TFLOPs.

Tasks are scheduled on the multiprocessors in units called *thread blocks*. Each thread block contains up to 512 threads, which can synchronize and share data among one another through a small shared memory. Thread blocks, however, are assumed to be independent and do not generally synchronize or communicate during a single parallel task. As an example, in our application each small block QR decomposition is done by a different thread block. Within the thread block there are 64 threads that work together to compute the QR.

There is one large global memory (DRAM) shared by all multiprocessors. The peak bandwidth to and from global memory for the C2050 with ECC enabled is 144 GB/sec. There is a small 768 KB L2 cache shared by all multiprocessors. Each multiprocessor has access to 48 KB of shared memory and 16 KB of L1 cache, which can alternatively be configured as 48 KB of L1 and 16 KB of shared memory. The register file is the largest and fastest local memory available to the processing units. It is 128 KB per multiprocessor, can be accessed in one or two cycles, and generally provides enough bandwidth to saturate the floating point units. Registers are not shared. Any time threads need to communicate, in a parallel reduction for example, either shared memory or DRAM must be used.

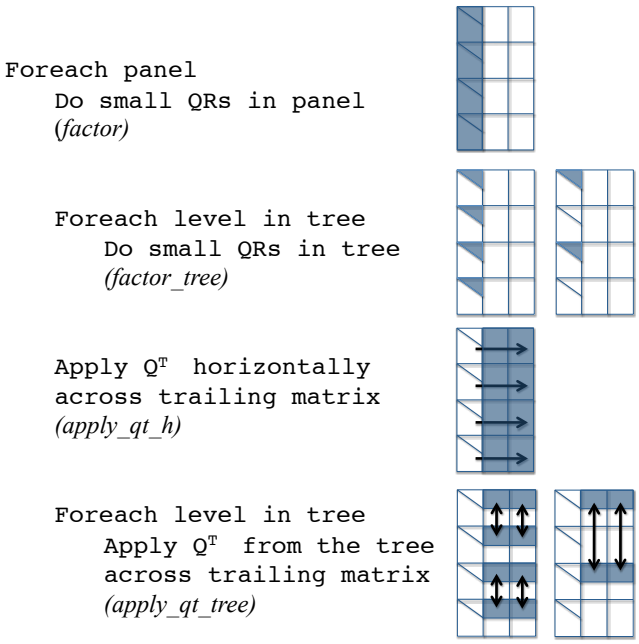

Figure 4: Host pseudocode for CAQR with a binomial reduction tree

B. Top-Level Pseudocode

Figure 4 shows pseudocode for our CAQR. Each function represents a GPU *kernel*, which is a subroutine that is performed in parallel over different parts of the matrix. Next to each kernel call there is a graphic of our matrix showing how it is divided among thread blocks and which portions of the matrix are being processed by that kernel. So for example, the TSQR panel factorization is done in the first two kernel calls to *factor*. Once that has completed, the trailing matrix is updated using *apply_qt_h* and *apply_qt_tree*. This pseudocode is executed on the host CPU. The CPU coordinates the GPU's actions and updates pointers so that GPU thread blocks know where to look in the matrix.

C. Reduction Tree

The shape of our reduction tree is a function of the block sizes. For example, if the block size is 64×16, each block produces an R which fits in a 16×16 space. When these Rs are stacked for the next level of the reduction tree, we can fit $\frac{64}{16} = 4$ of them in each 64×16 block. This means we reduce the height of the panel by a factor of 4 at each level and the reduction is a quad-tree. The tree reduction ends when the panel height becomes less than 64, meaning it can be processed completely in a single thread block.

D. Overview of Kernels

The following four kernels are the building blocks of our CAQR. The parts of the matrix affected by each kernel are

shown in Figure 4.

1) factor: Perform a QR decomposition of a small block in fast memory using customized BLAS2 routines. Overwrite the Householder vectors and upper triangular R on top of the original small input matrix.

2) factor_tree: Gather a stack of upper triangular Rs generated by *factor* and store them in fast memory. Then perform a QR decomposition on that small block, as was done in *factor*. The shape of the resulting Householder vectors and R is also a stack of upper triangular matrices, and thus can overwrite the Rs that were read into fast memory.

