SIAM J. SCI. COMPUT.
Vol. 35, No. 3, pp. C303–C322

A FAST DENSE TRIANGULAR SOLVE IN CUDA*

J. D. HOGG†

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 reducing the cost of compute-bound parts of the computation, memory-bound operations such as this kernel become increasingly important. This is particularly noticeable in sparse direct solvers used for optimization applications where multiple memory-bound solves follow each (traditionally expensive) compute-bound factorization. In this paper, a high performance implementation of the triangular solve is developed through an analysis of theoretical and practical bounds on its run time. This implementation outperforms the CUBLAS by a factor of 5–15.

Key words. BLAS, CUDA, GPU, triangular solve

AMS subject classifications. 65F05, 65Y05, 65Y10

DOI. 10.1137/12088358X

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 nonunit, 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. Traditional thinking that the factorization phase dominates the run time in such applications is now being reevaluated [3, 4] as the performance balance between compute- and memory-bound operations continues to change. When sparse factorizations are considered, the situation becomes more extreme as there are fewer flops per entry in the factorization phase and a single solve will typically employ multiple calls to a dense triangular solve routine for small submatrices of dimension ranging from one to several hundred, and occasionally several thousand. However, as sparsity introduces additional parallelism into the triangular solve it becomes possible to run multiple solves in parallel, so the ability to simultaneously run solves on both the host and an attached accelerator is very attractive.

The nontransposed, 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.

*Submitted to the journal's Software and High-Performance Computing section July 6, 2012; accepted for publication (in revised form) April 22, 2013; published electronically June 27, 2013.
http://www.siam.org/journals/sisc/35-3/88358.html
†Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Oxford, Didcot, UK (jonathan.hogg@stfc.ac.uk). This author's work was supported by EPSRC grant EP/J010553/1.

TABLE 1.1
Description of hardware for test machine.

Host		C2050	
Architecture	Intel Xeon E5620	Architecture	Fermi (capability 2.0)
Compiler	Intel Fortran 12.0.0	CUDA Driver	4.20
	ifort -g -fast -openmp	CUDA Runtime	4.10
BLAS	MKL 10.3.0		
Cores	$2 \times 4 = 8$	Cores	$14 \times 32 = 448$
Memory	24 GB	Memory	3 GB
Memory b/w	25.6 GB/s	Memory b/w	144 GB/s

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 CUBLAS version 4.1 [7].

In this paper we consider the lower triangular, nontranspose variant of `_trsv()`, where L has unit diagonal and x has unit stride. The transpose algorithm is also briefly addressed. Nonunit diagonal and nonunit stride of x can be trivially accommodated within the algorithms presented but are omitted here for clarity. The upper triangular versions can be derived through small changes to the lower triangular algorithms.

We restrict our attention to column-major storage, as our intention is to implement the BLAS API that requires this. However, it is worth observing that row-major format is easily accommodated by treating row-major $Lx = b$ as the equivalent column-major $U^T x = b$. While in performance terms one of row-major or column-major will yield the best performance, this is a choice that can be made by the user through this equivalence. As we expect to perform a single floating-point operation per matrix entry, the overheads involved in conversion to more exotic formats mean that they are not worth considering.

Most results in this paper are obtained on the test machine detailed in Table 1.1 using an NVIDIA C2050 Fermi card. Some additional results are given on an M2090 Fermi and GTX 680 Kepler cards at the end of the paper.

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 utilizing the full power of the graphics processing unit (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, and a performance model is developed in section 5. Finally, results are presented on other cards in section 6, and conclusions are presented in section 8.

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.

TABLE 2.1
Storage required for dense matrices.

	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: 48 KB
Max reg/block: 32768

LISTING 1
32×32 *solve.*

```
1  template <int blkSize>
2  void __device__ dblkSolve(const double *a, int lda, double &val) {
3
4      volatile double __shared__ xs;
5
6  #pragma unroll 16
7      for(int i=0; i<blkSize; i++) {
8          if(threadIdx.x==i) xs = val;
9          if(threadIdx.x>=i+1)
10             val -= a[i*lda+threadIdx.x] * xs;
11     }
12 }
```

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 in Table 2.1. It shows the amount of storage required to read the matrix fully into memory for multiples of the warp size (32 for all recent NVIDIA cards, including Kepler).

