QR Decomposition on GPUs

Andrew Kerr, Dan Campbell, Mark Richards

Georgia Institute of Technology, Georgia Tech Research Institute

{andrew.kerr, dan.campbell}@gtri.gatech.edu, mark.richards@ece.gatech.edu

ABSTRACT

QR decomposition is a computationally intensive linear algebra operation that factors a matrix A into the product of a unitary matrix Q and upper triangular matrix R. Adaptive systems commonly employ QR decomposition to solve overdetermined least squares problems. Performance of QR decomposition is typically the crucial factor limiting problem sizes.

Graphics Processing Units (GPUs) are high-performance processors capable of executing hundreds of floating point operations in parallel. As commodity accelerators for 3D graphics, GPUs offer tremendous computational performance at relatively low costs. While GPUs are favorable to applications with much inherent parallelism requiring coarse-grain synchronization between processors, methods for efficiently utilizing GPUs for algorithms computing QR decomposition remain elusive.

In this paper[1], we discuss the architectural characteristics of GPUs and explain how a high-performance implementation of QR decomposition may be implemented. We provide detailed performance analysis of the resulting implementation for real-valued matrices and offer recommendations for achieving high performance to future developers of dense linear algebra procedures for GPUs. Our implementation sustains 143 GFLOP/s, and we believe this is the fastest announced QR implementation executing entirely on the GPU.

1. INTRODUCTION

Graphics Processing Units are massively parallel processors capable of completing hundreds of floating point operations in parallel. Their large register files and high-performance scratchpad memories are well-suited to streaming execution models. Matrix factorization algorithms such as Cholesky, LU, and QR decomposition, however, typically require fine-grain synchronization between processors and contain short serial routines as well as massively parallel operations. Achieving good utilization on a GPU requires a careful implementation informed by detailed understanding of the performance characteristics of the underlying architecture.

In this paper, we focus on QR decomposition in particular and discuss the suitability of several algorithms for implementation on GPUs. Then, we provide a detailed discussion and analysis of how blocked Householder reflections may be used to implement QR on CUDA-compatible GPUs supported by performance measurements. Our real-valued QR implementation achieves more than 10x speedup over the native QR algorithm in MATLAB and over 4x speedup beyond the Intel Math Kernel Library executing on a multicore CPU, all in single-precision floating-point. We present throughput for real-valued implementations of this QR implementation executing on several GPU architectures and believe this is the fastest strictly GPU implementation yet presented. Our implementation performs all processing on the selected GPU, enabling threads on the host processor to perform other computations and to devote PCI-Express bandwidth to other tasks.

2. BACKGROUND

Commodity GPUs are inexpensive resources for delivering very high computing throughput for certain classes of applications. GPUs are sold primarily as an integrated component in display adapters for desktop personal computers. High-throughput GPUs are primarily aimed for the video game market. The primary application of GPUs has a very large degree of potential parallelism at the most computationally intensive step: applying final shading effects to each pixel in a polygon as it is rendered into the display buffer. This fact has allowed GPU vendors to exploit microarchitecture parallelism for increased performance without constraint by the application and without requiring much architectural infrastructure to facilitate parallel execution. The volume of the GPU market provides tremendous competitive pressure to improve performance and keep prices low over successive product generations.

Simultaneously, GPU execution models have grown in flexibility in response to the needs of graphics programmers thereby enabling a wide range of computing tasks. GPU vendors have consequently developed graphics-agnostic programming models such as NVIDIA's Compute Unified De-

[1]This work was supported in part by DARPA and AFRL under contracts FA8750-06-1-0012 and FA8650-07-C-7724. The opinions expressed are those of the authors.

GPGPU '09, March 8, 2009, Washington, D.C., USA.

vice Architecture (CUDA) [1] and Open Compute Layer (OpenCL) [2] to facilitate general purpose computing on GPUs. Nevertheless, fully exploiting the peak performance capacity of GPUs has remained a challenge. Algorithms with very high arithmetic intensity, very little need to synchronize between execution paths, and very few scatter operations typically perform well on GPUs without the need for careful optimization, but many computing tasks do not fit these idealized constraints. Several algorithms for fast QR decomposition exhibit a high degree of parallelism, but have low arithmetic intensity and are highly coupled between execution paths, requiring synchronization between elements after small numbers of arithmetic operations. As a result, attempts to exploit GPUs to accelerate QR decomposition have been moderately successful achieving 4x speedup [3] over a reference implementation distributed with Lincoln' Laboratory's HPEC Challenge [4]. In this paper, we base speedup numbers on the highly optimized Intel Math Kernel Library which is among the fastest QR implementations available for multicore CPUs.

