A Predictive Model for Solving Small Linear Algebra Problems in GPU Registers

Michael J. Anderson, David Sheffield, Kurt Keutzer
UC Berkeley: Department of Electrical Engineering and Computer Sciences
Berkeley, CA USA
{*mjanders,dsheffie,keutzer*}*@eecs.berkeley.edu*

***Abstract*—We examine the problem of solving many thousands of small dense linear algebra factorizations simultaneously on Graphics Processing Units (GPUs). We are interested in problems ranging from several hundred of rows and columns to 4×4 matrices. Problems of this size are common, especially in signal processing. However, they have received very little attention from current numerical linear algebra libraries for GPUs, which have thus far focused only on very large problems found in traditional supercomputing applications and benchmarks. To solve small problems efficiently we tailor our implementation to the GPUs inverted memory hierarchy and multi-level parallelism hierarchy. We provide a model of the GPU memory subsystem that can accurately predict and explain the performance of our approach across different problem sizes.**

As a motivating example, we look at space-time adaptive radar processing, a real-time application that requires hundreds of independent QR factorizations of small complex matrices (e.g. 240×66). For realistic matrix sizes from a standard radar processing benchmark, our implementation on an NVIDIA Quadro 6000 GPU runs $2.8\times$ to $25\times$ faster than Intel's Math Kernel Library (MKL) on an Intel Core i7-2600. For the QR factorizations of 5,000 56×56 single-precision matrices, our approach runs $29\times$ faster than MKL and $140\times$ faster than the state-of-the-art linear algebra library for GPUs. In each of these cases we are using the GPU's hardware-accelerated division and square root functions that are accurate up to 22 mantissa bits.

***Keywords*-GPGPU, Dense Linear Algebra, Modeling**

I. Introduction

Dense linear algebra routines such as solving systems of equations, least squares, and eigenvalue problems are widely used across the computational sciences. Accordingly, it is worthwhile to study these problems closely and tune the algorithms and implementation styles to new architectures. Several linear algebra libraries are currently available for hybrid CPU+GPU systems that achieve near peak matrix-multiply performance for large, dense factorizations. In this case *large* means thousands of rows and columns. Unfortunately, current GPU libraries are not able to efficiently solve small problems, where *small* means matrix sizes from 4x4 up to several hundred rows and columns. For example, in the MAGMA linear algebra library for GPUs, QR factorizations smaller than 96 columns wide are copied to the CPU and factored while the GPU sits idle. This is not an arbitrary decision on the part of these libraries. Code that can factor entire matrices on the GPU typically does not exist.

Small problems do not provide enough work or parallelism to saturate the GPU and they can be solved with relatively low latency on CPUs. However, there are a great number cases in which many independent small to medium-size problems need to be solved simultaneously. For linear algebra specifically, these problems arise in a diverse range of applications in which performance matters. One example is MRI reconstruction, which requires solving up to a billion small (8x8 or 32x32) complex eigenvalue problems, one for each voxel in an MRI image, and requires high performance for clinical applicability [16]. Space-time adaptive processing, used in real-time radar signal processing, requires hundreds of simultaneous complex QR factorizations of size 240x66 and is typically limited by the processing capabilities of the radar system [8][20]. To compute observation probabilities with a Gaussian mixture model, large-vocabulary continuous speech recognition applications multiply thousands of 79x16 matrices roughly every one-tenth second [12].

The challenge of factoring small matrices on GPUs has been pronounced in several recurring threads on the CUDA, CULA, and MAGMA online forums [1][3][4]. It was also cited by the developers of the CULA GPU linear algebra library as an important unsolved problem currently under investigation [2].

Dealing with many small (e.g. 100x100) matrices on the GPU is different than dealing with large matrices (e.g. several thousands of rows and columns). This is because these two classes of matrices benefit from a different mapping onto both the memory hierarchy and parallelism hierarchy of today's GPUs. Specifically, we distribute independent problems across different multiprocessors and solve them entirely in the multiprocessor register file. In order to understand the design space, we develop a performance model for the GPU that considers both global and intraprocessor communication and we measure relevant parameters with microbenchmarks. This model accurately predicts and explains our performance across different problem sizes.

On a realistic radar processing benchmark, our QR implementation on an NVIDIA Quadro 6000 GPU runs $2.8\times$ to $25\times$ faster than Intel's Math Kernel Library (MKL) on an Intel Core i7-2600, where each core is assigned a subset of the problems. For 5,000 56×56 single-precision QR factorizations we are $140\times$ faster than the existing GPU library that is designed and tuned for large matrices, and 29× faster than MKL. In each of these cases we are using the GPU's hardware-accelerated division and square root functions that are accurate up to 22 mantissa bits. We also show large speedups on the LU factorization and solving systems of linear equations with Gauss-Jordan elimination, although we do not pivot for stability for these problems.

To summarize, our work makes the following three contributions:

- We present solutions to efficiently solve small QR decomposition, LU decomposition, linear systems, and least-squares problems on GPUs [1]
- We present an analytical model that accurately predicts GPU performance for these problems by considering both global and intraprocessor communication
- Our performance for these small problems is up to 140× faster than the existing state-of-the-art GPU linear algebra libraries, and up to 29× faster than Intel MKL on a state-of-the-art multicore CPU

We begin by motivating our GPU performance model and deriving relevant parameters from microbenchmarks in Section II. We describe the algorithmic approaches used for our linear algebra kernels in Section III. We describe our approach for solving problems in the register file and thread block, respectively, in Sections IV and V. We discuss and analyze other existing and potential approaches to solving these problems in Section VI. We describe how to apply one of these approaches to solve a common radar processing problem in Section VII. Finally, we present conclusions in Section VIII.

	Quadro 6000
Number of multiprocessors (SIMT unit)	14
Total number of FPUs	448
Core clock rate	1.15 GHz
Max registers per FPU	64
Shared memory per SIMT unit	64 kB
Global memory bandwidth	144 GB/s
Global memory size	6 GB
Peak SP flops	1.03 TFlop/s
Peak SP per FPU	2.3 GFlop/s

Table I
SUMMARY OF THE NVIDIA GF100 CHIP AND THE QUADRO 6000

II. GPU PERFORMANCE MODEL AND MICRO-BENCHMARKS

A. Hardware Overview

The architecture and programming model of contemporary programmable GPUs has been widely documented in other work [19]; however, we provide a quick review of the salient features of the GF100 relevant to our work. The GF100 variant used in our NVIDIA Quadro 6000 has 14 multiprocesssors, each of which has 32 single-precision FPUs operating at 1.15 Ghz. The combined throughput of GF100 is 1.03 Tflops/sec. Individual tasks are executed on the multiprocessors in units called thread blocks. Threads executing in a thread block synchronize and share data through a KB scratchpad memory called "shared memory". Global memory (DRAM) is shared between all multiprocessors and has a peak bandwidth of 144 GB/sec. A small 768 KB L2 cache shared between all multiprocessors is used as a bandwidth amplifier.