3) apply_qt_h: Apply Q^T from the Householder vectors generated in *factor* horizontally to small blocks across the trailing matrix. Write back the updated trailing matrix blocks to the locations from which they were read.

4) apply_qt_tree: Apply Q^T from the Householder vectors generated by *factor_tree* during the TSQR reduction tree to the correct locations in the trailing matrix. To do so, collect the distributed components of the trailing matrix to be updated as well as the distributed Householder vectors. Perform the application of Q^T, and write back the updated trailing matrix blocks to the same distributed locations from which they were read.

Inserting these kernels into the pseudocode in Figure 4 should complete a high-level understanding of our CAQR implementation.

E. Kernel Tuning

Now we examine an individual kernel so as to understand the specifics of the data layout and thread-level execution. Fortunately, all four kernels do the same two core computations: matrix-vector multiply and rank-1 update. We will analyze only the simplest of these which is $apply\_qt\_h$.

Suppose we have a set of Householder vectors in shared memory and a small block A to which we'd like to apply the vectors. This is equivalent to applying Q^T, represented by the Householder vectors, i.e. $Q = \prod_{i=1}^{n}(I - \tau_i u_i u_i^T); \tau_i = \frac{2}{||u_i||_2^2}$, to the block A. For each Householder vector u we first compute the matrix-vector product $A^T * u$. This is shown in Figure 5(a). Then we update each element of A with the scaled outer product of $A^T * u$ and u, Figure 5 (b). So the computation here consists of a reduction sum of each column of A during the matrix-vector product, and a data-parallel rank-1 update of A. This is repeated for each Householder vector.

Two important questions must be answered in our design. How should we assign threads to computation, and how do we store the matrix in our fast memories such that it can be accessed most quickly? During the course of tuning we tried several different approaches. The main difference between these lies in how the reductions in the matrix-vector product are carried out. The following is a description of each approach to the matrix-vector product, with its

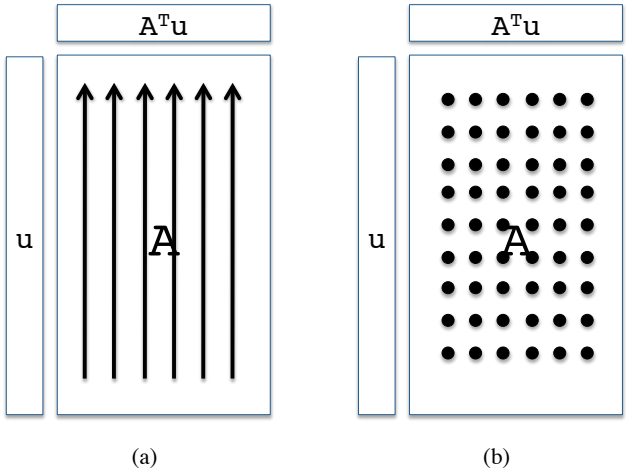

Figure 5: Matrix-vector multiply and rank-1 update. These two operations make up the core computations in each kernel

corresponding performance for the matrix-vector product and rank-1 update on 128×16 blocks.

1) Shared Memory Parallel Reductions (55 GFLOPS): The most obvious approach is to store the matrix in the register file assigning each thread one row. Then reduce each column one at a time using parallel reductions in shared memory. This approach is inefficient because many of the threads sit idle during the consecutive parallel reductions.

2) Shared Memory Serial Reductions (168 GFLOPS): The 128×16 matrix is originally stored in column major order, so we put it into shared memory to reduce it serially. The advantage of this approach is that it gives us near optimal thread utilization for the reduction.

3) Register File Serial Reductions (194 GFLOPS): Store the matrix entirely in the register file. Instead of distributing the data to each thread by row, distribute it cyclically among the threads, as shown in Figure 6. Notice that each thread's data is located in a single column. This means that each thread can do a serial reduction over its part of the matrix and write only the result into shared memory for parallel reduction. This minimizes traffic in and out of shared memory, which helps performance.

4) Register File Serial Reductions + Transpose (388 GFLOPS): The register file serial reduction (Approach 2) can be improved if the block is already stored in transposed form. Instead of doing a small transpose in each thread block, this transpose can be done as a preprocessing step. This is beneficial because these kernels are called many times on the same block of the matrix. This is not a transpose of the entire matrix. We only need to transpose each panel from column major to row major. Unfortunately this means that the factorization is done out of place, as an in-place transpose is difficult for non-square matrices.