3. QR ALGORITHMS

QR decomposition factors an m-by-n matrix A into the product $A = QR$, where Q is an m-by-m unitary matrix and R is an m-by-n upper triangular matrix. Several one-sided factorization methods compute the QR decomposition. Two of these, Givens and Householder [5], apply a set of orthogonal transformations to the input matrix to bring it into upper triangular form. By concatenating these orthogonal transforms, the matrix Q is formed, all while preserving the invariants $A = QR$ and $Q^H Q = I$. Modified Gram-Schmidt computes the QR factorization via projection operations. Each of these methods has favorable numerical properties and offers some parallelism. The suitability of each algorithm for GPU implementation depends on memory access patterns, the frequency of inter-processor synchronization, and the scalability of parallelism within the algorithm.

3.1 Modified Gram-Schmidt

The modified Gram-Schmidt QR (MGS) method computes $A = Q_1 R_1$ where A is m-by-n, Q_1 is m-by-n with orthonormal columns, and R_1 is n-by-n. This factorization differs from what we have defined in that Q is not square. MGS offers fairly good numerical properties, but it consists of operations that are not favorable to high performance on GPUs: computing each element of the output matrix R requires large vector inner products and many synchronizations [5]. Implementing MGS on a GPU would incur prohibitive overhead. A blocked method exists in which the matrix to be factored is first partitioned into a number of sufficiently small submatrices which are independently factored. However, this approach is susceptible to precision problems in which columns of Q are not sufficiently orthonormal [6]. We did not consider a fast implementation of blocked MGS QR decomposition for this paper.

3.2 Givens QR

In the Givens method of QR, a sequence of rotations applied to the input matrix A place zeros in the trapezoidal submatrix below the main diagonal. Each rotation $G(\theta)$ is a Givens rotation, a unitary matrix chosen such that

$$G(\theta)\begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} c & s \\ -s^* & c \end{bmatrix}\begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}$$

The algorithm begins by choosing elements from the bottom two rows in A in the left-most column. These determine the the Givens rotation $G_{m,1}(\theta)$ which may then be used to multiply in-place the submatrix of A formed by the bottom two rows. Then, $G_{m,1}(\theta)^H$ may be used to multiply a pair of columns $Q(:, m-1:m)$ in place. The multiplication in A has the desired effect of overwriting the bottom left element with 0 while preserving the invariants in the algorithm. Next, the algorithm proceeds up one row in A and repeats, computing $G_{m-1,1}(\theta)$ and applying it to $A(m-2:m-1,:)$ and $Q(:, m-2:m-1)$. This proceeds throughout the entire column stopping at the main diagonal at which point all of the left column in A except the first element is overwritten with zeros. The algorithm moves right one column and repeats from the bottom row, again stopping at the main diagonal. When all columns have been visited, A is in upper triangular form, and the QR decomposition is complete.

$G(\theta)$ may be computed from f and g without explicitly computing θ or evaluating any trigonometic functions [7]. Moreover, Sameh and Kuck [8] demonstrate a pattern in which Givens rotations may be computed in parallel. This method is adaptable to streaming architectures and systolic arrays such as MIT RAW [9]. In general, this approach achieves good performance on MIMD architectures that support low-latency communication and synchronization. GPUs, on the other hand, do not offer on-chip mechanisms for synchronizing the several SIMD processors and require either invoking a sequence of kernels with implicit synchronization barriers between each invocation or assuming a consistency model for global memory and implementing barriers in the shared global address space. Performance results for Givens QR implemented on CUDA-compatible GPUs have been presented for the HPEC Challenge benchmarks [3]. However, attempts to develop a high performance implementation of this algorithm in CUDA were met with limited success and do not scale well to arbitrarily large matrix sizes.

3.3 Householder QR

Householder QR computes the upper triangular matrix R from an m-by-n matrix A by applying a sequence of *Householder reflections* to A in place [5]. A Householder reflection is an orthogonal transform of the form

$$P = I - \frac{2}{v^H v} v v^H$$

where v is a *Householder vector*. v may be chosen from a vector x such that $Px = e^{j\theta}||x||e_1$, where P is unitary and e_1 is a column vector with 1 in the first element and 0 in all other elements. Part of column k of a matrix A denoted x_k is chosen such that the first element $x_k(1)$ is on the diagonal and the rest of x_k occupies the lower part of A. A Householder reflection P_k computed from x_k may be applied to A in place overwriting column x_k with $P_k x_k$ and updating other columns also. We see that $P_k x_k$ is nonzero

only in the first element, and all elements below the main diagonal of $P_k A$ are now 0.

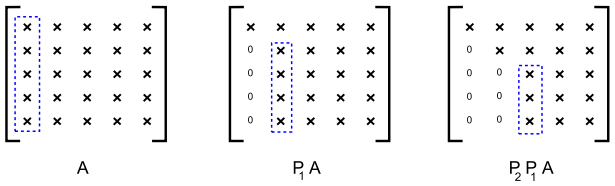

Figure 1: Triangularizing A with Householder reflections.

