Science & Technology Facilities Council

Preprint

RAL-P-2012-002

A fast triangular solve on GPUs

JD Hogg

July 2012

Submitted for publication in SIAM Journal on Scientific Computing

RAL Library
STFC Rutherford Appleton Laboratory
R61
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: libraryral@stfc.ac.uk

ISSN 1361- 4762

A fast triangular solve on GPUs

J. D. Hogg

July 5, 2012

Abstract

The level 2 BLAS operation _trsv performs a dense triangular solve, and is often used in the solve phase of a direct solver following a matrix factorization. With the advent of manycore architectures the importance of this memory-bound kernel is increasingly important, particularly for sparse direct solvers used in optimization applications.

In this paper, a high performance implementation of the triangular solve is developed through a careful analysis of theoretical and practical bounds on the possible performance. This implementation outperforms the the CUBLAS by a factor of 5–15.

1 Introduction

The solution of a dense triangular system $Lx = b$ (or $L^T x = b$) through forward (or backward) substitution is implemented as a level 2 BLAS operation _trsv. The diagonal entries of the system can either be unit or non-unit, and the solution is typically performed in place.

This operation is of particular interest in the solution of linear equations using direct methods. Users of such algorithms often need to solve the same system repeatedly with different right-hand sides. If iterative refinement is also used, the triangular solve can be performed tens or hundreds of times for each matrix factorized. Further, a single solve using a sparse factor will typically employ multiple calls to the triangular solve routine for small dense submatrices of dimension ranging from one to several hundred, and occasionally several thousand.

The non-transposed, unit-diagonal algorithm is straightforwardly described in the following pseudocode.

Input: Lower-triangular $n \times n$ matrix L, right-hand-side vector x.
for $i = 1, n$ **do**
  $x(i+1:n) = x(i+1:n) - L(i+1:n, i) * x(i)$
end for
Output: solution vector x.

Asymptotically, for large n, a single multiply-add is performed for each read (accesses to L will not be cached, accesses to x should be). Therefore, on modern computing hardware, the operation is memory-bandwidth bound. For small matrices (of interest in the sparse case) the algorithm will instead be memory-latency bound. For example, an NVIDIA C2050 can deliver approximately 9 double precision operands per clock cycle from main memory if the bandwidth is fully saturated. If the cache latency is, say, 200 cycles then a straightforward implementation will require $n > 1800$ for multithreading to fully hide the latency. For a matrix with $n = 32$, at least 195 cycles per column might be spent waiting for data to arrive.

This memory-latency bound can be overcome by the separation of the data request from the data usage. Done correctly, the main-memory latency penalty is then incurred only once. This allows a 5–10 times performance increase to be achieved over the current version of the CUBLAS [3].

In this paper we consider the lower triangular, non-transpose, variant of _trsv() where L has unit diagonal and x has unit stride. The transpose algorithm is also briefly addressed. Non-unit diagonal and non-unit stride of x can be trivially accommodated within the algorithms presented, but is omitted here

for clarity. The upper triangular versions can be derived through small changes to the lower triangular algorithms.

The remainder of this paper is set out as follows. Section 2 describes experiments and implementation on small blocks for $n \leq 128$ on a single Streaming Multiprocessor (SM). Section 3 extends this work to larger blocks utilising the full power of the GPU, and demonstrates advantages to using global memory rather than kernel launches for synchronization. The use of explicit inversion methods is addressed in Section 4. Finally, conclusions are presented in Section 6.

2 Small matrices

This section concerns the implementation of `_trsv()` using a single thread block, and hence a single SM. This will provide the best performance for small matrices and can be combined with an efficient matrix-vector product (`_gemv()`) to achieve good performance for larger matrices.

The size of these small blocks has been determined through preliminary experimentation, but can be motivated by looking at the amount of storage and data movement required for varying size of block. The following data looks at the amount of storage required to read the matrix fully into memory for multiples of the warp size (32 on a C2050).

	Half matrix		Full matrix	
n	$nz(L)$	mem	$nz(L)$	mem
32	528	4.1 KB	1024	8.0 KB
64	2080	16.3 KB	4096	32.0 KB
96	4656	36.4 KB	9216	72.0 KB
128	8256	64.5 KB	16384	128.0 KB

Max shmem/block: 48KB
Max reg/block: 32768

As all threads of a warp execute the same instruction stream, there is no requirement for synchronization between them except to ensure that values are passed between them using `volatile` variables. If multiple warps are used, then synchronization using `__syncthreads()` is necessary. Microbenchmarking indicates that the overhead of such synchronization is 40–90 clock cycles depending on the number of threads involved. As a version of `_trsv()` will be developed that requires fewer than 150 clock cycles per iteration, the cost of using multiple warps to execute dependant operations within the same or consecutive columns is too high to yield a feasible alternative to the algorithms presented below.

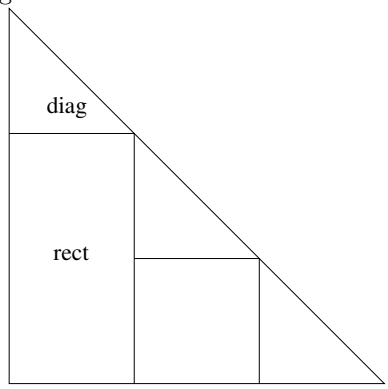
Two obvious mechanisms exist for pre-caching matrix values to avoid incurring the global memory latency for each column. The first is to use shared memory, and the second to use registers.

The shared memory variant is shown as Listings 1 and 2. The matrix is divided into block columns as shown in Figure 1. For each block column from left to right, the diagonal block (marked diag in the figure) is first solved using a single warp, then a matrix-vector product is performed with the rectangular block below it (marked rect) to update the entries for the next block column using one thread per row.