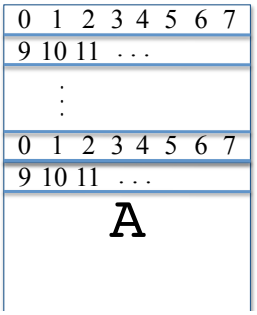

Figure 6: In approaches 2 and 4, the matrix is stored in the register file and distributed among threads in this manner. All data owned by a thread belongs to the same column.

F. Autotuning Block Size

In the previous section we showed how changing the data layout and reduction technique could improve performance. After commiting to a data layout, we can write scripts to test many different block sizes and choose the best.

Our block size is fundamentally limited by our shared memory size and/or register file size. We must however decide what the shape of the block should be. The *apply_qt_h* kernel gets better performance when the block width is wider. This is for several reasons. First, since the number of FLOPs performed in the Householder algorithm is $\mathcal{O}(mn^2)$, we get far higher arithmetic intensity, i.e. FLOPs/byte, by increasing the width n, than we do by increasing the height m. Secondly, the wider the block is the more parallelism there is in each matrix-vector product. If the block width were equal to the number of threads, for example, then the matrix-vector product could be done entirely using efficient serial reductions.

However, the Householder vector u must be communicated to every thread. If the block is so wide that each thread owns an entire column, then each thread must read the entire u vector from shared memory. The optimal solution is somewhere between the two extremes. The performance of each block size is shown in Figure 7. Our best overall performance comes from using 128×16 blocks. For the *apply_qt_h* kernel we are able to get 388 GFLOPS.

G. Tuning Summary

Through our tuning process we were able to improve the performance of *apply_qt_h*, our main kernel, from 55 GFLOPS to 388 GFLOPS using low-level tuning. The main focus was optimizing the matrix-vector product and rank-1 update that is at the core of the Householder QR algorithm. The most important tuning optimization was to avoid shared memory and L1 cache in favor of the register file. Also, choosing the right block size was critical in achieving good performance.

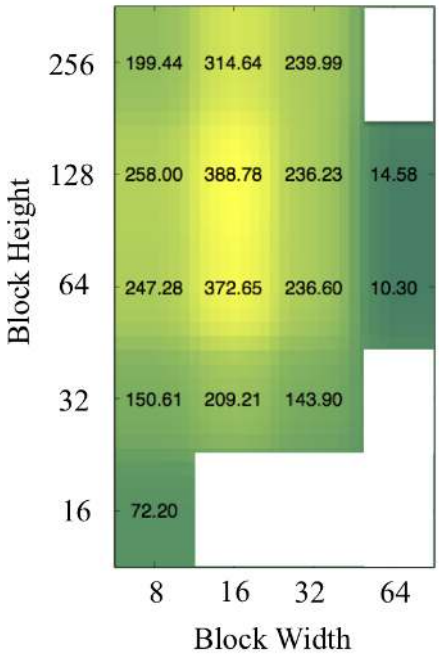

Figure 7: Performance for various choices of block size in single precision GFLOPS.

V. Performance

We compare our performance against common commercially available and open source software for both GPU and multicore. Below we describe the software to which we compare and the details of the hardware platforms. Our code is optimized for tall-skinny matrices, so we will focus mostly on this case. All FLOPs are single precision.

A. Software

1) Intel Math Kernel Library (MKL): MKL contains BLAS, LAPACK, and various other common functions tuned specifically for Intel's processors. In our comparison we use version 10.2.2.025, and link to the multithreaded library.

2) MAGMA: Matrix Algebra on GPU and Multicore Architectures (MAGMA) is an open source dense linear algebra library for heterogeneous CPU+GPU systems. It is developed by the Innovative Computing Laboratory at the University of Tennessee. Version 1.0 is used in this comparison.