Figure 1 illustrates how a sequence of Householder transforms may be chosen from columns of A to bring it into triangular form. In the figure, the dashed rectangle highlights the vector x_k from which a Householder vector v_k is computed. v_k is used to construct the unitary matrix $P = I_{m-k+1} - \beta v_k v_k^H$. By overwriting A with $P_k A$, zeros in column k are placed below the main diagonal. Moving to the right one column and down one row, the process repeats until A is triangularized. Because P_k is unitary, this sequence may be applied while maintaining the invariant that $A = (P_1^H P_2^H \cdots P_{n-1}^H)(P_{n-1} \cdots P_2 P_1 A)$. We see this is the QR decomposition with

$$\begin{aligned} Q &= (P_1^H P_2^H \cdots P_{n-1}^H) \\ R &= (P_{n-1} \cdots P_2 P_1 A) \end{aligned}$$

Applying a Householder reflection does not require a general matrix-matrix product, for

$$\begin{aligned} PA &= (I - \beta v v^H) A \\ &= A - v w^H \end{aligned}$$

where $w = \beta A^H v$. This gives rise to the following QR algorithm.

Algorithm 1 Compute the QR factorization of A via Householder reflections [5].

Require: $Q^H Q = I$
1: $Q \leftarrow I$
2: **for** $k = 1$ to n **do**
3: $[v, \beta] =$ **house**$(A(k:m,\ k))$
4: $A(k:m, k:n) = A(k:m, k:n) - \beta v v^H A(k:m, k:n)$
5: $Q(1:m, k:m) = Q(1:m, k:m) - \beta Q(1:m, k:m) v v^H$
6: **end for**

The function **house**(x) returns the Householder vector $v = x - e^{j\theta}||x||e_1$, where $x(1) = re^{j\theta}$, and $\beta = \frac{2}{v^H v}$.

3.4 Blocked Householder QR

The above algorithm is conceptually simple and dominated by matrix-vector multiplies. However, the amount of computation per memory element fetched from global memory is quite low. To improve performance, an algorithm in which several Householder transforms may be applied in a single operation was sought. Bischof and Van Loan [6] explain how the Householder QR algorithm may be generalized to represent multiple Householder transforms as a single transformation matrix. Rather than apply Householder reflections as rank-1 updates to the identity matrix,

$$\begin{aligned} P &= P_1 P_2 \cdots P_r \\ &= (I - \beta_1 v_1 v_1^H)(I - \beta_2 v_2 v_2^H) \cdots (I - \beta_r v_r v_r^H) \end{aligned}$$

the m-by-r matrices W and Y are computed such that

$$\begin{aligned} P_{wy} &= P_1 P_2 \cdots P_r \\ &= I + W Y^H \end{aligned}$$

The above operation is rich in matrix-matrix products and can be expected to achieve high performance on GPUs given sufficiently large problem sizes. Moreover, $P_{wy} A$ may be computed as $P_{wy} A = A + W(Y^H A)$ requiring $5mnr$ FLOPs if the operations are performed in the order indicated by the parentheses. This is fewer FLOPs than multiplying the m-by-n matrix A by the m-by-m matrix P_{wy}. Block size may be chosen based on the shared memory capacity of the target architecture and the warp size that leads to maximum performance for matrix multiply. The formation of W may be performed from a block of Householder vectors stored in the lower trapezoidal part of V and from the vector B containing corresponding β_h. The algorithm to form W and Y from r Householder vectors is as follows.

Algorithm 2 Computation of W and Y from V and B [5]

1: $Y = V(1:end, 1)$
2: $W = -B(1) \cdot V(1:end, 1)$
3: **for** $j = 2$ to r **do**
4: $v = V(:, j)$
5: $z = -B(j) \cdot v - B(j) \cdot W Y^H\ v$
6: $W = [W\ z]$
7: $Y = [Y\ v]$
8: **end for**

Algorithm 1 may be modified by partitioning the columns of the input matrix A into blocks of r columns. For block k, r Householder reflections are computed and applied to triangularize the columns of that block as in the original algorithm. Rather than apply a single Householder transform to the columns of the remaining blocks as before, we instead compute W_k and Y_k as in Algorithm 2 and then apply $P_{wy} = I + W_k Y_k^H$ to the remaining blocks of A and to all of Q. The matrix-vector products are performed on matrices of size $(m - kr) \times r$, and the updates to the remaining blocks and to Q are accomplished by matrix-matrix products. r should be chosen to minimize total runtime on the target architecture. For $r = n$, the blocked Householder algorithm degenerates into Algorithm 1. The entire procedure is specified in Algorithm 3. Workloads in units of real floating point operations are expressed for each phase of the above algorithm in Table 1 for problem sizes of interest.