The first listing shows a subroutine `dblkSolve()` that performs the solve on the diagonal block, of size `blkSize` $\times$ `blkSize`. As a single warp is used, `blkSize` $\leq$ `warpSize`. Best performance is obtained if `blkSize` = `warpSize` (= 32 for the C2050). The second listing shows both how the rectangular block's matrix-vector product is performed and how memory is managed to avoid incurring global-memory latency waits in `dblkSolve()`. A large team of non-computing threads (those with `threadIdx.y` > 0) is used to load the matrix into a shared memory cache during the stalls in execution of `dblkSolve()`.

The performance of `dblkSolve()` is critical to the overall performance, as each column must be solved in sequence. Given that `a` (used in place of `l` to ensure typographic distinction from the number `1`) and `x` are stored in shared memory, no accesses to global memory are required. Each iteration, stalls occur due to shared memory latency (approx 35 cycles per access) and pipeline depth (approx 5 cycles per dependant instruction). Timing indicates that in practice each iteration takes on average of 135 cycles (corresponding

Figure 1: Partitioning of matrix into block columns for small matrix solves



Listing 1: 32×32 solve

```
template <int blkSize>
void __device__ dblkSolve(const double *a, int lda, double &val) {

    volatile double __shared__ xs;

#pragma unroll 16
    for(int i=0; i<blkSize; i++) {
        if(threadIdx.x==i) xs = val;
        if(threadIdx.x>=i+1)
            val -= a[i*lda+threadIdx.x] * xs;
    }
}
```

to 12 PTX instructions and 2 shared memory loads). This provides ample time for precaching of the next diagonal and rectangular blocks required.

Use of registers rather than shared memory is now considered. Whereas execution of a thread will block on a write to shared memory, it will not for a load into a register. Note, however, that a thread will block on the later use of that register. This allows instruction-level parallelism. If this property could be exploited then it may be possible to accelerate the inner loops. However, the `xs` variable or equivalent will still need to be transferred via shared memory. Each thread has a maximum of 63 registers available, some of which are required for normal computation use; in practice, it is difficult to use more than 32 registers per thread to store part of the matrix. Multiple warps are thus required to fully cache the matrix. Two alternative algorithms are tested, using different mappings of registers to parts of the matrix. Variant A (shown as Listing 3) assigns columns to warps in a cyclic fashion, while variant B (not shown) uses a block cyclic mapping. The former requires a thread synchronization after each column, whereas the latter involves more complex control logic.

Table 1 compares these against the shared memory implementation and the NVIDIA CUBLAS implementation of `_trsv()`. The shared-memory implementation outperforms the register implementations. This is probably due to synchronization overhead in variant A and complex control flow in variant B. Further, for larger block sizes there are insufficient registers to hold the matrices and register spill occurs, resulting in a significant performance drop. Both the shared memory and register implementations described outperform the CUBLAS.

Listing 2: Cache handling for larger blocks in shared memory

```
// threadsx = threadIdx.x = n-blkSize
// threadsy = threadIdx.y = 4 or 8, say (autotune).
template <int n, int blkSize, int threadsx, int threadsy>
void __global__ blkSolve(const double *a, int lda, double *xglobal) {
   double __shared__ xshared_actual[n], rect[(n-blkSize)*blkSize],
      cache_even[blkSize*blkSize], cache_odd[blkSize*blkSize];
   int tid = threadsx*threadIdx.y+threadIdx.x;

   /* Precache x and first diagonal block */
   if(tid<n) xshared[tid] = xglobal[tid];
   tocache <blkSize,threadsx*threadsy> (a, lda, cache_even);
   __syncthreads();

   /* Main loop */
   double *xshared = xshared_actual;
   for(int ii=0; ii<n; ii+=blkSize) {
      /* Preload entries for rectangular block */
      if(threadIdx.y!=0)
         cacherect <blkSize, threadsx*(threadsy-1)> (a+blkSize, lda, rect,
            n-blkSize);
      /* Solve diagonal block */
      if(threadIdx.y==0 && threadIdx.x<blkSize) {
         double val = xshared[threadIdx.x];
         if(ii%(2*blkSize)==0) dblkSolve<blkSize>(cache_even, blkSize, val);
         else                  dblkSolve<blkSize>(cache_odd, blkSize, val);
         xshared[threadIdx.x] = val;
      } else if(ii+blkSize<n)
         if(ii%(2*blkSize)==0) tocache <blkSize,threadsx*threadsy-32>
                               (&a[blkSize*(lda+1)], lda, cache_odd);
         else                  tocache <blkSize,threadsx*threadsy-32>
                               (&a[blkSize*(lda+1)], lda, cache_even);
      __syncthreads();
      /* Apply rectangular block */
      if(threadIdx.y==0 && threadIdx.x<n-blkSize)
         for(int j=blkSize; j+ii<n; j+=n-blkSize) {
            double val=0;
            if(ii+j+threadIdx.x<n) {
               for(int i=0; i<blkSize; i++)
                  val += rect[i*(n-blkSize)+j-blkSize+threadIdx.x] *
                     xshared[i];
               xshared[j+threadIdx.x] -= val;
            }
         }
      __syncthreads();
      a+=blkSize*(lda+1); xshared+=blkSize;
   }
   /* Store x back to global memory */
   if(tid<n) xglobal[tid] = xshared_actual[tid];
}
```