Mapping small problems into the architecture is not immediately obvious due to the complex architecture. Our characterization focuses on the bandwidth and latency at each level of the GF100 memory hierarchy in a desire to understand and predict the performance of small linear algebra kernels on the GPU. To that end, we use two simple models to predict GPU performance when operands are stored in global or shared memory. Our model is based on the LogP model for distributed systems [10]. The global model is defined in Equation 1 while the shared memory model is shown in Equation 2.

$$\tau_{gbl} = \#msg \times \alpha_{glb} + msize \times \beta_{glb} + flops \times \gamma \qquad (1)$$

$$\tau_{lcl} = \#msg \times \alpha_{sh} + nsync \times \alpha_{sync} + msize \times \beta_{sh} + flops \times \gamma \qquad (2)$$

Our model has three key parameters: α, β, and γ. Total time is the sum of the costs of memory bandwidth (β_{glb} or β_{sh}), global latency (α_{glb} or α_{sh}), and time per FLOP (γ). The shared memory model also considers a synchronization cost between threads (α_{sync}). As we will show later, we do not need to overlap global and local computation for the problems considered in this paper and therefore we can consider these two models separately.

A suite of microbenchmarks were used to record values for the parameters in our model. The details of the latency benchmarks are presented in Section II-C and bandwidth benchmarks in Section II-B. Finally, Table IV presents values all the parameters used in our model.

B. Bandwidth measurements

1) Shared memory bandwidth (β_{sh}): We measure shared memory bandwidth by repeatedly issuing load instructions to shared memory and accumulating results into the register file. We believe the overhead of the add instruction used to accumulate is hidden by the superscalar pipeline in the GF100 microarchitecture. The shared memory and integer operations use independent functional units and should be able to execute simultaneously. Our source for the shared memory bandwidth benchmark is shown in Listing 1. Our experiments yield a peak of 880 GB/s from all shared memories (14) on the Quadro 6000. By comparison, the

[1]Source code is available at http://www.cs.berkeley.edu/ mjanders/

theoretical peak bandwidth for the Quadro 6000 is 1030 GB/s (14 SIMT units * 32 banks * 4 bytes per cycle * 575 MHz). We are able to achieve 85.4% of peak shared memory bandwidth; by comparison, we are able to achieve only 75% of global memory bandwidth (section II-B2)

```
shr_start = clock();
for(int i=0;i<NITRS;i++)
    for(int j=0;j<NCOPIES;j++)
        acc[j] += sMem[tid+j*256];
shr_stop = clock();
```

Listing 1. Shared memory copy

2) Global memory bandwidth (β_{glb}): The stated DRAM bandwidth of our Nvidia Quadro 6000 is 144 GB/s (384-bits * 3 GHz). As shown in Listing 2, we perform a copy of a 16 MB array to measure peak achievable global memory bandwidth. The runtime is computed by taking timestamps on the host CPU using the `gettimeofday()` function. On our Quadro 6000, our approach yields 108 GB/s, while vendor-provided copy routine, `cudaMemcpy`, yields 84 GB/s. Our simple copy code achieves 75% of peak DRAM bandwidth while `cudaMemcpy` achieves 58.3% of peak.

```
for(int i = 0; i < NUNROLL; i++)
  gbl_y[i*SIZE+idx] = gbl_x[i*SIZE+idx];
```

Listing 2. Global memory copy

Table II shows the a summary of measured bandwidth from shared memory and global memory.

	GB/s
Shared memory (per core)	62.8
Shared memory (all cores)	880
Global memory	108

Table II
BANDWIDTH FOR EACH LEVEL OF THE GF100 MEMORY HIERARCHY IN CYCLES AND MICROSECONDS

C. Latency measurements

1) Shared memory latency (α_{sh}): Our latency measurements issue a series of dependent reads in order to determine memory latency. We measure memory latency with the following procedure: recording a time stamp using the CUDA **clock()** function, performing pointer chasing to issue a series of dependent reads, recording a final time stamp, and computing the average cycles per memory operation. Our pointer chasing benchmark is similar fashion to both Volkov [21] and Wong [23]. We are unable to find GF100 shared memory latency numbers; however, our latency benchmark gives identical results to Volkov's published numbers when we run our benchmark on G80 (36 cycles).

Due to changes in the ISA, implementing a pointer chasing benchmark in shared memory on GF100 presents some minor challenges. On G80, the result of a shared memory load can be combined with an arithmetic operation. The ability to fuse an arithmetic operation with a shared memory operation eliminates address computation in pointer chasing code. The subsequent GF100 architecture eliminated the ability to fuse a shared memory operation with an arithmetic operation, introducing additional address computation arithmetic into the benchmark code. Without careful implementation of the pointer chasing benchmark, **nvcc** is unable to disambiguate load operations, and it may mistakenly emit global instructions (GF100 opcode `LD`) instead of shared memory loads (GF100 opcode `LDS`). While the address space is unified on GF100, we measured a penalty of approximately 14 cycles to access shared memory with a global memory instruction. We used Nvidia's **cuobjdump** to generate assembly listings for each microbenchmark to ensure our microbenchmarks were not encumbered by unnecessary address generation instructions.

```
pre_chase = clock();
for(int i = 0; i < NCHASE; i++)
    acc = sMem[acc];
post_chase = clock();
```

Listing 3. Shared memory pointer chasing

We experiment with implementations to address this problem. One implementation uses bytes instead of integers. Another implementation uses integers, but we subtract the overhead of the address computation. Listing 3 shows the source code skeleton for both implementations. `sMem` uses "char" for the byte implementation and the "int" datatype for the integer implementation.

We measure a latency of 18 (the arithmetic pipeline length) for the shift operation (GF100 opcode `SHL.W`) used in address calculation, while the latency of the combination of both the shared memory load and shift instruction to be 45 cycles. This approach yields 27 cycles for shared memory latency. Our other approach uses pointer chasing with bytes as operand storage. Using bytes eliminates the need for a shift at the expense of a possible penalty for byte operations. Our byte pointer chasing benchmark yields the exact same results as our other approach.

We believe 27 cycles for shared memory latency is reasonable given Nvidia's focus on the memory subsystem of GF100. In addition, we believe our methodology is sound given that our benchmark code matches existing results when run on G80.

2) Global memory latency (α_{glb}): We measure global memory latency using pointer chasing. Unlike shared memory, the latency of global memory is significantly larger than any required arithmetic instructions used for address calculation. Figure 1 shows global memory latency when

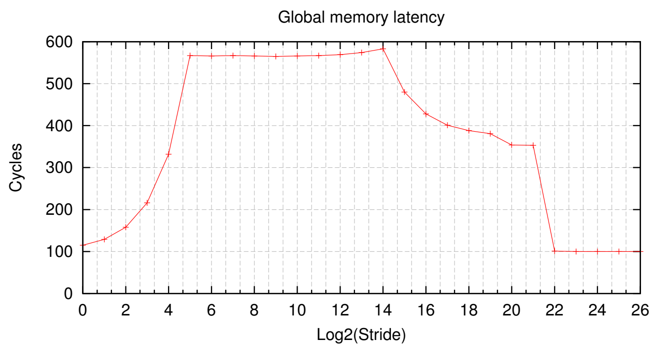


Figure 1. Global memory latency as a function of access stride

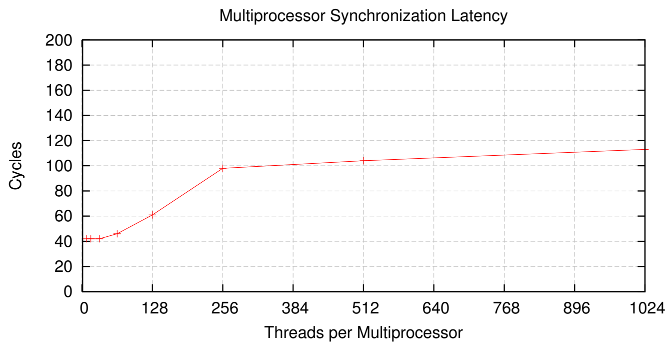


Figure 2. Synchronization latency as a function of number of threads per multiprocessor

walking through an array with a stride from 1 word to 64M words.