Table 1: Workload for real blocked Householder QR in GFLOPs.

Dimension	**house**$(A(:,u))$	$A = P \cdot A$	WY	$A = P_{wy}^H \cdot A$	$Q = Q \cdot P_{wy}$	Total
512×256	0.000196	0.00463	0.00656	0.0518	0.211	0.275
1024×512	0.000786	0.0184	0.0257	0.433	1.66	2.14
1536×768	0.00177	0.0413	0.0571	1.48	5.55	7.14
2048×1024	0.00314	0.0734	0.102	3.54	13.1	16.8
2560×1280	0.00491	0.115	0.158	6.94	25.6	32.8
3072×1536	0.00708	0.165	0.228	12.0	44.1	56.5
3584×1792	0.00963	0.224	0.310	19.1	70.0	89.7
4096×2048	0.0126	0.293	0.404	28.6	104	134
4608×2304	0.0159	0.371	0.511	40.7	149	190
5120×2560	0.0197	0.458	0.631	55.9	204	261
6656×3328	0.0332	0.773	1.65	123	447	572
8192×4096	0.0503	1.17	25.0	230	833	1090

Algorithm 3 Block Householder QR

Require: $A \in C^{m \times n}$, $Q^H Q = I$
1: $Q \leftarrow I$
2: **for** $k = 1$ to n/r **do**
3: $s = (k-1) \cdot r + 1$
4: **for** $j = 1$ to r **do**
5: $u = s + j - 1$
6: $[v, \beta] =$ **house**$(A(u : m, u))$
7: $A(u : m,\ u : s + r - 1) = A(u : m,\ u : s + r - 1) - \beta v v^H A(u : m,\ u : s + r - 1)$
8: $V(:, j) = [\text{zeros}(j - 1, 1);\ v]$
9: $B(j) = \beta$
10: **end for**
11: $Y = V(1 : end, 1)$
12: $W = -B(1) \cdot V(1 : end, 1)$
13: **for** $j = 2$ to r **do**
14: $v = V(:,\ j)$
15: $z = -B(j) \cdot v - B(j) \cdot W Y^H v$
16: $W = [W\ z]$
17: $Y = [Y\ v]$
18: **end for**
19: $A(s : m, s + r : n) = A(s : m, s + r : n) + Y W^H A(s : m, s + r : n)$
20: $Q(1 : m, s : m) = Q(1 : m, s : m) + Q(1 : m, s : m) W Y^H$
21: **end for**

4. IMPLEMENTATION

Algorithm 3 was first implemented in C++ with the CUBLAS library distributed with CUDA 2.0 to obtain baseline performance results. Each function call into CUBLAS was instrumented with performance counters to measure the fraction of total runtime spent in each operation. The impact of timing instrumentation on total runtime is small compared to total runtime. The instrumented algorithm was executed five times, and the timing profile across all runs was averaged. Functions dominating runtime were then selected for custom implementations as one or more CUDA kernels.

The CUDA execution model [1] specifies a collection of multiprocessors that share a global address space. Each multiprocessor is composed of 8 datapaths that execute a "warp" of 32 threads in SIMD fashion. When the threads of a warp stall due to reaching a synchronization barrier or memory operation, another warp on the same processor is scheduled for execution. Kernels are designed with several concurrent warps executing on each multiprocessor such that the GPU performs computation while memory transfers complete.

In this implementation, where possible, operations requiring several CUBLAS function calls were combined into single kernels. For example, the norm computation dominates the runtime of the **house**() function. Because each block of A is transformed exclusively by reflections, the norms of the columns of each block are invariant and may be computed in parallel by a single kernel invocation before the block is triangularized. This avoids the overhead of calling shorter kernels many times and utilizes all multiprocessors, with each multiprocessor performing a reduction operation corresponding to a particular column.

The datapaths of a multiprocessor access a large register file partitioned among the many threads and a 16 kB scratchpad known as "shared memory." Shared memory is addressable by load and store instructions and is striped across 16 ports. Each port is 32 bits wide, and if every thread of a "half-warp" accesses a different port, the load or store instruction will not stall. Selecting access patterns to shared memory that avoid port conflicts reduces stall rates and maximizes throughput. Typically, this requires skewed access to two-dimensional arrays in shared memory among threads of the same warp. Storing blocks of frequently used data in registers reduces pressure on shared memory and avoids additional instructions to required load values into registers before they may be used as operands.