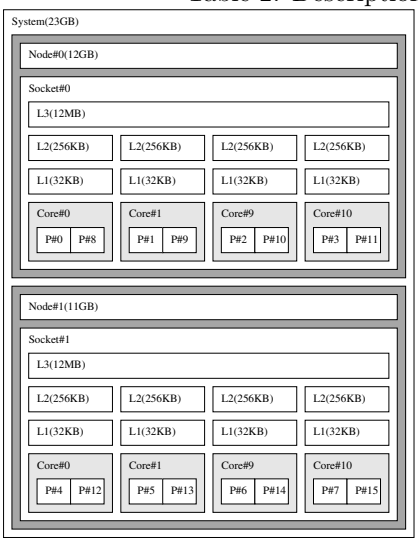
Listing 3: Register-based solve A

```
template <int threadsx, int threadsy, int n>
__launch_bounds__(threadsx*threadsy, 1)
void __global__ regSolveA(const double *a, int lda, double *xglobal) {
   double aval[n/threadsx][n/threadsy];
   volatile double __shared__ xshared[n];

   /* Read x into shared memory and a into registers */
   int tid = threadsx*threadIdx.y+threadIdx.x;
   if(tid<n) xshared[tid] = xglobal[tid];
#pragma unroll
   for(int i=0; i<n; i+=threadsx) {
      const double *mya = &a[threadIdx.y*lda+i+threadIdx.x];
#pragma unroll
      for(int j=0; j<n; j+=threadsy) {
         aval[i/threadsx][j/threadsy] = *mya;
         mya += lda*threadsy;
      }
   }

   /* Perform solve */
#pragma unroll
   for(int i=0; i<n; i++) {
      __syncthreads();
      if(threadIdx.y != i % threadsy) continue;
#pragma unroll
      for(int j=0; j<n; j+=threadsx) {
         if(j+threadIdx.x>i)
            xshared[j+threadIdx.x] -= aval[j/threadsx][i/threadsy] *
               xshared[i];
      }
   }

   /* Copy x back from shared memory */
   __syncthreads();
   if(tid<n) xglobal[tid] = xshared[tid];
}
```

Table 1: Performance of different implementation on small sizes. A $*$ indicates register spill reported by compiler. Times are in μs measured using `nvvp`.

$n =$	32	64	96	128
Shared-memory variant	7	13	19	25
Register variant A	17	38	68	149*
Register variant B	19	37	75*	125*
CUBLAS `dtrsv()`	31	58	85	113

Table 2: Description of hardware for machine `mitchell`.



Host	
Architecture	Intel Xeon E5620
Compiler	Intel Fortran 12.0.0
	ifort -g -fast -openmp
BLAS	MKL 10.3.0
Cores	$2 \times 4 = 8$
Memory	24GB
Memory bandwidth	25.6 GB/s

C2050	
Architecture	Fermi (capability 2.0)
CUDA Driver	4.20
CUDA Runtime	4.10
Cores	$14 \times 32 = 448$
Memory	3GB
Memory bandwidth	144 GB/s

3 Large matrices

By extending the approach exemplified by Figure 1 to a further level, multiple thread blocks (and hence SMs) can be used. At the coarsest level, the work of the previous section is applied to the block diag, while a high-performance matrix-vector product (`_gemv`) implementation can be applied to the block rect. If the kernels are launched in the same stream, the driver will ensure they are executed in order. Exploitation of this for purposes of synchronization is referred to as *launch-synchronization* in this paper.

Figure 2 compares the performance of this approach for various block sizes `nb` to that of the Intel MKL running on the host (see Table 2), and the NVIDIA CUBLAS 4.1. The maximum size of matrix that can be held in memory on the C2050 is just under $n = 20000$, so the full range of possible n values is shown. If n is not an exact multiple of `nb` then a variant of the `blkSolve` kernel is used that permits blocks of variable size less than `nb` at the cost of some loss of performance. The `nb` $= 128$ version outperforms the others except when $n = 32, 64$ or 96, where the variants with $n =$ `nb` are better. Larger block sizes were not tested as the decreasing returns with increasing block size suggests this is unlikely to yield significant improvement.

Close examination of the actual performance obtained shows that there is much room for improvement. The table below highlights the two main issues, namely the kernel launch overheads and a disproportionate amount of time spent in the `blkSolve()` routine.

$n =$	512	1024	4096
Time (μs) in `blkSolve()`	108.3	217.3	904.7
Time (μs) in `dgemv()`	37.8	95.1	842.0
Execution time (μs)	171.0	370.8	2006.5
Launch overhead	17.0%	18.7%	14.9%
Matrix entries used by `blkSolve()`	18%	9%	2%

To remove the kernel launch overhead, synchronization can instead be performed using global memory. This approach involves the combination of a level 2 cache read with a `__threadfence()` instruction, both of which are relatively cheap. This is achieved by combining the solve for a diagonal block with the matrix-vector multiplication in a single kernel. Two additional benefits derive from this approach. First, while the diagonal block solve is proceeding on one SM, others can be performing matrix-vector multiplies for remaining rows. Second, additional work is available to mask the pre-caching of matrix blocks.

Figure 2: Performance of CPU BLAS (MKL), CUBLAS and the launch-synchronized `blkSolve/dgemv` kernels for different block sizes `nb`. Times are measured using `clock_gettime()`. Lower picture shows small n detail.