3) CULA: CULA is a library of LAPACK routines for GPUs [5]. It is a commercial library, made by EM Photonics. Since CULA source is not available, we do not know exactly how they implement their QR. However, the performance of the QR routine across different square matrix sizes is

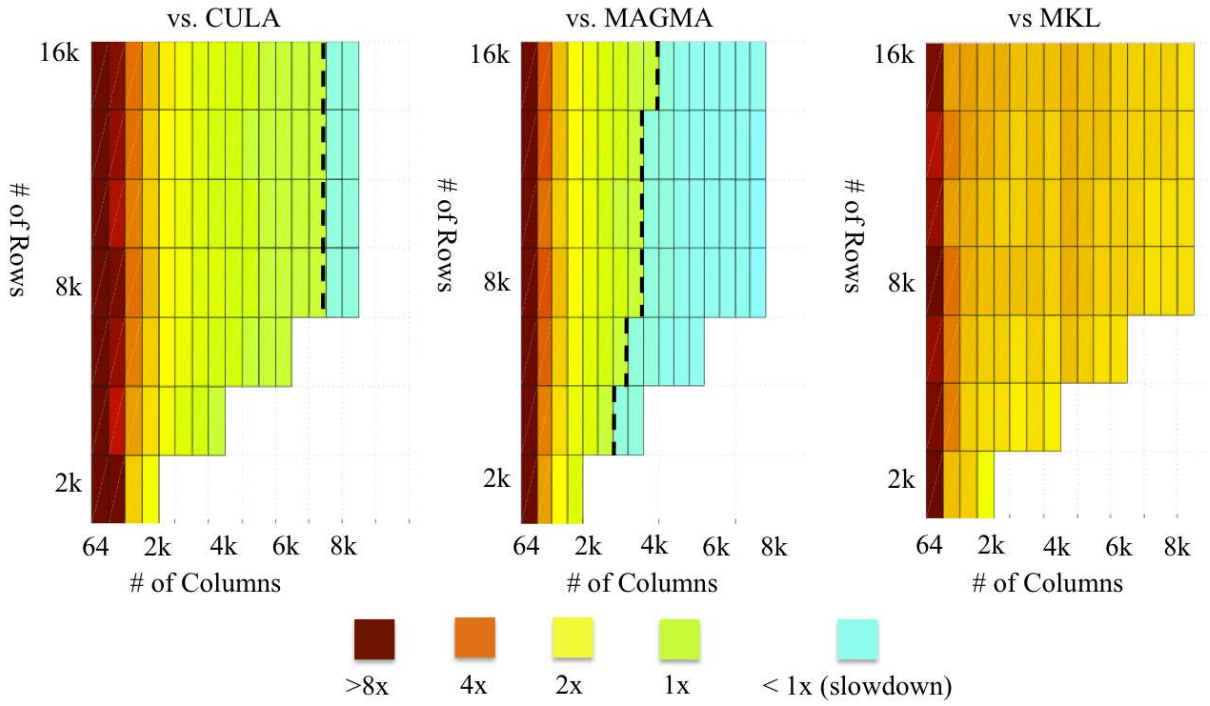

Figure 8: Speedup vs. SGEQRF from popular linear algebra libraries on a range of different matrix sizes. The dashed line is a crossover point, to the right of which the libraries outperform our QR.

very similar to the performance of a previous blocked Householder approach by Volkov et al.[3]. For this reason, we will not separately report the performance of CULA and Volkov.

B. Hardware

Our test platform is the Dirac GPU cluster at the National Energy Research Scientific Computing Center (NERSC) [18]. One node consists of dual-socket quad-core Intel 5530 processors, 8 cores total, running at 2.4 GHz connected over PCI-express to an NVIDIA C2050 (Fermi) GPU. The GPU has ECC enabled, so its effective bandwidth is reduced to 144 GB/s.

C. Performance vs. Matrix Width

Figure 8 shows the performance of our single-precision CAQR code for a range of matrix sizes compared to MAGMA, CULA, and MKL. Each point in the chart represents a different matrix size. The points on the left are skinnier matrices, such as those found in robust background subtraction, and the points on the far right are square matrices. Our CAQR implementation is tuned for the tall-skinny matrices. As a result, we see large speedups compared to other linear algebra libraries for this case. The operation being performed is a single precision QR factorization defined by the LAPACK SGEQRF routine. This routine doesn't return Q explicitly, instead it returns an intermediate representation that can later be used to retreive or apply Q. Though not shown on the graph, retreiving Q explicitly (SORGQR) using CAQR is just as efficient as factoring the matrix. All preprocessing (e.g. transpose) done for CAQR is included in these runtimes. For all GPU routines, the matrix is assumed to begin on the GPU. The initial data transfer from CPU to GPU is not counted.