Algorithm 3 was expected to be bottlenecked by matrix-vector products within the main loop of the algorithm. The CUBLAS prototype makes several calls to the CUBLAS function `cublasSgemv()` to compute the product $\beta A' v$ and the product $\beta A v$. To improve performance, these were implemented with custom CUDA kernels according to the architecture characteristics discussed in this section. Shared memory is used only to store part of the vector v. Columns of the matrix A are streamed into the register file with coalesced read operations where they are used to multiply corresponding elements of the V vector. The CUDA compiler is capable of automatically unrolling loops with constant cycle counts [[1] §4.2.5.2]. All threads attempt read access to the

same address in shared memory, so no bank conflict occurs when loading `V_shared[j]`. To avoid potentially expensive thread divergence, guard conditionals are excluded and matrix dimensions are assumed to be multiples of 64. The following kernel exhibits an average 2.5x speedup over the CUBLAS 2.0 [10] function `cublasSgemv('n',..)` for problem sizes of interest. As described in [11], a relatively small block size of 64×1 threads was selected to reduce overheads associated with loop index and address calculations. This departs from the CUDA Programming Guide which recommends a large number of threads to maximize occupancy [[1] §5.2].

Listing 1: Matrix-vector product

```
/*
    Computes W = beta A V + alpha W
*/
__global__ void gtSgemv(
    float alpha, float beta, float *A,
    float *V, int M, int N, float *W) {

    int row = blockIdx.x * 64 + threadIdx.x;
    __shared__ float V_shared[64];
    float w = alpha * W[row];

    A += row;
    V += threadIdx.x;

    for (int k = 0; k < N; k += 64) {
        V_shared[threadIdx.x] = *V;
        V += 64;
        __syncthreads();
        for (int j = 0; j < 64; j++) {
            w += *A * V_shared[j];
            A += M;
        }
        __syncthreads();
    }
    W[row] = beta * w;
}
```

Performance of an m-by-m matrix multiplying an m element vector is illustrated in Figure 2. The target GPUs for this benchmark are the GeForce GTX 280 and GeForce 9800 GX2. Additionally, a theoretical upper bound for floating-point performance in GFLOP/s suggested by global memory bandwidth is plotted for each architecture. For the GeForce GTX280, theoretical bandwidth is 141 GB/s; for the GeForce 9800GX2, theoretical peak bandwidth is 64 GB/s. The number of FLOPs for the `Sgemv` operation is computed as $2mn + 2m$, and total data transferred assuming no redundancy is $4mn + 4n$ bytes. This performance bound ignores kernel launch overheads which are deemed to be negligible for large problems. As illustrated, the custom `gtSgemv` kernel achieves a maximum of 69 GFLOP/s or 97% of theoretical performance when executed on the GeForce GTX280 for large matrices. Examining the CUBLAS source, we see that the access patterns of `cublasSegmv('n', ..)` are similar to the kernel in Listing 1, but the kernel body includes a significant number of integer operations and shared memory loads and stores, all of which reduce floating point intensity.

Similarly, a kernel computing a matrix-vector product was written to compute $\beta A^H v$. Together, these functions demand the majority of runtime during the triangularization

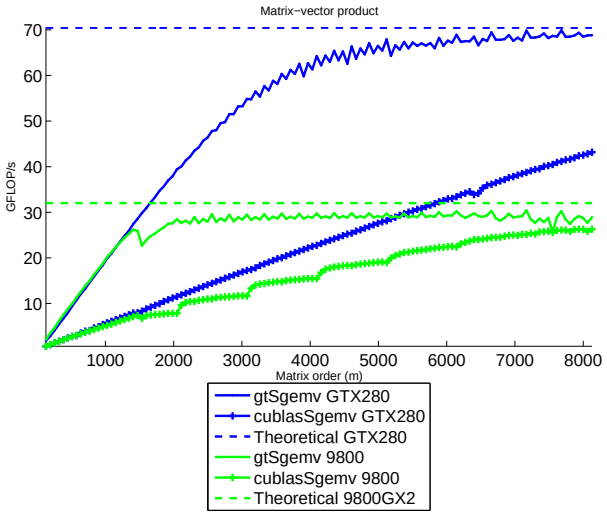

Figure 2: Performance of matrix-vector product for matrices of dimension m

phase and the formation of the WY representation of a set of Householder reflections. All custom kernels access submatrices and vectors beginning on rows that are multiples of 16 to ensure memory transfers are aligned to 64 byte boundaries and complete in as few tranfer operations as possible. The additional elements are overwritten with zeros once they are loaded into registers. CUBLAS matrix-matrix product functions are called to compute the updates for A and Q, lines 19 and 20 in Algorithm 3. Matrix product operations in CUBLAS have been measured to obtain over 400 GFLOP/s sustained and approach peak multiply-add performance for the architectures under test [11].