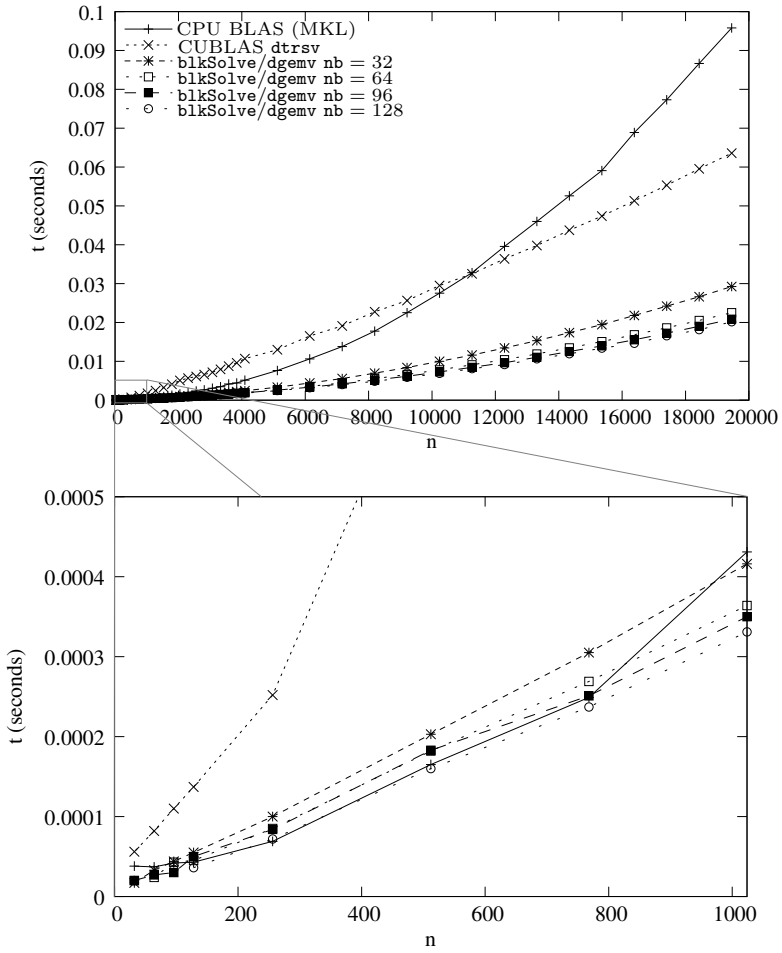
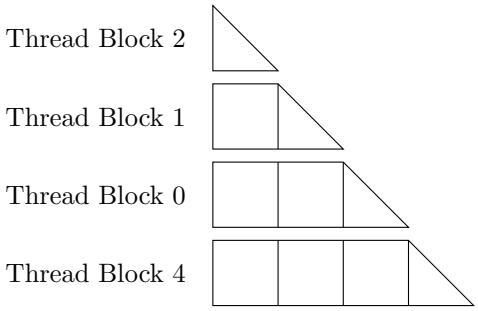


Figure 3: Division of matrix into block rows, then into blocks. Assignment to thread blocks is performed dynamically at run time.



If the matrix is divided symmetrically into blocks, then a matrix-vector multiply can be associated with each off-diagonal block and a triangular solve with each diagonal block. The work associated with a given off-diagonal block cannot begin until the solve for the diagonal block in the column has completed. The diagonal solve in a given row cannot commence until all matrix-vector multiplies for blocks in that row have completed.

Listing 4 outlines an implementation that obeys these constraints. Each thread block is assigned a block row of the matrix that is further subdivided into blocks (Figure 3). For each row the matrix-vector multiplies are executed in order from left to right followed by the triangular solve. If the required data for a matrix-vector multiply is not ready then execution blocks until it becomes available.

As triangular solves on diagonal blocks must occur in order, it is sufficient to track only the latest row that has completed. This requires only a single scalar value in global memory that is incremented upon completion of a row. A `__threadfence()` instruction is used to ensure the solution vector is visible to all threads before incrementing this counter.

A further global memory synchronization is required to dynamically allocate matrix rows to thread blocks. This is required as there is no guarantee that thread blocks are scheduled to run in index order, and static allocation is likely to lead to deadlock. Such dynamic allocation is easily accomplished using a scalar in global memory and `atomicAdd()`.

Both these global memory synchronizations require initialization before execution of the main kernel. This is easily accomplished using a trivial kernel run on a single thread.

Figure 4 shows the start of a typical execution trace for a large matrix where each SM executes 4 thread blocks simultaneously (limited by shared memory). Each thread block exhibits two modes of operation. Mode 1 operates while `col` $\leq$ `sync[0]` (i.e. until it has "caught up" with the current column) and is characterised by constant execution of useful work. Mode 2 operates once the thread block begins waiting for data before proceeding with computation. It is characterised by short spurts of execution immediately following release of data for a column followed by a wait for more data. As these spurts of useful execution are dependant on the same trigger, they are synchronized between thread blocks, even within the same SM. If no thread block on an SM is operating in mode 1, then the SM idles while waiting for the next diagonal solve to complete. This occurs only for the first (and in some cases second) thread block executed on each SM, for which mode 1 execution is short or non-existent. It also occurs to a lesser degree near the end of execution when the number of thread blocks per SM decreases (not shown).

Mode 2 operation is bandwidth bound, and efficiency is relatively high. Further, any improvement is unlikely to impact the overall performance of the kernel as mode 1 operation dominates the critical path. Mode 1 operation is bound on the performance of the matrix-vector multiply of block $(row, row - 1)$ and the solve on the diagonal block that constitute the critical path required to begin execution on the next column. The pre-caching techniques of Section 2 can be used to accelerate these operations. These suggest loading the diagonal block into shared memory is required to minimize latency. The critical rectangular block can also be loaded into shared memory, but registers can also be used as no internal synchronizations