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. Exploiting this feature is known as *warp-level programming*. If multiple warps are used, 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 precaching 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, and 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`. The code represents the best of many different variants tested and exploits warp-level programming, the key optimization

LISTING 2
Cache handling for larger blocks in shared memory.

```
1  Subroutine blkSolve(L, x):
2     /* Symmetrically partition L into nblk rows and nblk columns */
3
4     All Warps:
5        Precache x from global into shared memory
6        Precache diagonal block L(1,1) into shared memory
7     __syncthreads()
8
9     For i = 1:nblk /* Loop over block columns of a */
10       Warp 0:
11          Copy x(i) from shared memory to registers
12          dblkSolve() on precached diagonal−block A(i,i)
13       Other Warps:
14          Precache off−diagonal blocks L(i+1:nblk,1) into shared memory
15          Precache diagonal block L(i+1,i+1) into shared memory
16       __syncthreads()
17
18       Warps 0:nblk−i−1 /*, i.e.,\ one thread per row below diagonal block */
19          Perform matrix−vector multiplication x(i+1:nblk) −= L(i+1:nblk)*x(i)
20       Other Warps:
21          No−op
22       __syncthreads()
23    End For
24
25    All Warps:
26       Store x back from shared memory to global memory.
27 End Subroutine blkSolve
```

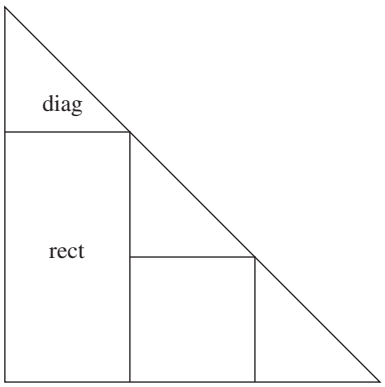

FIG. 1. *Partitioning of matrix into block columns for small matrix solves.*

being that each thread's element of the right-hand side vector is stored in a register (through explicit use of a local variable `val`), with explicit communication being via the single volatile shared memory location `xs`. As a single warp is used, `blkSize` $\leq$ `warpSize`. Best performance is obtained if `blkSize` = `warpSize` (= 32 for the C2050).

The second listing shows pseudocode demonstrating how memory management is performed using caching to avoid global-memory latency waits in `dblkSolve()`. A large team of noncomputing threads (typically 3–7 times as many as are used for computing) is used to load the matrix into a shared memory cache during the stalls in execution of `dblkSolve()`.

TABLE 2.2
Performance of different implementations on small matrix 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

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. In each iteration, stalls occur due to shared memory latency (approximately 35 cycles per access) and pipeline depth (approximately 5 cycles per dependant instruction). Timing indicates that in practice each iteration takes an average of 135 cycles (corresponding 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 assigns columns to warps in a cyclic fashion, while variant B uses a block cyclic mapping. The former requires a thread synchronization after each column, whereas the latter involves more complex control logic.

Table 2.2 compares these against the shared memory implementation and the NVIDIA CUBLAS 4.1 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.

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 of size `nb`, 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, and the NVIDIA CUBLAS. 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`, a variant of the `blkSolve` kernel is used that permits blocks of variable size less than