5. PERFORMANCE RESULTS

The performance of this implementation of QR decomposition was measured by computing the QR factorization of real-valued rectangular matrices with twice as many rows as columns corresponding to typical overdetermined least squares problems. The input matrix A was initialized with random values from -1 to 1 below the main diagonal, 0 written above the main diagonal, and 1s along the diagonal to ensure full rank. Then, randomly selected Givens rotations were applied to A to conceal all apparent structure while preserving rank. Although some applications of QR do not require Q to be explicitly formed, performance results presented here include the computation of the full m-by-m Q matrix. Error criteria were selected as follows.

$$\begin{aligned} ||QR - A|| &\leq m \cdot 2^{-23} ||A|| \\ ||Q^H Q - I|| &\leq m \cdot 2^{-23} \\ ||L|| &\leq m \cdot 2^{-23} \end{aligned}$$

where L is the trapezoidal submatrix below the main diagonal of R. All tests for which performance results are reported satisfy these error criteria. The testbed application was compiled and linked with the CUDA 2.0 toolchain and executed on a Q6600 quad-core Intel Core2 at 2.4 GHz running Windows XP. Additionally, a reference application was implemented with the Intel Math Kernel Library 10.0

and executed on a 64-bit Linux machine with a quad-core Intel Xeon CPU at 2.8 GHz. For both applications, the matrix data type was in single-precision, and all matrices were in column major order. For the GPU test application, all data was resident in device memory before timing measurements were made and no PCI-Express bus traffic was considered. This scenario is typical of a real-world GPU application using CUBLAS or GPU VSIPL [12] in which intermediate results are stored on the device, and only final outputs are copied back to host memory.

The QR procedure was instrumented with CPU-based performance counters to record the number of cycles required by each phase of the algorithm. `cudaThreadSynchronize()` was called after each kernel invocation to ensure the kernel had completed before stopping the cycle counter. No numerical processing was performed on the CPU for the GPU algorithm, and runtime is almost entirely a function of the target GPU's performance. This implementation was tested on two GPU architectures supporting CUDA: NVIDIA GeForce 9800 GX2 and GeForce GTX 280. For each target, the CUDA compiler `nvcc` was invoked with a flag for the maximum possible shader model version supported by the target GPU. The flag `-maxrregcount` was issued to `nvcc` to clamp the register usage to 32 registers per thread to avoid spilling to local memory on the 9800 GX2. The register file for the GTX280 is twice as large, accommodating larger warp sizes without register spilling.

Overall runtime for various matrix sizes is presented in Figure 3, and performance with respect to workload is illustrated in Figure 4. The maximum problem size for the GeForce 9800 GX2 was limited by its 512 MB of global memory per GPU. Figure 5 illustrates the speedup of the GeForce GTX 280 with our QR algorithm over the Intel MKL library's support for QR decomposition implemented by the LAPACK functions `sgeqrf()` and `sormqr()`; MKL was run with 1, 2, and 4 threads.

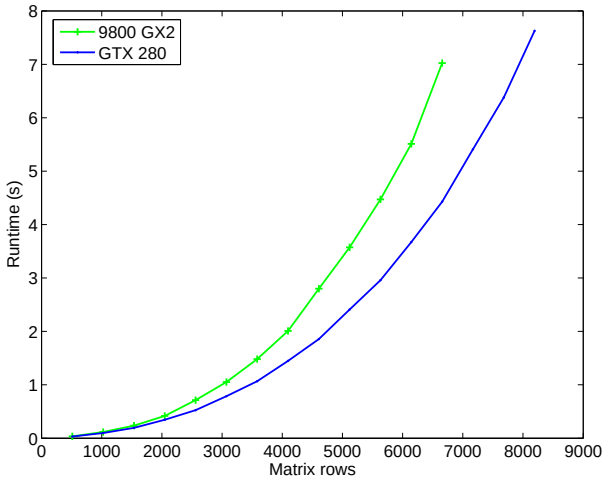

Figure 3: Runtimes of QR decomposition on GPUs.

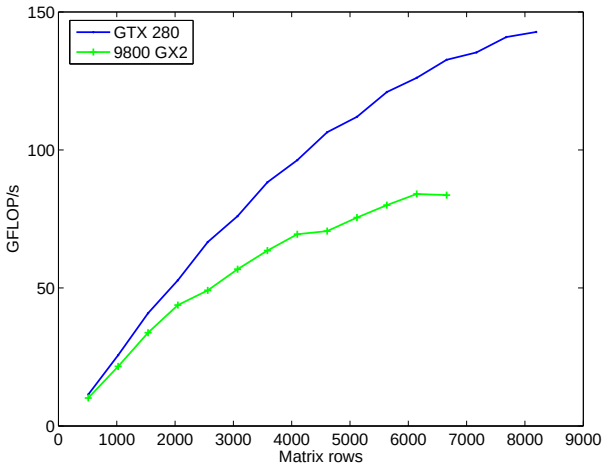

Figure 4: Sustained performance of QR decomposition on GPUs.