Listing 4: Outline code for global-memory synchronized dtrsv

```
/* Sets sync values correctly prior to call to dtrsv_lu_exec */
void __global__ dtrsv_init(int *sync) {
   sync[0] = -1; // Last ready column
   sync[1] = 0; // Next row to assign
}

/* Performs dtrsv for Non-transposed Lower-triangular Unit matrices
 * Requires dtrsv_init() to be called first to initialize sync[].
 * Best performance on C2050 with threadsx=32, threadsy=8.
 */
template <int nb, int threadsx, int threadsy>
void __global__ dtrsv_lu_exec(int n, const double *a, int lda,
      double *xglobal, int *sync) {

   int nblk = n / nb;
   int tid = threadsx*threadIdx.y + threadIdx.x;
   double __shared__ cache[nb*nb], partSum[threadsy*threadsx];

   /* Get row handled by this thread block */
   int row = nextRow(&sync[1]);

   /* Loop over blocks as they become available */
   double val;
   if(threadIdx.y==0) val = xglobal[row*nb+threadIdx.x];
   else               val = 0;
   for(int col=0; col<row; col++) {
      /* apply update from block (row, col) */
      const double *aval = &a[(col*nb+threadIdx.y)*lda + row*nb+threadIdx.x];
      wait_until_ge(&sync[0], col); // Wait for diagonal block to be done
      for(int j=0; j<nb; j+=threadsy)
         val -= aval[j*lda] * xglobal[col*nb+j];
   }
   partSum[threadIdx.y*threadsx+threadIdx.x] = val; __syncthreads();
   if(threadIdx.y != 0) return; /* Only use first warp from here on */
   for(int i=1; i<threadsy; i++) val += partSum[i*threadsx+threadIdx.x];

   /* Apply update from diagonal block (row, row) */
   const double *aval = &a[(row*nb+threadIdx.y)*lda + row*nb+threadIdx.x];
   dblkSolve<nb>(aval, nb, val);
   xglobal[row*nb+tid] = val;

   /* Notify other blocks that soln is ready for this row */
   colDone(sync, row); // Note: contains __threadfence();
}
```

Figure 4: Start of execution trace for global memory-synchronized code with $n = 4096$. Each SM executes 4 thread blocks simultaneously. The bottom black trace for each SM represents useful activity across all thread blocks. Colour is used to indicate where different thread blocks start and stop.

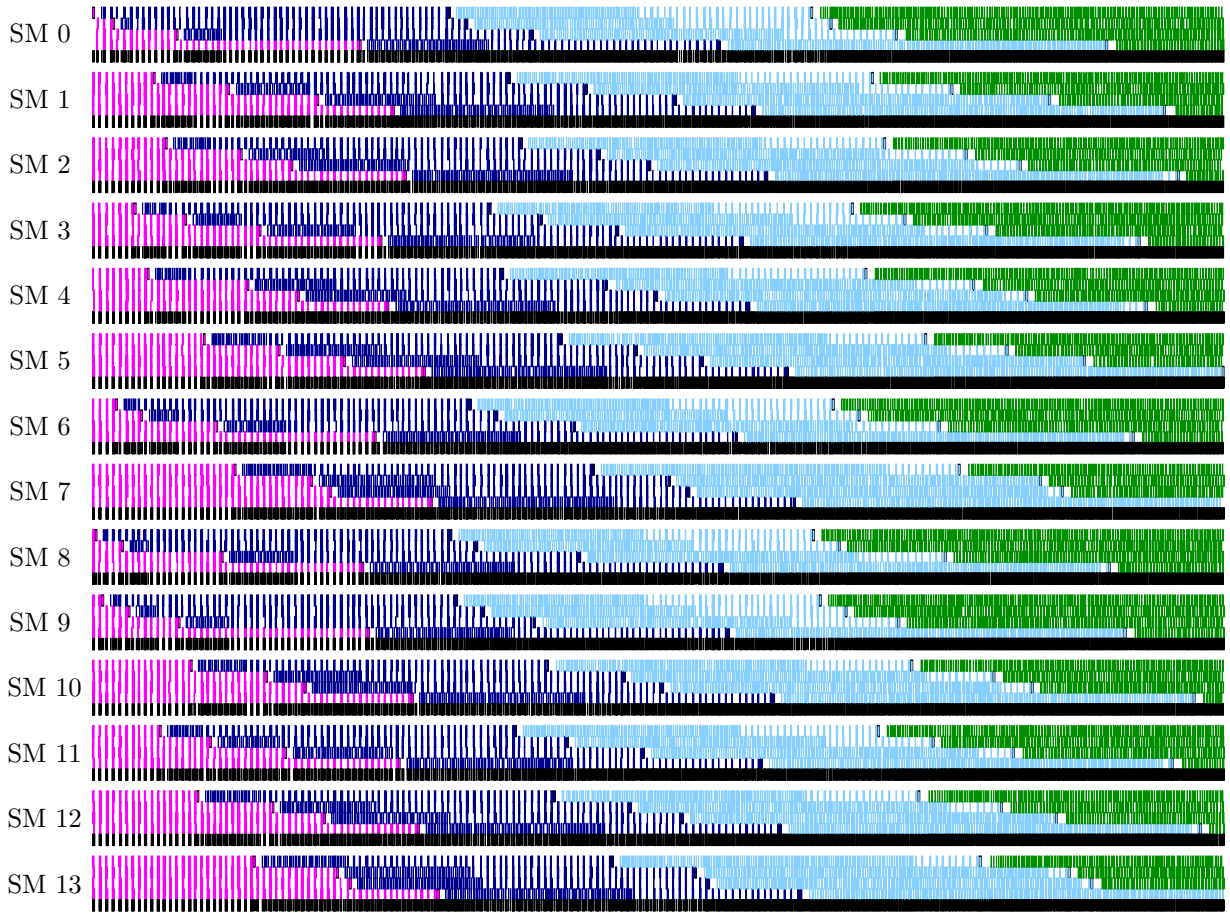
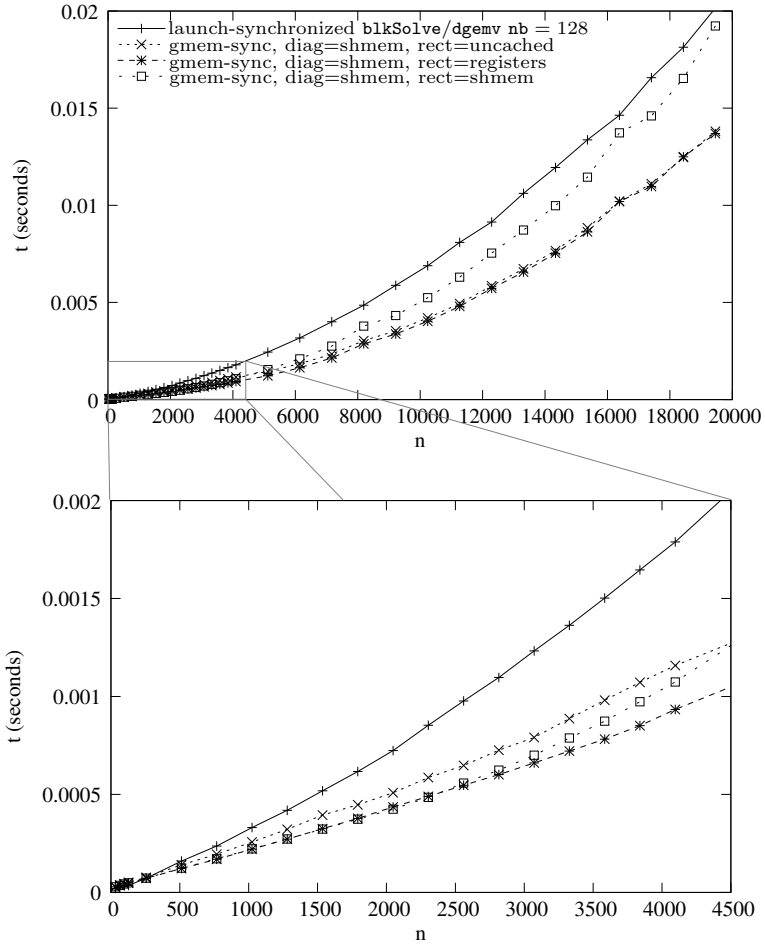