We do not show the source code for the global memory latency benchmark because it is significantly similar to the shared memory code shown in Listing 3.

Table III shows the a summary of the latency of each level of the memory hierarchy.

	Cycles
Shared memory	27
Global memory	570

Table III
LATENCIES FOR EACH LEVEL OF THE GF100 MEMORY HIERARCHY IN CYCLES AND MICROSECONDS

D. *Synchronization latency (α_{sync})*

The CUDA programming model allows threads executing on a single multiprocessor to synchronize through a call to **__syncthreads()**. As the synchronization latency is a function of the number of threads active in a multiprocessor we have plotted thread latency in Figure 2.

III. LINEAR ALGEBRA ALGORITHMS

We focus on the following basic linear algebra factorizations, that solve systems of linear equations, least squares,

	Quadro 6000
Global memory latency (α_{gbl})	570 cycles
Global memory inverse bandwidth (β_{gbl})	$\frac{1}{108}$ s/GB
Shared memory latency (α_{sh})	27 cycles
Shared memory inverse bandwidth (β_{sh})	$\frac{1}{880}$ s/GB
Synchronization of 64 threads in a SIMT (α_{sync})	46 cycles
Pipeline latency for FP operations (γ)	18 cycles

Table IV
RELEVANT PARAMETERS FOR OUR MODEL OF GPU PERFORMANCE

and are building blocks for many more complex algorithms [11]. All of these factorizations have arithmetic complexity O(n^3) and operate on a matrix that has O(n^2) words. As a result, these algorithms can generally be made compute-bound for most architectures and problem sizes.

A. *Gauss-Jordan Elimination*

This algorithm solves a system of equations $Ax = b$ by converting A to reduced row echelon form using row operations and applying the same operations to the vector b producing x. We do not do any pivoting in this implementation. The vector b is attached to the right side of the matrix. We proceed from left to right, scaling each row by the diagonal element and updating everything to the right of the current column with an outer product of the scaled row and current column. This algorithm performs n^3 FLOPs where n is the dimension of the matrix.

B. *LU*

The LU factorization is a FLOP-efficient method of solving linear systems. Instead of reducing the matrix to reduced row echelon form, LU produces a lower triangular matrix L and upper triangular matrix U such that $A = LU$. The solution to $LUx = b$ can then be solved using forward and backward substitution. Our implementation does not pivot so the output of the factorization is simply the lower triangular L and the upper triangular U written over the original matrix A. We proceed from left to right, scaling each column by the diagonal element and updating the Schur complement with the outer product of l, the scaled column, and u, the current row. This algorithm performs $\frac{2}{3}n^3$ FLOPs.

C. *QR*

The QR factorization decomposes an $m \times n$ matrix A into an orthogonal matrix Q and an upper triangular matrix R. This is a numerically stable way to solve least squares (when $m > n$) and systems of linear equations (when $m = n$). There are several algorithms that can be used to compute the QR factorization of a matrix. For example, one could use any of the following algorithms: Cholesky QR, Gram-Schmidt, Givens rotations, or Householder reflectors. Unfortunately, Cholesky QR and Gram-Schmidt are numerically unstable, so we are limited to using either Givens rotations or Householder reflectors. We use the Householder algorithm

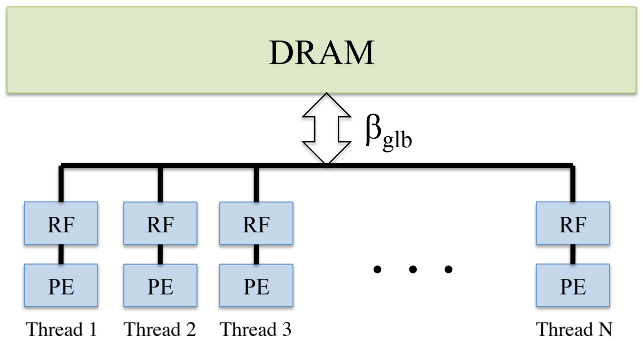


Figure 3. Simplified GPU model for the one-problem-per-thread approach. The register files (RF) of each thread's processing element (PE) are connected to a global bus which is capable of moving data at the speed of global bandwidth (β_{glb}). Once in the register file, all FLOPs performed on the data are considered to be free.

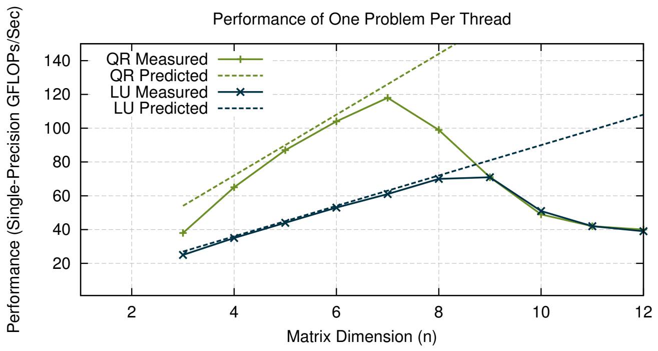


Figure 4. Performance for 64000 independent small QR and LU (No pivoting) factorizations. The dashed lines indicate model-predicted performance. For dimensions past 8 the problems no longer fit in the register file and the problems run at the speed of DRAM.

because it is consistent with LAPACK. The Householder QR algorithm performs $2mn^2 - \frac{2}{3}n^3$ FLOPs.

D. Least Squares

Least squares can be solved using QR by rewriting the the normal equations ($A^TAx = A^Tb$) in terms of Q and R ($(QR)^T(QR)x = (QR)^Tb$) that simplify to $Rx = Q^Tb$. So we simply have to find the QR factorization of A, apply Q^T to b, and solve the resulting upper triangular system for x. Note that this is more numerically stable than solving the normal equations directly. We compute Q^Tb by appending b to the right side of the matrix during the factorization. We then solve the upper triangular system using row operations. The total number of FLOPs done in our least squares solver is $2mn - \frac{2}{3}n^3 + \frac{1}{3}n^3$.

IV. One Problem Per Thread

For very small problems (e.g. $n < 16$) it is possible for each thread to store most of the matrix in its register file and solve the problem serially. Each thread works independently and there is no communication between threads. We choose to register allocate the matrix rather than relying on the L1 cache or shared memory because the register file is significantly faster and over twice the size as these other memories. Register array indices must be known at compile time, so we unroll loops using `#pragma unroll` and C++ templates. Problem sizes that exceed the maximum number of registers available per thread automatically spill into the L1 cache and eventually into DRAM. On GF100, the maximum number of registers per thread is limited to 64. We use the compiler flag `--use_fast_math` that allows for the use of hardware reciprocal and square root functions that are accurate up to 22 mantissa bits [18]. In this approach, the median performance penalty for not using these hardware functions is 5.6%.