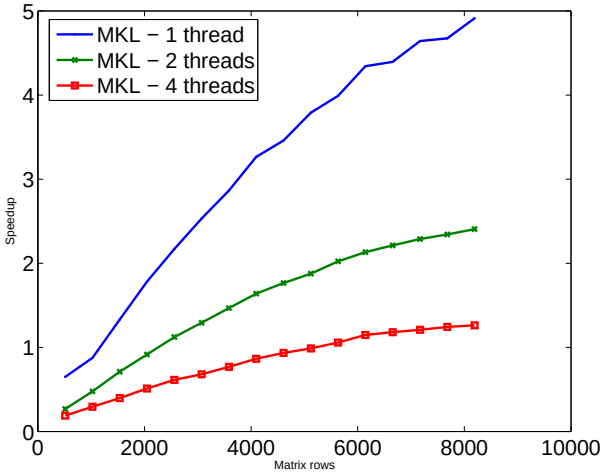

Figure 5: Speedup of GTX280 QR implementation over MKL.

The GeForce GTX280 achieves 143 GFLOP/s sustained performance with our QR implementation. This constitutes the highest performance of a single processor for real QR decomposition we are aware of. As predicted, triangularizing a block of A and forming the WY representation of the Householder reflections requires much more runtime than applying the blocked Householder reflections to the rest of A. Applying the reflections to Q requires a significant fraction of runtime, as Q is full with m rows and m columns. Moreover, while the submatrix of A to which P_{wy} is applied grows smaller as the algorithm progresses, the number of rows in Q to which P_{wy} is applied remains constant. As Figure 6 illustrates, the large workload in transforming Q achieves the highest performance, as it consists of large matrix-matrix products. Table 2 expresses the distribution of runtime across the several phases of our QR implementation for the largest possible problem sizes on both the GeForce 9800 GX2 and the GeForce GTX 280. The fraction of total runtime for each phase did not change appreciably with problem size except for very small matrices.

Table 2: Runtime in seconds for phases of blocked Householder QR on GPUs.

Operation	GeForce 9800 GX2	GeForce GTX 280	GeForce GTX 280
Problem size	6656×3328	6656×3328	8192×4096
Householder	0.360	0.326	0.565
$A = P \cdot A$	1.25	0.952	1.45
WY Computation	1.10	1.25	1.86
$A \leftarrow (I + WY^H)^H A$	1.10	0.534	0.971
$Q \leftarrow Q(I + WY^H)$	3.21	1.36	2.79
Total (seconds)	7.02	4.43	7.629
GFLOP/s	81.5 GFLOP/s	129 GFLOP/s	143 GFLOP/s

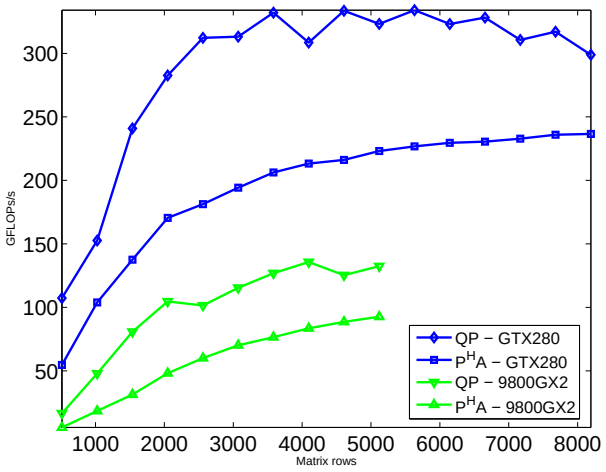

Figure 6: Performance of $A \leftarrow P^H A$ and $Q \leftarrow QP$ for GeForce GTX280 and GeForce 9800.

6. RELATED WORK

Volkov and Demmel [11] present strong performance results for a general matrix product in the context of a QR algorithm. We confirm several of their performance measurements such as kernel invocation and synchronization costs and maximum achievable floating-point performance. In [11], however, computation of W and Y appears to be performed on the host CPU and transferred to the GPU to transform the remaining parts of A and Q. Moreover, their timing measurements do not appear to include transfer times. Our implementation of QR decomposition is performed entirely on the GPU with the host processor remaining idle or free to complete other tasks. Our sustained floating-point performance reflects actual achievable throughput of a GPU-based QR algorithm. PCI-Express bandwidth may be dedicated to other purposes and threads on the CPU can perform other computations while the QR decomposition is completed. Nevertheless, we remain interested in Volkov and Demmel's work in this area and envision a many-core CPU-GPU solution as they have proposed to algorithms such as singular value decomposition with both serial and highly parallel parts.