Figure 5: Performance of global memory-synchronized kernel using different pre-caching techniques. The launch-synchronized method with `nb` = 128 is shown for comparison. Times are measured using `clock_gettime()`. Lower picture shows small n detail.



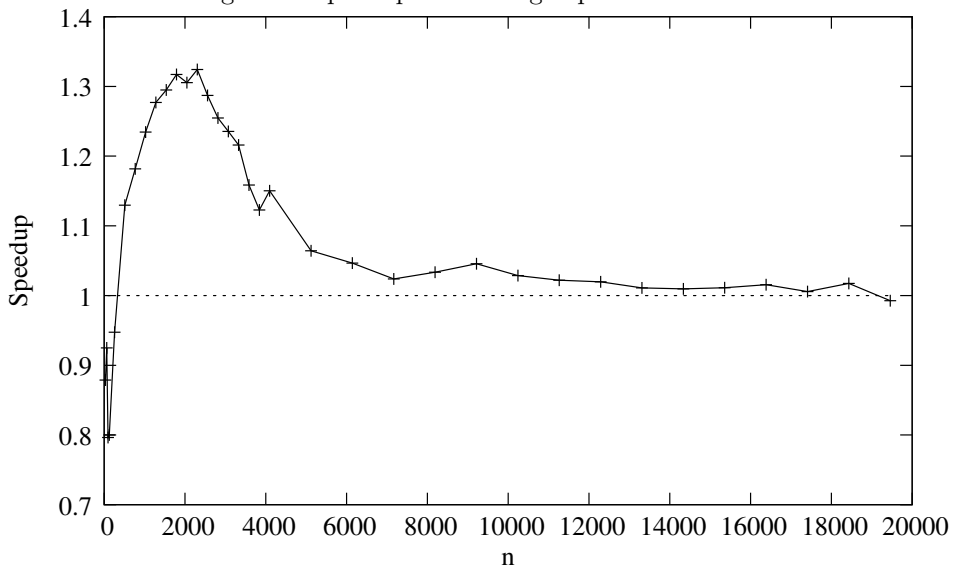
are required, unlike the triangular solve. If shared memory is only used for the diagonal block, up to four thread blocks can execute on a single SM. If shared memory is used for both blocks, this reduces to two thread blocks per SM.

Figure 5 shows results for different caching options for the rectangular block (no caching, shared memory and registers) and includes a comparison against the best kernel-synchronized variant. For small matrices pre-caching is important and the shared memory variant outperforms the non-caching variant. For large matrices occupancy is more important and the non-caching variant outperforms the shared memory version. Both are consistently equalled or outperformed by the register caching variant that combines their best features. A transition from latency-bound (time linear in n) to bandwidth-bound (time quadratic in n) behaviour is apparent around $n = 1500$.

4 Explicit inversion

In the memory-synchronized algorithm, each thread block spends a significant amount of time waiting for data in its mode 2 operation. The resulting under-utilised resource can be used to perform the explicit

Figure 6: Speedup from using explicit inversion.



inversion of the diagonal block, allowing the replacement of the critical path's latency-bound solve with a bandwidth-bound matrix-vector multiply. This is similar to the approach of Ltaief et al. [2] used in the MAGMA library.

This differs from other approaches [4] that perform an explicit inversion of the whole matrix. A whole matrix inversion increases the main memory traffic significantly, and often converts the problem from being memory to compute bound.