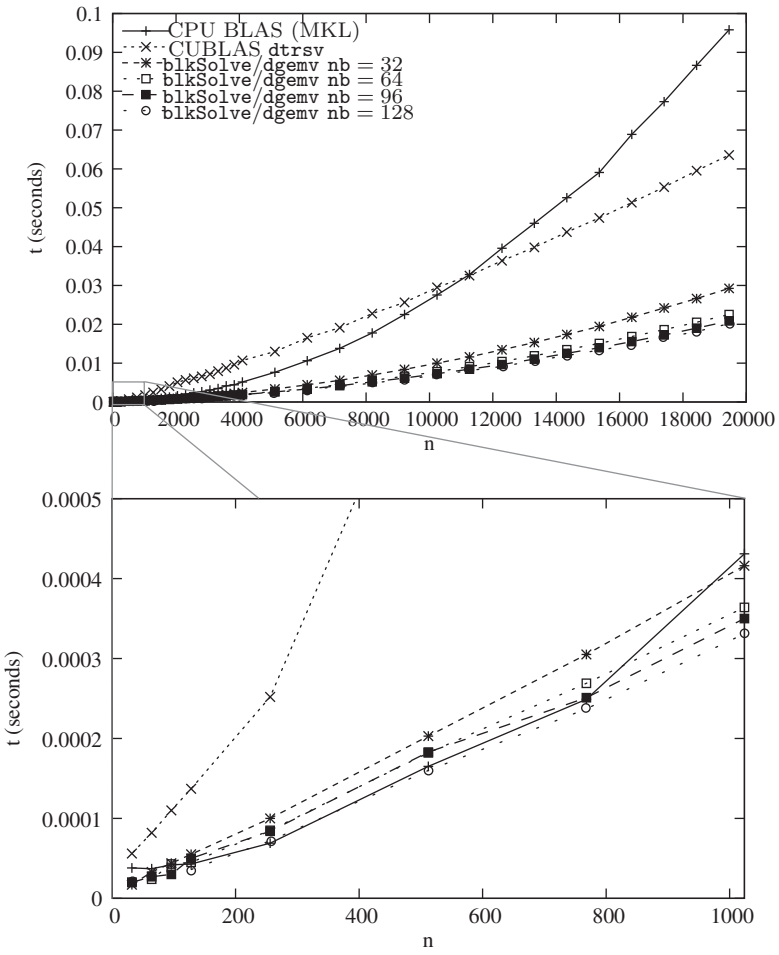

FIG. 2. *Performance of CPU BLAS (MKL), CUBLAS, and the launch-synchronized* `blkSolve/dgemv` *kernels for different block sizes* `nb`*. Times are measured using* `clock_gettime()`*. Lower picture shows small n detail.*

`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 they would require more than the 48k of available shared memory. Given the decreasing returns it is unlikely that any modification to overcome this limitation would yield any significant improvement.

For comparison with other people's implementations of the `_trsv` kernel, Figure 3 shows kernel execution traces for the current major implementations. We observe that our implementation is using a much finer-grained decomposition than the CUBLAS, while CULA Dense [1] is limiting its performance by limiting the number of blocks it uses per kernel to 6, and is therefore not fully utilizing all available SMs. MAGMA [5, 6, 9] uses a very different approach from the other implementations, inverting the diagonal blocks to transform the problem into a series of matrix multiplications (a technique we extend to our algorithm in section 4).

Close examination of the actual performance obtained shows that there is much room for improvement. Table 3.1 highlights the two main issues, namely, the kernel