To understand the performance of this approach we consider the simplified model pictured in Figure 3 and described in Section II. We assume that FLOPs are free ($\gamma = 0$) and the register file is infinite. We only count the bandwidth cost between DRAM and register files, as specified by the global DRAM bandwidth (β_{glb}). We also choose to ignore global DRAM latency ($\alpha_{glb} = 0$) since we assume this latency is sufficiently hidden through multi-threading. Expected performance is simply the product of the problem's arithmetic intensity and the global DRAM bandwidth [22].

A problem's arithmetic intensity is the total number of FLOPs performed divided by the total number of words moved. For example, a 7×7 single-precision QR factorization performs $\frac{4}{3}mn^2 - \frac{2}{3}n^3 = 457$ FLOPs. The entire matrix must be read and written which generates DRAM traffic of $2 \times 7 \times 7 \times 4$ bytes $= 392$ bytes. Thus, the arithmetic intensity of this problem is $\frac{457}{392} = 1.17$ FLOPs/byte. Our measured bandwidth is $\beta_{glb} = 108$ GBytes/s. We can expect 1.17×108 GBytes/s $= 126$ GFLOPS, which roughly matches the measured performance.

Figure 4 shows the expected and measured performance of both LU and QR factorizations for problems of size $n = 3$ to 12 with the one-problem-per-thread approach. Performance follows arithmetic intensity nearly perfectly for both LU and QR until $n = 8$, at which point the problem no longer fits in the register file and spills to L1 and DRAM. For problems where the matrix fits in the register file ($n < 8$) we can say that the bandwidth cost of reading in and out the matrix presents an upper bound on performance. This implementation is optimal in that case.

We could extend the one-problem-per-thread approach to larger problems and potentially sustain the same high performance by using blocked algorithms within a thread [13]. However, for a single-level memory hierarchy such as the GPU's, the performance of this approach would even then face theoretical limits determined by the amount of global bandwidth and the amount of local storage per thread and regardless of the blocking strategy or algorithm [6]. Another approach is to assign multiple threads to work together to

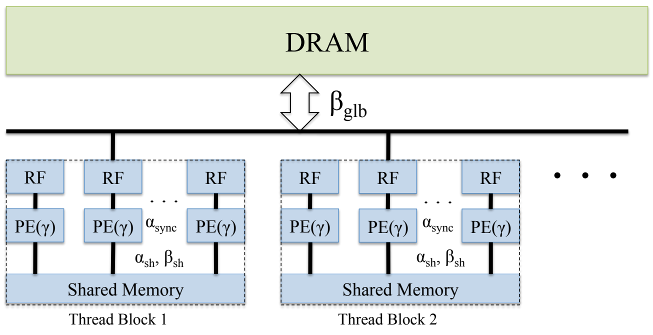


Figure 5. Complete GPU model for the one-problem-per-block approach. Each thread block can access DRAM at the speed of global DRAM bandwidth (β_{glb}). Once the data is inside the thread block, it can be stored in the thread's register files (RF) or in shared memory. All threads can access shared memory with a bandwidth (β_{sh}) and latency (α_{sh}). A FLOP takes γ cycles and the operands must be in the register file.

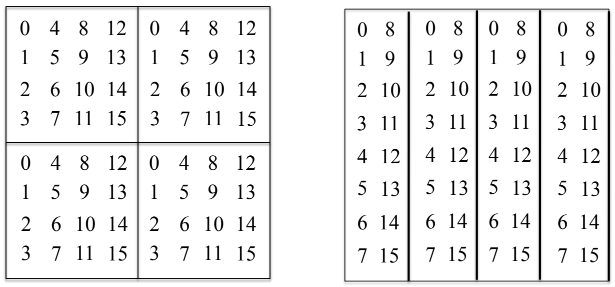


Figure 6. A 2D cyclic layout (left) and 1D row cyclic layout (right). Numbers indicate which thread owns each matrix entry.

solve one problem. This approach can increase the amount of local storage per problem and thus the arithmetic intensity. This approach is analyzed next.

V. One Problem Per Block

In the previous section we considered the case in which every thread on the GPU loaded a small matrix into its register file and performed a factorization locally. However, if multiple threads work together in a block (e.g. 64 or 256 threads), then we can factor larger matrices without having to communicate with DRAM. One benefit of larger matrices is increased arithmetic intensity. On GF100, 256 threads can store a 112x112 single-precision matrix in a distributed fashion with each thread storing a 7x7 sub-matrix. A QR factorization on a 112x112 matrix performs 1.87 MFLOPs while moving only 100 KBytes of data to and from DRAM. Following the simple bandwidth-only model from Section IV tells us the potential performance of this problem is over 2 TFLOPS, which is beyond the max theoretical arithmetic throughput for the chip.

Since the problem is no longer bandwidth-constrained, the costs of doing arithmetic, communicating data between threads in a block, and synchronizing within a block are clearly going to limit performance. These costs are captured by the γ, α_{sh}, β_{sh}, and α_{sync} parameters, respectively. The complete model that is used to understand this approach is drawn in Figure 5.

Next, we describe the implementation details of the one-problem-per-block approach. This includes how the matrix is distributed across threads, how threads communicate, and how computation is divided among threads. Results and analysis with the performance model will follow that.

A. Distributed Data Layouts

Even though threads have access to a global memory space, the thread block is essentially a distributed system. Each thread represents a machine in the distributed system and each thread's register file represents the private memory. GF100 is a load-store architecture, so all data must be communicated to the register file even for operations with shared memory operands. As in any distributed system, a decision must be made about data layout. We consider three classic distributed data layouts: 1D row cyclic, 1D column cyclic, 2D cyclic. A sketch of these layouts is shown in Figure 6.

The simplest data layouts are 1D row or column cyclic. In a 1D row cyclic layout each thread is assigned a row that it stores locally in its register file. If there are more threads than rows, then a row is divided between several threads, preferably in a way that divides the number of elements per thread evenly. 1D column cyclic is similar except columns are assigned instead of rows. The traditional advantages of 1D layouts are that either row or column operations (e.g. computing a Householder reflector) can be carried out within a thread without any communication. One major disadvantage is the load imbalance that occurs in factorizations that proceed from left-to-right, as one thread must drop out after each column is processed.

In a 2D layout, threads own elements from several rows and several columns. We consider a 2D cyclic layout in which elements are distributed evenly throughout the matrix. The 2D cyclic layout mitigates the load imbalance problem inherent in 1D-layout one-sided factorizations, but it introduces communication between ($\sqrt{p}$) threads for both row and column reductions. This can be seen as a compromise between the 1D row and 1D column layouts.