Several attempts to complete the HPEC Challenge [4] on GPUs have included QR performance results. The HPEC Challenge, however, specifies the Fast Givens QR algorithm be implemented. While this was a reasonable decision befitting the novel architectures for which the HPEC Challenge was intended, Fast Givens does not match GPU architectures well. The implementation covered in this paper is unconstrained by algorithm selection and consequently offers higher performance over a large range of matrix sizes.

Baboulin [13] achieved 50 GFLOP/s for a CUBLAS implementation of QR decomposition on an NVIDIA Quadro FX 5600 for very large matrices. The Quadro FX 5600 is comparable to the GeForce 9800 GX2 used here though it has considerably more global memory and slightly higher memory bandwidth. For the the 8192×4096 problem size, their implementation achieved approximately 35 GFLOP/s. The largest matrix they tested had 19,000 rows and achieved a peak sustained performance of 50 GFLOP/s. While they employed a block Householder algorithm as we did, their use of CUBLAS rather than custom kernels misses opportunities for performance.

7. CONCLUSION

We have demonstrated an implementation of QR decomposition that runs entirely on the GPU. Restructuring algorithms so they are composed of dense operations on blocks of data takes advantage of high bandwidth to the register file, permitting each multiprocessor to approach theoretical peak performance. Structuring algorithms in terms of operations on blocks of matrices leverages the streaming architecture of CUDA-capable GPUs. Kernels were implemented with detailed knowledge of the underlying GPU architecture and offer performance beyond what is available in CUBLAS 2.0.

Our QR implementation achieves nearly 5x speedup for large matrices over Intel's MKL native QR algorithm. The algorithm selection and implementation details covered here apply to other architectures with deep memory hierarchies and data parallel arithmetic units. In the future, we hope to investigate load balancing between the CPU and the GPU, permitting high-performance libraries such as MKL to take advantage of the CPU's strengths while tasking the GPU with large block-oriented procedures. Nevertheless, we consider 143 GFLOP/s of sustained performance entirely on the GPU a notable achievement.

This implementation has been included in GPU VSIPL [12], an implementation of the Vector Signal Image Processing Library [14] for NVIDIA GPUs. VSIPL is a standardized API for developing platform-independent applications capable of taking advantage of specialized accelerator architectures. Additionally, we intend to apply the techniques and kernels developed for our QR decomposition to imple-

ment VSIPL's support for other linear algebra decompositions such as SVD, LU, and Cholesky.

8. REFERENCES

[1] NVIDIA Corporation, Santa Clara, California, *NVIDIA CUDA Compute Unified Device Architecture*, 2008.

[2] Khronos OpenCL Working Group, *The OpenCL Specification*, 2008.

[3] A. Kerr, D. Campbell, and M. Richards, GPU Performance Assessment with the HPEC Challenge, in *HPEC Workshop 2008*, Lexington, MA, 2008, MIT Lincoln Laboratory.

[4] R. Haney, T. Meuse, J. Kepner, and J. Lebak, *HPEC Challenge Overview*, MIT Lincoln Laboratory, 2005.

[5] G. Golub and C. V. Loan, *Matrix Computations*, Third ed. (Johns Hopkins University Press, Baltimore, MD., 1996).

[6] C. H. Bischof and C. V. Loan, *The WY Representation for Products of Householder Matrices*, Cornell University, Ithaca, NY, USA, 1985.

[7] D. Bindel, J. Demmel, W. Kahan, and O. Marques, *On computing Givens rotations reliably and efficiently*, ACM Trans. Math. Softw., New York, NY, USA, 2002.

[8] A. H. Sameh and D. J. Kuck, *On Stable Parallel Linear System Solvers*, Journal of the ACM, 1978.

[9] H. Hoffmann, Stream Algorithms and Architecture, Master's thesis, Massachusetts Institute of Technology, 2003.

[10] NVIDIA, *CUDA CUBLAS Library*, NVIDIA Corporation, Santa Clara, California, 2008.

[11] V. Volkov and J. W. Demmel, Benchmarking GPUs to Tune Dense Linear Algebra, in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pp. 1–11, Piscataway, NJ, USA, 2008, IEEE Press.

[12] A. Kerr, D. Campbell, and M. Richards, GPU VSIPL, in *HPEC Workshop 2008*, Lexington, MA, 2008, MIT Lincoln Laboratory.

[13] M. Baboulin, J. Dongarra, and S. Tomov, *Some Issues in Dense Linear Algebra for Multicore and Special Purpose Architectures*, Technical Report UT-CS-08-200, University of Tennessee, 2008.

[14] D. A. Schwartz, R. R. Judd, W. J. Harrod, and D. P. Manley, *VSIPL 1.3 API*, VSIPL Forum, 2008.