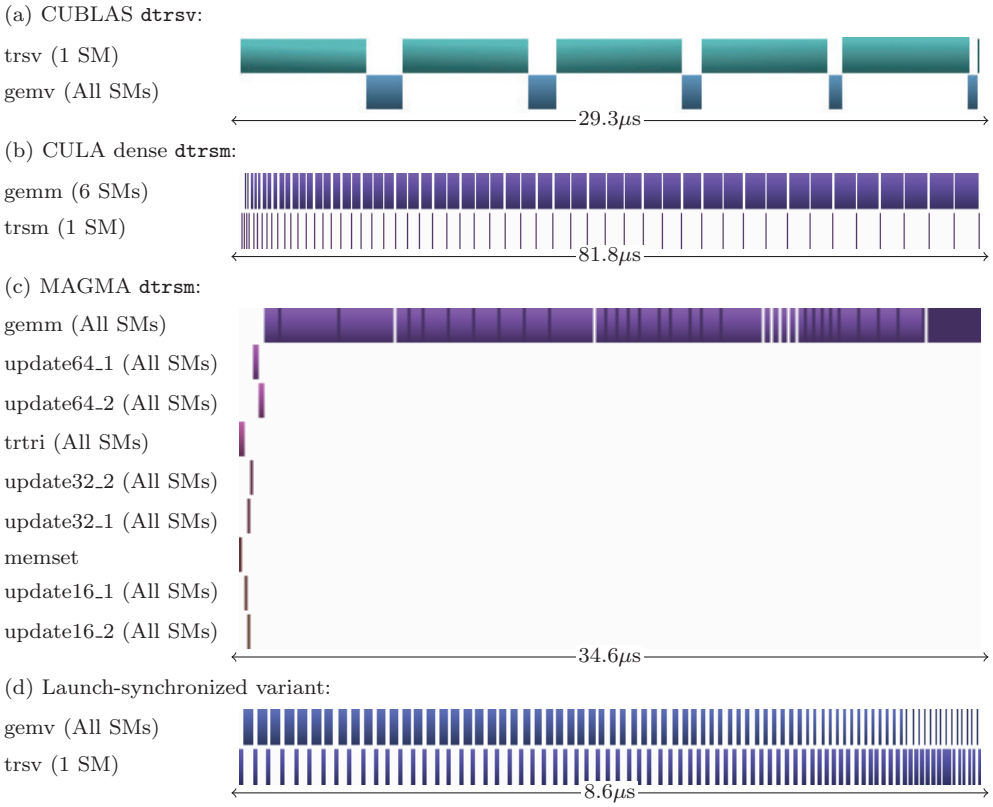

FIG. 3. *Kernel traces for* $n = 10240$ *with* (a) *CUBLAS* `dtrsv`, (b) *CULA dense* `dtrsm`, (c) *MAGMA BLAS* `dtrsm`, *and* (d) *Kernel-synchronized variant.*

TABLE 3.1
Performance analysis data for launch-synchronized kernel.

$n =$	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%

launch overheads and a disproportionate amount of time spent in the `blkSolve()` routine.

To remove the kernel launch overhead, synchronization can instead be performed using global memory, though care must be taken to avoid deadlocks caused by thread blocks being scheduled for execution out of order. 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 precaching of matrix blocks.

LISTING 3
Outline code for global-memory synchronized _trsv.

```
1  /* Sets sync values correctly prior to call to trsv_exec */
2  Subroutine trsv_init(sync[2]):
3     sync[0] = −1 // Last ready column
4     sync[1] = 0 // Next row to assign
5  End Subroutine trsv_init
6
7  /* Performs _trsv for Nontransposed Lower−triangular Unit matrices
8   * Requires trsv_init() to be called first to initialize sync[].
9   * Best performance on C2050 with 8 warps of 32 threads each. */
10 Subroutine trsv_exec(L, x, sync[]):
11    /* Symmetrically partition L into nblk rows and nblk columns */
12
13    Warp 0, Thread 0:
14       row = atomicIncrement(&sync[1])
15    __syncthreads();
16
17    All Warps:
18       Precache transposed L(row,row) into shared memory
19
20    Warp 0 only:
21       Local register val = x[threadIdx.x]
22    Other Warps:
23       Local register val = 0
24
25    For i = 1:row−1
26       Warp 0, Thread 0:
27          Wait until sync[0] >= i
28       __syncthreads()
29       All Warps:
30          Perform local matrix−vector multiply val += L(row, i)∗x(i).
31    End For
32    Store val into shared memory work array
33    __syncthreads()
34    Warp 0:
35       Perform reduction across shared memory work array
36       dblkSolve() on diagonal block L(row, row)
37       Store x(row) to global memory
38       __threadfence()
39       Thread 0:
40          sync[0]++ // Notify other thread blocks we have finished
41 End Subroutine trsv_exec
```

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 3 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 4). 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()`

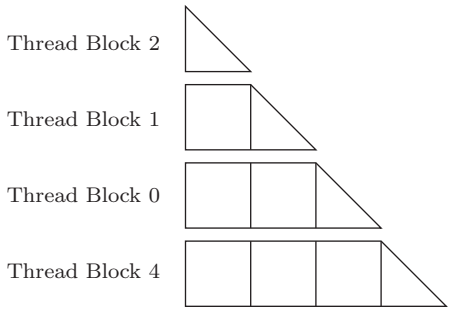

FIG. 4. *Division of matrix into block rows, then into blocks. Assignment to thread blocks is performed dynamically at run time.*

instruction is used to ensure that 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 of 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 5 shows the start of a typical execution trace for a large matrix where each SM executes four 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 characterized by constant execution of useful work. Mode 2 operates once the thread block begins waiting for data before proceeding with computation. It is characterized 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 nonexistent. 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 precaching techniques of section 2 can be used to accelerate these operations. These suggest that 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 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.