Figure 7 shows the performance we achieved using 1D row and column layouts and 2D cyclic on solving systems of equations using a Householder QR factorization followed by Gaussian elimination of the upper triangular result. Due to the large amount of column-wise communication inherent in the Householder QR algorithm, one expects the 1D column-cyclic layout to be considerably faster than the 1D row-cyclic layout. The 2D layout dominates 1D layouts in all tested cases, so we will use this layout by default for the remainder

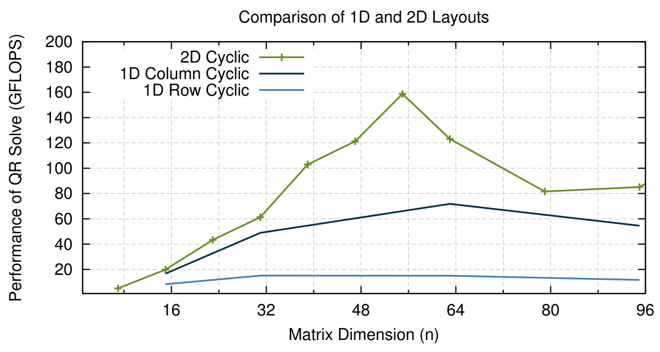


Figure 7. Solving 10,000 single-precision linear systems using QR with one-problem-per-block approach and various ways of laying out the data in the register file.

of the paper. The MAGMA BLAS library for GPUs also uses a 2D register layout for their matrix-matrix multiply routine [17].

B. Implementation Details

We implemented the factorizations described in Section III using the one-problem-per-block approach for both square and non-square matrices. The problem dimensions are hardcoded because register file indices must be known at compile time. However, each implementation is parametrized with compile-time constants. This allows the same code to be compiled for problems of several different sizes and shapes.

The matrix is stored in the register file in a 2D cyclic layout. The following code loads the matrix from DRAM to the register file. WREG and HREG are the width and height, respectively, of the registered sub-matrix. RDIM is $\sqrt{p}$, where p is the number of threads in a block. The global pointer d_A has already been offset to point to the correct matrix for this block and the correct starting location in the matrix for this thread. This code achieves over 90 GB/sec. for most matrix sizes and thread configurations despite some non-contiguous memory accesses.

```
#pragma unroll
for(int j = 0 ; j < WREG ; j++)
  for(int i = 0 ; i < HREG ; i++)
    A[i][j] = d_A[i*RDIM + j*RDIM*lda];
```

Listing 4. Loading a matrix into the register file

To begin a factorization, we must modify a column, typically either scaling it or normalizing it, and then copy that column to shared memory. There it can be used to update the trailing matrix. Scaling the vector requires a broadcast of the scaling factor. Normalizing the vector requires a reduction followed by a broadcast of the norm. The following code, found in LU and Gauss-Jordan, creates a scale factor and assigns it to *scale*. This is a shared memory variable. The variable *N* indicates the panel being factored (0 through $\frac{n}{\sqrt{p}}$). Variable *j* is the location of the column being scaled in the panel. Variable *col* is the thread's column location in the column panel, and *tid* is the thread's row location in the row panel. Alternatively, since threads are laid out in a $\sqrt{p} \times \sqrt{p}$ grid, (*tid*,*col*) are the thread's coordinates in that grid.

```
if(tid == j && col == j) {
  if(A[N][N]!=ZERO){scale = ONE/A[N][N];}
  else {scale = ZERO; *not_solved = 1;}
} __syncthreads();
```

Listing 5. The thread on the diagonal determines the scaling factor and assigns it to shared memory

Scaling and copying the column vector from the matrix in the register file to shared memory looks like the following.

```
if(col == j) {
  #pragma unroll
  for(int ii = N ; ii < HREG ; ii++)
    l[col+ii*RDIM] = A[ii][N] * scale;
}
```

Listing 6. Scaling while extracting a column from the matrix in the register file to a vector l in shared memory.

Finally, updating the trailing matrix involves matrix-vector operations such as matrix-vector multiply and rank-1 update. A matrix-vector multiply requires many reductions across threads and is not shown here. The following is a rank-1 update in which two shared memory vectors l and u are broadcast to update the trailing sub-matrix in A.

```
#pragma unroll
for(int ii = N ; ii < HREG ; ii++)
  for(int jj = N ; jj < WREG ; jj++)
    A[ii][jj] -= l[tid+ii*RDIM]
                 * u[col+jj*RDIM];
```

Listing 7. Rank-1 update of a by shared vectors l and u

The column operation and trailing matrix update are encapsulated in a loop over columns from 0 to RDIM. This loop completes the factorization of an entire panel and updates the trailing matrix. To factor more than one panel, we encode the panel factorization and trailing matrix update as a C++ template and use tail recursion to unroll the entire factorization. Unrolling is necessary because all accesses to the register file must be known at compile time. Register file arrays cannot be indexed by loop indices, for example. N is a template parameter indicating the current panel that begins at zero. The base case for the recursion is when N = WREG and the factorization is complete.

C. Measured Performance

The average number of cycles measured for a 56x56 single-precision LU and QR factorization for the one-

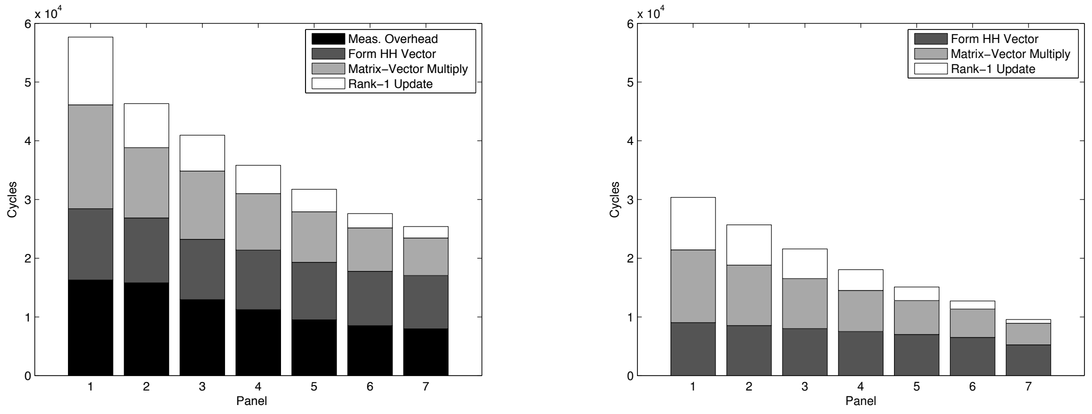


Figure 8. The number of cycles spent in each panel of QR measured (left) and modeled (right) for a 56x56 single-precision matrix broken down between the three main operations and measurement overhead. As the factorization proceeds the matrix becomes smaller so each panel takes less time.

	Load	Compute	Store
LU	8800	68250	8740
QR	9120	150203	9762

Table V
CYCLE COUNTS FOR 56X56 LU AND QR DECOMPOSITIONS

problem-per-block approach is shown in Table V. The cycles are broken down between memory access time and on-chip compute time. For the 56x56 size, the GPU is executing eight thread blocks per multiprocessor for a total of $14 \times 8 = 112$ problems simultaneously across the entire chip. Each thread block has 64 threads. The measured 9000 cycles for reading and writing seems to indicate that the warp scheduler is overlapping some global communication with computation so that fewer than 8 thread blocks are competing for memory bandwidth at a time. If all 8 thread blocks were able to load the matrix in 9000 cycles this would mean we were achieving nearly double the peak bandwidth that we measured earlier.

Again, we used the compiler flag `--use_fast_math` which allows for the use of hardware reciprocal and square root functions that are accurate up to 22 mantissa bits [18]. For this approach, not using the hardware functions resulted in a median performance penalty of 30%.