The absolute performance numbers in single precision GFLOPS are shown in Figure 9. In this case we consider a matrix with a fixed height of 8192 and varying width from 64 to 8192. The crossover point, where CAQR becomes slower than the best best GPU libraries, is around 4000 columns wide. This suggests an autotuning framework for QR where a different algorithm may be chosen depending on the matrix size.

D. Very Skinny Matrices

In the case of extremely tall-skinny matrices, such as those found in our video processing application, we see up to 17x speedups vs. GPU libraries and 12x vs. MKL. Table I shows the performance on extremely tall-skinny matrices for CAQR, MAGMA, CULA, and MKL. Another application where matrices like these appear is communication-avoiding linear solvers, when vectors must be orthoganalized periodically [2].

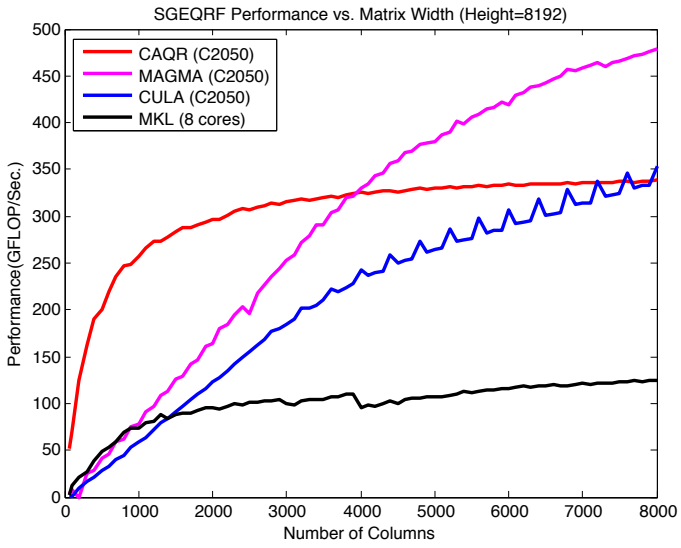

Figure 9: Performance on matrices with 8192 rows and varying numbers of columns. For the tall-skinny case CAQR performs best

		Performance (SP FLOPS)			
		CAQR	MAGMA	CULA	MKL
Matrix Size	1k x 192	39.6	5.01	2.99	3.12
	10k x 192	111	18.7	9.67	16.9
	50k x 192	174	20.8	9.42	22.8
	100k x 192	180	18.8	8.90	21.4
	500k x 192	194	12.4	8.40	17.8
	1M x 192	195	11.4	7.79	16.5

Table I: Performance in single precision GFLOPS for very tall-skinny matrices.

VI. Application: Robust PCA for Surveillance Video Background Subtraction

The motivating application for this work is stationary video background subtraction using a recent statistical algorithm for Robust Principal Component Analysis (PCA) [1]. This section will present specifics of the application, how it uses the QR decomposition, and the performance of the application using ours and other QR implementations.

A. Robust PCA

Principal Component Analysis is a widely used method for data analysis. The goal is to find the best low rank approximation of a given matrix, as judged by minimization of the difference between the original matrix and the low rank approximation. However, the classical method is not robust to large sparse errors. In Robust PCA, a matrix M is decomposed as the sum of a low rank component L_0 and a sparse component S_0. S_0 is allowed to have entries that are large in absolute value, as long as they are sparse.

Figure 10: Sample output of Robust PCA for stationary video background subtraction

$$M = L_0 + S_0$$

The problem is solved using ℓ_1 regularized nuclear norm minimization. Minimizing the nuclear norm, the sum of the singular values, of L_0 enforces low-rank. Meanwhile, the ℓ_1 norm of S_0 enforces sparsity. Minimizing a weighted combination of these two penalty functions with linear constraints is convex.

An application of Robust PCA is stationary video background subtraction. A survellience video is transformed into a tall-skinny matrix where each column contains all pixels in a frame, and the number of columns is equal to the number of frames. The low-rank component of this matrix is the background and the sparse component is the people walking in the foreground. To give a better idea of the problem being solved, Figure 10 shows a sample of the output of the Robust PCA code.