Following the component-wise backwards stable divide and conquer approach laid out by Higham [1], the triangular matrix L, and its inverse $X = L^{-1}$, are partitioned as

$$L = \begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \qquad X = \begin{pmatrix} X_{11} & \\ X_{21} & X_{22} \end{pmatrix}.$$

By considering the expression $LX = I$, observe that $L_{11}X_{11} = I$, $L_{21}X_{11} + L_{22}X_{21} = 0$ and $L_{22}X_{22} = I$. X_{11} and X_{22} can therefore be found as the inverse of L_{11} and L_{22} using recursion or a trivial base case. Care must be taken in determination of X_{21} to ensure stability. Higham's method B can be used, solving $L_{22}X_{21} = -L_{21}X_{11}$ by substitution.

Timings show that the inversion of a 32×32 block completes in the same time as 3–4 block substitutions and their associated matrix-vector products. However, the replacement of the block substitution by a matrix-vector product halves the execution time on the critical path. Inversion is therefore used only for the fifth block row and beyond.

Figure 6 shows the speedup achieved against the best global-memory variant from the previous section. For small matrices ($n < 500$) the extra overhead of the inversion causes a slowdown. Larger matrices are bandwidth bound, so latency improvements are of limited effect. Further, as the inversion algorithm is more complicated than substitution, register pressure reduces the number of threads than can be employed, slowing the matrix-vector multiplications. However, for medium-sized matrices in the range 500–5000 up to a 30% performance improvement is possible.

5 Transpose problem

The transpose problem can be approached in a similar manner to the non-transpose case so far treated. However, to achieve maximal memory bandwidth it is necessary for threads of a warp to read entries in a

Figure 7: Non-transpose versus Transpose matrix access for optimal bandwidth. Labelling of matrix elements $x.y$ indicates accessing thread coordinates within thread block. Threads with the same y value belong to the same warp.

$$\begin{pmatrix} 0.0 & & & & \\ 1.0 & 1.1 & & & \\ 2.0 & 2.1 & 2.2 & & \\ \vdots & \vdots & \vdots & \ddots & \\ 31.0 & 31.1 & 31.2 & \cdots & 31.y \end{pmatrix}$$

Non-transpose access pattern

$$\begin{pmatrix} 1.0 & 1.0 & 2.0 & \cdots & 31.0 \\ & 1.1 & 2.1 & \cdots & 31.1 \\ & & 2.2 & \cdots & 31.2 \\ & & & \ddots & \vdots \\ & & & & 31.y \end{pmatrix}$$

Transpose access pattern

row-wise rather than column-wise fashion (see Figure 7). This introduces the requirement for an additional reduction.

As insufficient shared memory is available to perform the reduction in an optimal fashion at the same time as maintaining high occupancy, it must be removed from the critical path. This can be achieved by performing a transpose operation on the blocks belonging to the critical path as they are cached.

The results of this approach are shown in Figure 8. Observe that the performance is slightly lower than for the non-transpose case, especially for smaller matrices, due to the additional overhead of the extra reduction. However, the code still significantly outperforms existing CUDA implementations.

6 Conclusions

By selecting the best algorithm from the previous section for relevant domains of n values, a high performance `_trsv` implementation can be developed. Figures 9 and 10 demonstrate performance comparisons against other notable implementations.

By analysing the performance limiting factors that affect triangular solve and addressing these with different techniques depending on the size of matrix, a 5–15× improvement was achieved over the current CUBLAS implementation. Explicit pre-fetching and selective and stable inversion to reduce dependencies between operations are critical to reducing latency. Avoiding kernel-launch overheads and ensuring high occupancy are necessary to maximize bandwidth utilisation for large matrices, allowing over 75% of peak to be achieved.

The code described in this paper is available under a 3-clause BSD licence from the CCP Forge website:
`http://ccpforge.cse.rl.ac.uk/gf/project/asearchralna/`

ACKNOWLEDGEMENTS

With many thanks to Mike Giles and Jennifer Scott for comments on drafts of this paper. Further credit accrues to Mike for the observation that registers can be used to pre-cache the rectangular block on the critical path for the global memory version, allowing high occupancy.

References

[1] N. J. Higham, *Stability of parallel triangular system solvers*, SIAM J. Scientific Computing, 16 (1995), pp. 400–413.

[2] H. Ltaief, S. Tomov, R. Nath, P. Du, and J. Dongarra, *A scalable high performant cholesky factorization for multicore with gpu accelerators*, in High Performance Computing for Computational

Figure 8: Performance of best kernels adapted for transpose solve. Times are measured using `clock_gettime()`. Lower picture shows small n detail.