The measured compute times are broken down further in Figure 8 into panels and operations within each panel. Each panel has $\sqrt{p}$ columns, so there are 7 panels in a 56x56 matrix with 64 threads. With each new panel the matrix becomes smaller by $\sqrt{p}$ rows and $\sqrt{p}$ columns, so the total time needed to factor that panel and update the trailing matrix decreases over time. Figure 9 also contains our model's estimated cycle counts for the same operations. The next section will explain how these estimates were generated.

D. Modeling Performance

We estimate the performance of LU and QR using the one-problem-per-block approach by counting the number of FLOPs, accesses to shared memory, and thread synchronizations present in our implementation. We assume there is no overlap of either shared or global memory communication and computation. Code essentially similar to those of our LU and QR compose the Gauss-Jordan and least squares solvers. Therefore, we will not analyze them here.

We are not attempting to define upper or lower bounds for the performance on these problems. Rather, we are just trying to understand and predict why our specific implementation goes the speed it does and where specifically the time goes. There are certainly other ways to solve these problems in the thread block that may perform more or less communication.

The following paragraphs describe how we arrived at the computation and communication estimates for our LU and QR implementations. The final estimates are shown in Table VI. For the purposes of this analysis, a floating-point multiply-add is counted as one γ because the pipeline is dual-issue.

This LU implementation can be broken down into two phases: the column operation and the trailing matrix update. Each happens $n-1$ times where n is the number of columns in the matrix. To compute and communicate the scale factor, the column operation requires a division followed by a shared memory write and a synchronization. Then there is another shared memory read of the scale factor and N FLOPs to scale the column, where N is the height of the current column divided by the $\sqrt{p}$ threads in that column, rounded up if necessary. Finally, the column l and row u are written into shared memory, which requires $2*N$ shared memory accesses and a synchronization at the end. These writes cannot happen in parallel because the thread on the diagonal owns some of both l and u. The panel update is a rank-1 update. Each thread must read N elements from both l and u for a total of $2 * N$ shared memory accesses. Then each thread performs N^2 FLOPs and 1 synchronization at the end.

For the QR factorization we choose to do serial reductions instead of parallel. Each reduction is across $\sqrt{p}$ threads so $cost_{red}$ will be $(1+\sqrt{p})\beta + \sqrt{p}\gamma$. For the matrix-vector multiply we assume that there are at least as many threads as columns so the total cost will be the $cost_{red}$. A column norm reduction will have the same cost as a matrix-vector multiply reduction even though only one thread is needed for the column norm reduction.

The column operation of the QR factorization requires N FLOPs plus a reduction by thread 0 to compute the norm. Then thread 0 computes the scale factor that does a square root, two divides, and two multiplies, followed by one shared memory write of the scale factor. Then the column is scaled and written to shared memory, which requires a read of the scale factor, N FLOPs, and N writes to shared memory followed by synchronization. The trailing matrix update does N shared memory reads to get the Householder vector. Then there is a matrix-vector multiply with is N^2 FLOPs, a synchronization, a reduction, and another synchronization. Next there is a rank-1 update which involves N reads from shared memory of the matrix-vector multiply result, N^2 FLOPs, and finally a synchronization at the end. There are a total of $(n-1)$ column and trailing matrix operations in QR.

LU Estimates	
Column	
$\gamma_{div}\alpha_{sync}$	Thread 0 compute scale factor
2β	Write and read scale factor
$N\gamma$	Scale l vector
$2N\beta + \alpha sync$	Write l & u to shared
Trailing Matrix	
$2N\beta$	Read l & u from shared
$N^2\gamma + \alpha_{sync}$	Rank-1 update

QR Estimates	
Column	
$N\gamma$	Column norm
$(1+\sqrt{p})\beta + \sqrt{p}\gamma)$	Thread 0 norm reduction
$\gamma_{sqrt} + 2\gamma_{div} + 2\gamma$	Thread 0 compute scaling factor
2β	Write and read scale factor
$N\gamma + N\beta + \alpha_{sync}$	Column scale & write to shared
Trailing Matrix	
$N\beta$	Read Householder vector
$N^2\gamma$	Matrix-vector multiply
$2\alpha_{sync} + (1+\sqrt{p})\beta + \sqrt{p}\gamma$	Matrix-vector multiply reduction
$N\beta + N^2\gamma + \alpha sync$	Rank-1 update

Table VI
ESTIMATES OF THE FLOPS, SHARED MEMORY COMMUNICATION, AND SYNCHRONIZATION DONE IN LU AND QR

We calculate total cycles by plugging in the parameter values from Section II and adding the cost of reading and writing the matrix from DRAM. We use the division and square root cycle times from a previous benchmarking paper on the GT200 architecture [23]. To derive global GFLOPS estimates from the number of cycles predicted by the model, we multiply the FLOPs done by a single thread block by the total number of blocks executed simultaneously on the chip. Then we divide that by the number of cycles, and multiply by the number of cycles per second. The number of simultaneous blocks is given by the CUDA occupancy calculator.

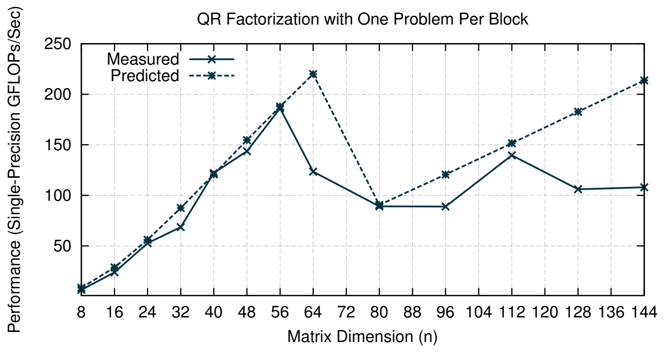
Figure 9 shows the performance of LU and QR using the one-problem-per-block approach. The dashed lines indicate the performance predicted by the model including the bandwidth cost to get the matrix to and from DRAM. Again we can see the effect of register spilling when the number of registers per thread meets or exceeds 64. This spilling is not captured in the model. The abrupt change in performance at $n = 80$ occurs because we switch from using 64 threads per block to 256 threads per block. This reduces the number of simultaneous blocks per multiprocessor from 8 to 2. Note that the number of threads must be a perfect square as a consequence of the 2D layout.

VI. Other Approaches

A. Hybrid CPU+GPU Blocked

The predominant approach taken by GPU-enabled linear algebra libraries such as MAGMA and CULA is the hybrid CPU+GPU blocked approach [5][14]. Panels are factored on the CPU and sent to the GPU where the trailing matrix is updated using matrix-matrix multiply. The trick to this approach is to overlap the communication between CPU and GPU such that the entire problem can run at the speed of the GPU matrix-matrix multiply routine. The other benefit of this approach is that the only GPU code that needs to be written and optimized is the matrix-multiply routine. The downside of this approach is clear for small or skinny problems. In this case the trailing matrix is much smaller so the majority of the computation, if not all, occurs on the CPU. The panel width in the current MAGMA release is 96 so all problems less than 96 wide are done entirely on the CPU.

Figure 10 shows the performance of our approach compared to MAGMA, the state-of-the-art linear algebra library for GPUs. The library does not provide the ability to run multiple problems simultaneously so we put a loop around the function call and run each problem sequentially. We call the routines in which data starts and ends on the GPU. As this plot illustrates, MAGMA is inefficient for small problems and the overall design space is not flat. For very large problems MAGMA is very fast, for reasons we describe below. However, for small problems our implementation up to two orders of magnitude faster.