B. SVD using QR

The main computation in Robust PCA is a singular value decomposition (SVD) of the tall-skinny video matrix. In the SVD of the video matrix, the top singular values, those that have a strong presence of every frame of the video, are usually associated with the background.

Instead of trying to do a large SVD on the GPU, we use the following well known technique for tall-skinny matrices to reduce the bulk of the work to a QR decomposition. First A is decomposed into $Q * R$. Then we find the SVD of R, which is cheap because R is an $n \times n$ matrix and done on the CPU using MKL. Next, we can multiply the orthogonal matrices $Q * U$ to get the left singular vectors of A.

$$\begin{aligned} A &= Q * R \\ &= Q * \left(U * \Sigma * V^T\right) \\ &= (Q * U) * \Sigma * V^T \\ &= U' * \Sigma * V^T \end{aligned}$$

C. Robust PCA Algorithm

The algorithm for Robust PCA tries to minimize the rank of L_0 and enforce sparsity on S_0. It does so with an iterative alternating-directions method [19]. The flowchart for the algorithm is shown in Figure 11. The algorithm thresholds

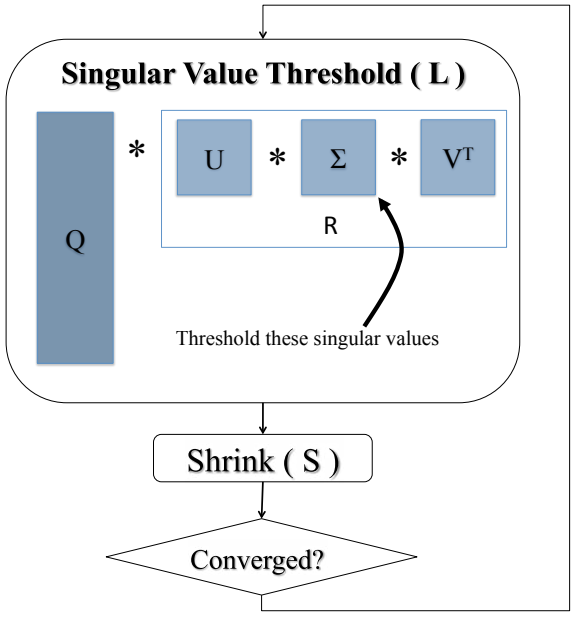

Figure 11: Flowchart of the alternating-directions algorithm for solving Robust PCA

(sets to zero) the smallest singular values of L_0 in order to make it low rank. Next, a shrinkage operation (pushing the values of the matrix towards zero) is done on S_0 to enforce sparsity. The vast majority of the runtime is spent in the singular value threshold, specifically the SVD of the L_0 matrix.

D. Performance using CAQR

We have three different versions of Robust PCA for video background subtraction. The first uses entirely a CPU (Intel Core i7 2.6 GHz), and relies on multithreaded MKL version 10.2.5.035 for the SVD as well as other basic BLAS routines. The second is done entirely on the GPU (GTX480), except for the small SVD of R which is done on the CPU, and uses our BLAS2 QR decomposition that was specifically designed and tuned for tall-skinny matrices. Finally, there is a version which also runs on the GPU and uses our CAQR.

Our benchmark video comes from the ViSOR surveillance video database [20]. We extract 100 frames for processing. Each frame is 288 pixels tall by 384 pixels wide, which is a total of 110,592 pixels per frame. This means the matrix dimensions are 110,592 by 100. The problem technically takes over 500 iterations to converge, however the solution begins to look good earlier than that. The quality of solution, and therefore number of iterations required, seems to depend on the application. We therefore report the number of iterations per second that each implementation is able to complete. All computation is done in single precision.

SVD type	Number of Iterations/Sec.
MKL SVD (4 cores)	0.9
BLAS2 QR (GTX480)	8.7
CAQR (GTX480)	27.0

Table II: Performance of various Robust PCA implementations

Table II shows moving from the CPU-only code to our BLAS2 GPU code results in a 9.6x speedup. This mostly reflects the fact that the GPU has much higher bandwidth and compute power than our CPU, and that the MKL SVD function may not be optimized for the tall-skinny case. However, we see an additional speedup of about 3x when using CAQR as compared to the BLAS2 QR. Even though the QR itself is sped up by more than a factor of 3, we only get 3x in the application overall due to Ahmdal's law. Overall our GPU solution gives us a 30x speedup over the original CPU code using MKL, reducing the time to solve the problem completely from over nine minutes to 17 seconds, making this approach feasible for latency-critical applications [21].