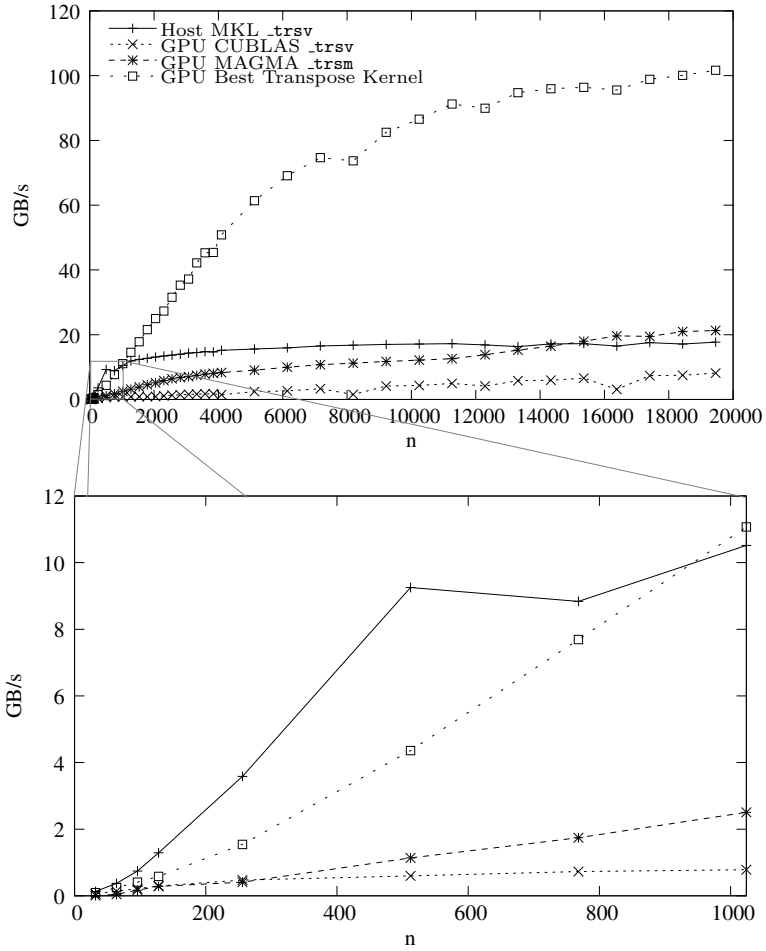


Figure 9: Comparison of bandwidth achieved using best kernels from this paper against Host MKL _trsv, and GPU CUBLAS _trsv and GPU MAGMA _trsm implementations. Lower picture shows small n detail.

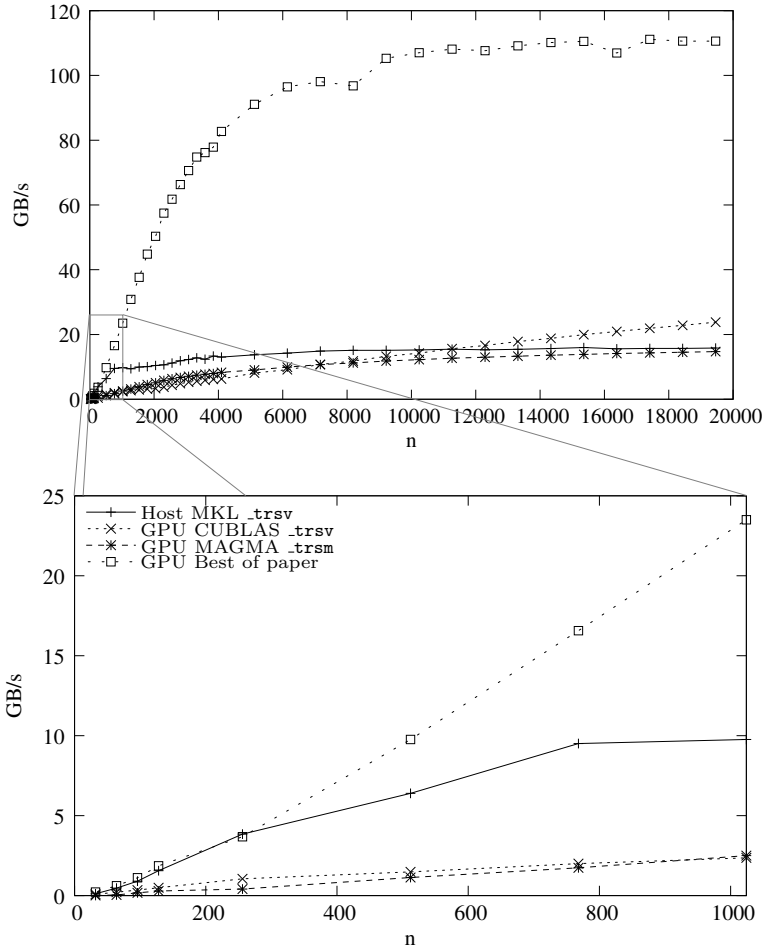
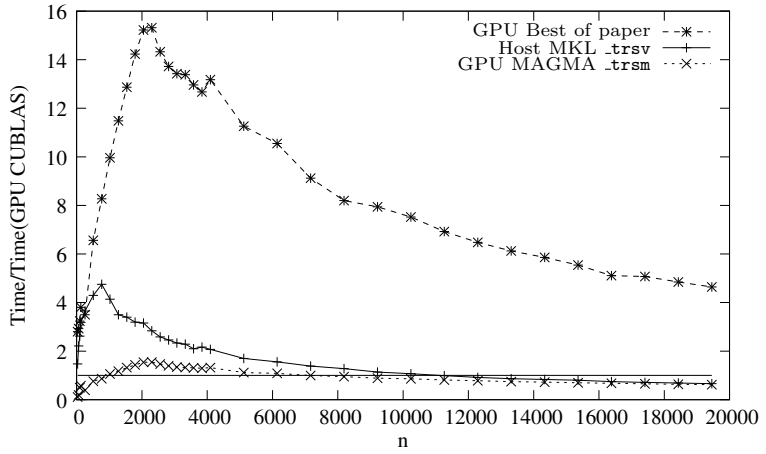


Figure 10: Comparison of time taken (speedup) against GPU CUBLAS _trsv for best kernels from this paper, Host MKL _trsv, and GPU MAGMA _trsm.



Science – VECPAR 2010, J. Palma, M. Dayd, O. Marques, and J. Lopes, eds., vol. 6449 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2011, pp. 93–101.

[3] NVIDIA, *CUDA Toolkit 4.1 CUBLAS Library*, January 2012.

[4] F. Ries, T. De Marco, M. Zivieri, and R. Guerrieri, *Triangular matrix inversion on graphics processing unit*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, New York, NY, USA, 2009, ACM, pp. 9:1–9:10.