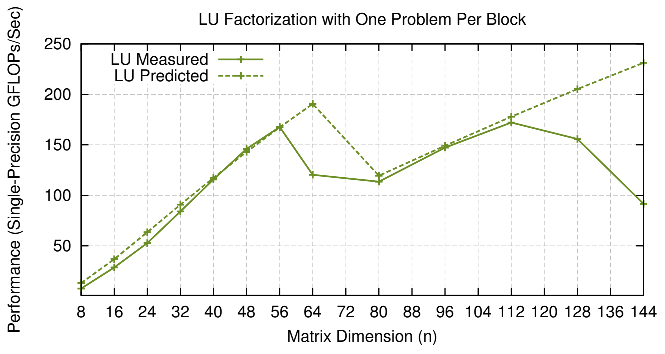
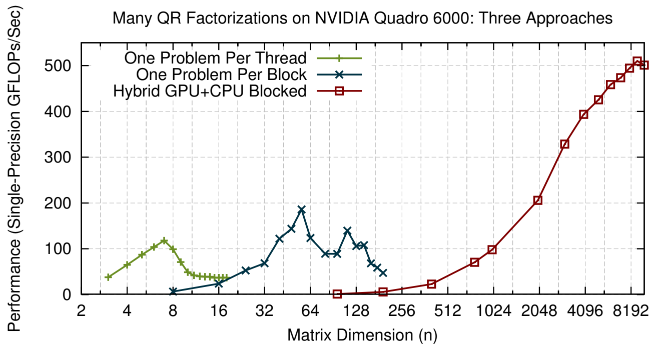


Figure 9. The performance of 8,000 LU and QR factorizations with the one-problem-per-block approach. The dashed line indicates the model-predicted performance. The sharp drop from 64 to 80 happens because we switch from 64 to 256 threads. The false predictions at 64 and above 112 are due to register spilling, which our model does not consider.



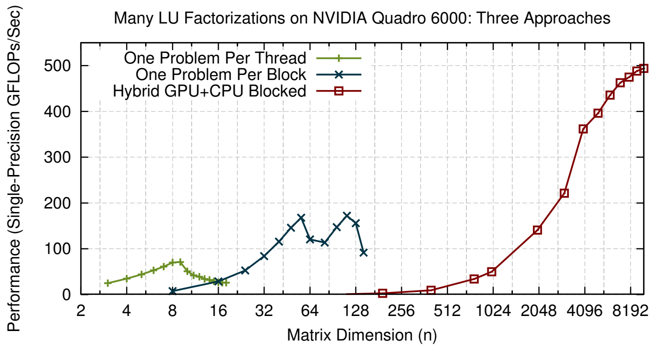
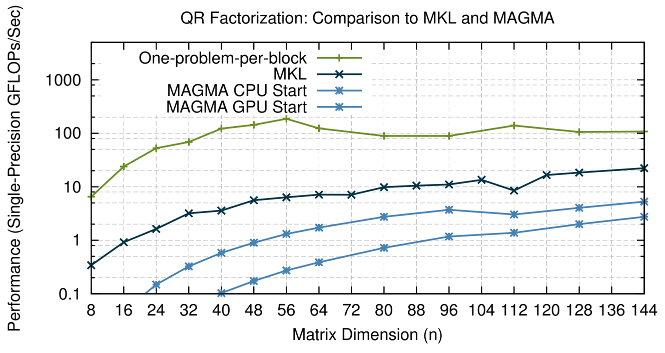


Figure 10. The design space for different sized problems is not flat. Our two approaches perform well for thousands of small problems and the MAGMA Hybrid GPU+CPU blocked approach performs well for single large problems.



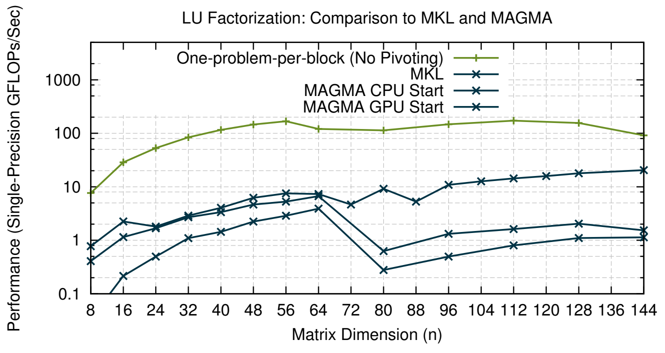
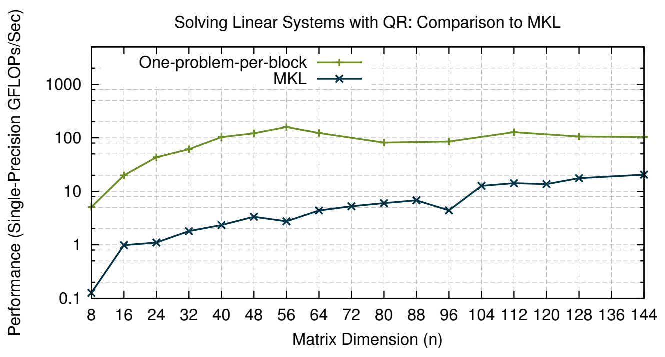


Figure 11. Comparison of the one-problem-per-block approach to Intel's MKL and MAGMA GPU linear algebra library for 8,000 LU and QR factorizations. Note our implementation does not pivot during Gaussian Elimination and both MKL and MAGMA do pivot. However, the matrices tested were diagonally dominant so no pivoting was necessary.

B. *Intel Math Kernel Library (MKL)*

Figures 11 and 12 show the performance of our one-problem-per-block approach compared to both MAGMA and Intel's Math Kernel Library (MKL) on an Intel Core i7-2600. We distribute the problems evenly across all four cores using pthreads. MAGMA performance is shown for two routines. In the first case the data starts on the CPU. In the second case the data starts on the GPU. The CPU-start is faster because MAGMA solves these problems mostly on the CPU anyway.

These are not ideal comparisons because both MAGMA and MKL do partial-pivoting for LU, while ours does not. However, we ran these examples on diagonally-dominant matrices so no pivoting was necessary. We also use the hardware reciprocal and square root on the GPU, while both



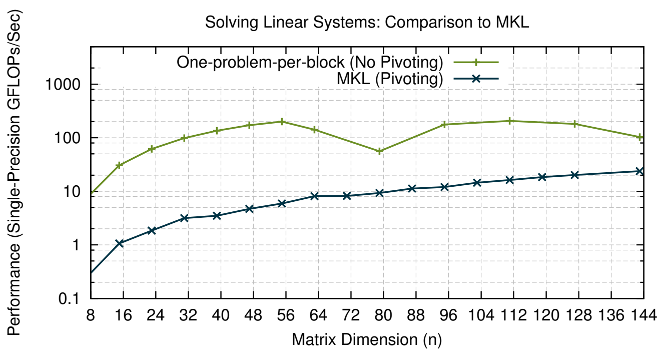


Figure 12. Comparison of the one-problem-per-block approach to Intel's MKL for solving 8,000 linear systems with QR and Gaussian Elimination. Note our implementation does not pivot during Gaussian Elimination.