VII. Conclusion

In this paper we described a high-performance implementation of Communication-Avoiding QR Decomposition entirely on a single GPU using compute-bound kernels. The main advantage of our approach over the traditional blocked Householder algorithm is that it can handle tall-skinny matrices without relying on bandwidth-bound BLAS2 panel factorizations or potentially high-latency GPU-CPU transfers.

We showed low-level implementation choices that allowed us to achieve good performance on the GPU. The best performance for our kernels came from using the register file as much as possible and arranging the data in transposed form so as to minimize necessary communication between threads. Our tuning improved the performance of the most heavily-used kernel from 55 GFLOPS to 337 GFLOPS.

Our CAQR code outperformed leading parallel CPU and GPU libraries for tall-skinny matrices up to roughly 4000 columns wide and 8192 rows tall. In more extreme ratios of rows to columns, such as 1 million by 192, we saw speedups of up to 17x over GPU linear algebra libraries. Note that these extreme cases were motivated by practical applications.

Finally, we applied the CAQR code to Robust PCA for stationary video background subtraction. We showed that using CAQR we could achieve a 3x speedup over our best BLAS2 QR tuned specifically for the tall-skinny case, and a 30x speedup over a CPU implementation using MKL's parallel SVD.

REFERENCES

[1] E. Candes, X. Li, Y. Ma, and J. Wright, "Robust principal component analysis," *Preprint*, 2009.

[2] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proceedings of the 2009 ACM/IEEE conference on supercomputing*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 36:1–36:12.

[3] V. Volkov and J. Demmel, "LU, QR and Cholesky factorizations using vector capabilities of GPUs," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2008-49, May*, pp. 2008–49, 2008.

[4] A. Kerr, D. Campbell, and M. Richards, "QR decomposition on GPUs," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2009, pp. 71–78.

[5] J. Humphrey Jr, D. Price, K. Spagnoli, A. Paolini, and E. Kelmelis, "CULA: hybrid GPU accelerated linear algebra routines," in *Proceedings of SPIE*, vol. 7705, 2010, p. 770502.

[6] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *LAPACK Working Note #204*.

[7] E. Anderson, Z. Bai, and C. Bischof, *LAPACK Users' guide*. Society for Industrial Mathematics, 1999.

[8] C. NVIDIA, "CUBLAS Library," *NVIDIA Corporation, Santa Clara, California*, 2008.

[9] J. Demmel, L. Grigori, and M. Hoemmen, *LAPACK Working Note #204*.

[10] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra, "Enhancing parallelism of tile qr factorization for multicore architectures," *Matrix*.

[11] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[12] E. Agullo, C. Coti, J. Dongarra, T. Herault, and J. Langem, "QR factorization of tall and skinny matrices in a grid computing environment," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–11.

[13] J. Demmel, *Applied numerical linear algebra*. Society for Industrial and Applied Mathematics, Philadelphia, 1997.

[14] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," in *Journal of Physics: Conference Series*, vol. 180. IOP Publishing, 2009, p. 012037.

[15] G. Quintana-Ortí, F. Igual, E. Quintana-Ortí, and R. van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," *ACM SIGPLAN Notices*, vol. 44, no. 4, pp. 121–130, 2009.

[16] V. Volkov and J. Demmel, "Benchmarking GPUs to tune dense linear algebra," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, pp. 1–11.

[17] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*. IEEE, 2009, pp. 1–12.

[18] NERSC, "Experimental GPU cluster: Dirac," http://www.nersc.gov/nusers/systems/dirac/.

[19] X. Yuan and J. Yang, "Sparse and low-rank matrix decomposition via alternating direction methods," *Preprint*, 2009.

[20] R. Vezzani and R. Cucchiara, "ViSOR: Video surveillance on-line repository for annotation retrieval," in *Multimedia and Expo, 2008 IEEE International Conference on*. IEEE, 2008, pp. 1281–1284.

[21] Y. Dong and G. DeSouza, "Adaptive Learning of Multi-Subspace for Foreground Detection under Illumination Changes," *Computer Vision and Image Understanding*, 2010.