MAGMA and MKL use full-precision functions on the CPU.

C. Using CUBLAS and CUDA Streams

In our implementations we mapped small problems to the GPU memory hierarchy, first to the thread level (one-problem-per-thread), then to the block level (one-problem-per-block). It is also possible to solve these problems at the global level. In this case we can use CUBLAS to do column norms and scales, matrix-vector multiplies, rank-1 updates, and trailing matrix-matrix updates. Since CUBLAS allows for multiple streams we can even call these operations in parallel over multiple problems. However, this approach does not take advantage of the memory hierarchy except for the trailing matrix-matrix multiplies, which are small for these problem sizes. Furthermore, it is practically difficult to get the current GPU to do small CUBLAS routines, such as column norms, in parallel in a fine-grained manner. As a result, our blocked implementation of QR using CUBLAS showed no benefit from using multiple streams. We could achieve better performance solving the problems sequentially on the CPU.

VII. Application: Space-Time Adaptive Radar Processing (STAP)

Size	# Matrices	GPU GFLOPS	MKL GFLOPS	Speedup
80x16	384	134	5.4	25×
240x66	128	99	36	2.8×
192x96	128	98	27	3.6×

Table VII
Results on single-precision complex QR factorizations from RT_STAP benchmark

Space-time adapative processing is a computationally demanding radar processing algorithm. STAP has many computational phases; however, the most demanding phase is multiple simultaneous complex QR decompositions. The challenge of obtaining real-time performance has resulted in several application specific parallel machines[15], [7].

While the size of the QR decomposition depends on the specifics of the radar system, the official MITRE RT STAP benchmark [8] specifies several sizes for the complex QR decomposition which we use for benchmarking. We also test the 192x96 size which was used in a paper for the Imagine stream processor [9]. We consider only single precision complex datatypes. The total number of FLOPs done is $8mn^2 - (8/3)n^3$.

Our performance is shown in Table VII. The 80x16 problem fits in a single thread block so it is relatively straightforward to solve. On the other hand, the larger size does not fit in a single thread block so we employ a sequential tiled QR factorization algorithm similar to the approach in the PLASMA multicore linear algebra library [5]. The 240x66 problem performs somewhat more slowly than the others because it does not fit well in our block sizes so some of the register file space is being wasted.

VIII. Conclusions

We examined the problem of doing many thousands of small dense linear algebra factorizations simultaneously on a GPU. Very small problems (e.g. $n < 16$) can be efficiently solved by assigning one problem per thread and factoring each problem locally in the register file. This approach is optimal for problems that fit entirely in the register file. For larger problems it makes sense to assign an entire thread block to a single problem because this gives higher arithmetic intensity and requires fewer problems to saturate the GPU. Tiled algorithms can be used to solve problems that are too large to fit in a single thread block's register file.

Our model considers both global (DRAM-to-processor) and local (interprocessor) communication. It is accurate for the LU and QR factorizations we analyzed. These problems have specific qualities that are motivated our specific model. First, we assume that global communication happens separately from local communication. In our case it does because the factorization takes so many more cycles than loading or storing the matrix. Second, there is a significant amount of inter-thread communication that is unlike some previously-studied computationally intense kernels such as matrix-multiply. We believe this model is particularly well suited for other problems that have these two properties.

The NVIDIA *streams* interface is a step toward a new style of GPU computing where a diverse set of small things can be happening simultaneously instead of only handling large monolithic data parallel computations. This appears to be an evolutionary process, as the current level of support for *streams* is not fine-grained enough to support such a programming style. However, as the support for *streams* improves, it may encourage programmers to think more in terms of solving problems individually in thread blocks than with entire grids of thread blocks. The costs captured by our model, such as shared memory bandwidth and synchronization latency, are relevent and necessary considerations for this style of programming.

Acknowledgment

The authors would like to thank Vasily Volkov for his help understanding the GF100 architecture.

References

[1] CUDA forums: Lots of small matrices. http://forums.nvidia.com/index.php?showtopic=188430. Accessed: 09/26/2011.

[2] CULA - a hybrid GPU linear algebra package. http://nvidia.fullviewmedia.com/gtc2010/0923-a3-2153.html. NVIDIA GPU Technology Conference 2010.

[3] CULA forums: Batch level parallelism. http://www.culatools.com/forums/viewtopic.php?f=14&t=774. Accessed: 09/26/2011.

[4] Magma forums: parallel execution of multiple magma dsyevd. http://icl.cs.utk.edu/magma/forum/viewtopic.php?f=2&t=204&p=848&hilit=streams#p848. Accessed: 09/26/2011.

[5] 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*, volume 180, page 012037. IOP Publishing, 2009.

[6] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in linear algebra. *Arxiv preprint arXiv:0905.2485*, 2009.

[7] P.B. Bhat, Y.W. Lim, and V.K. Prasanna. Issues in using heterogeneous hpc systems for embedded real time signal processing applications. In *rtcsa*, page 134. Published by the IEEE Computer Society, 1995.

[8] K.C. Cain. Rt_stap: Real-time space-time adaptive processing benchmark. Technical report, DTIC Document, 1997.

[9] S. Chatterji, M. Narayanan, J. Duell, and L. Oliker. Performance evaluation of two emerging media processors: VIRAM and Imagine. 2003.

[10] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. Von Eicken. *LogP: Towards a realistic model of parallel computation*, volume 28. ACM, 1993.

[11] J.W. Demmel et al. *Applied numerical linear algebra*, volume 150. SIAM Philadelphia, PA,, USA, 1997.

[12] P.R. Dixon, T. Oonishi, and S. Furui. Harnessing graphics processors for the fast computation of acoustic likelihoods in speech recognition. *Computer Speech & Language*, 23(4):510–526, 2009.

[13] J.J. Dongarra, J. Du Croz, S. Hammarling, and I.S. Duff. A set of level 3 basic linear algebra subprograms. *ACM TOMS*, 16(1):1–17, 1990.

[14] J.R. Humphrey, D.K. Price, K.E. Spagnoli, A.L. Paolini, and E.J. Kelmelis. CULA: hybrid GPU accelerated linear algebra routines. In *SPIE Conference Series*, volume 7705, page 1, 2010.

[15] Wei keng Liao, A. Choudhary, D. Weiner, and P. Varshney. Multi-threaded design and implementation of parallel pipelined stap on parallel computers with smp nodes. pages 448 –452, apr 1999.

[16] M. Murphy, K. Keutzer, S. Vasanawala, and M. Lustig. Clinically feasible reconstruction time for l1-spirit parallel imaging and compressed sensing mri. *ISMRM'10*, 2010.

[17] R. Nath, S. Tomov, and J. Dongarra. An improved magma GEMM for Fermi GPUs. *ICL, University of Tennessee, Tech. Rep*, 2010.

[18] J. Nickolls and W.J. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56–69, 2010.

[19] Nvidia. CUDA programming guide. 2009.

[20] Michael Parker. Radar basics. http://www.eetimes.com/design/programmable-logic/4216104/Radar-basics---Part-1. Accessed: 09/27/2011.

[21] V. Volkov and J.W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1 –11, Nov. 2008.

[22] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[23] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*, pages 235 –246, March 2010.