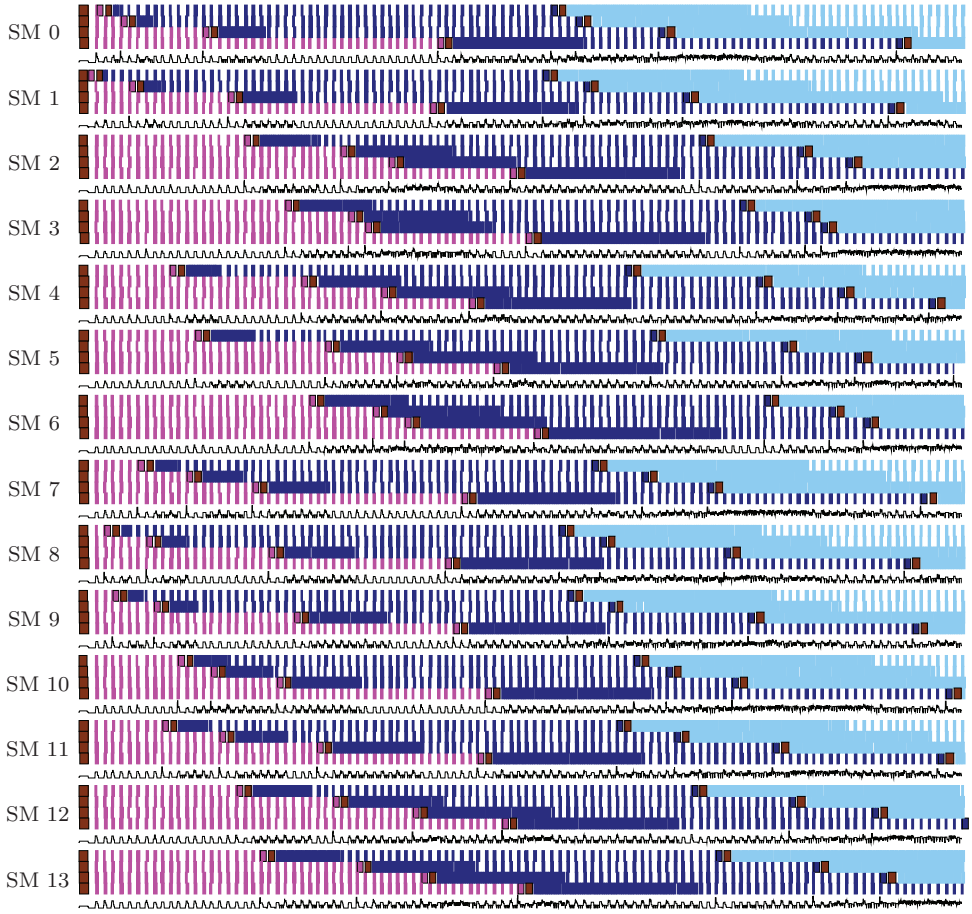

FIG. 5. *Start of execution trace for global memory-synchronized code with $n = 10{,}240$. Each SM executes four thread blocks simultaneously. The bottom black trace line for each SM represents useful activity across all the SMs, its height representing the average utilization of the SM at that point, with a maximum when all four blocks are active for the entire time span covered by that pixel. Color is used to indicate where different thread blocks start and stop, with brown indicating the caching action at the start of each block.*

Figure 6 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 precaching is important and the shared memory variant outperforms the noncaching variant. For large matrices occupancy is more important and the noncaching 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) behavior 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 underutilized resource can be used to perform the explicit 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 used in the MAGMA library [5, 6, 9].

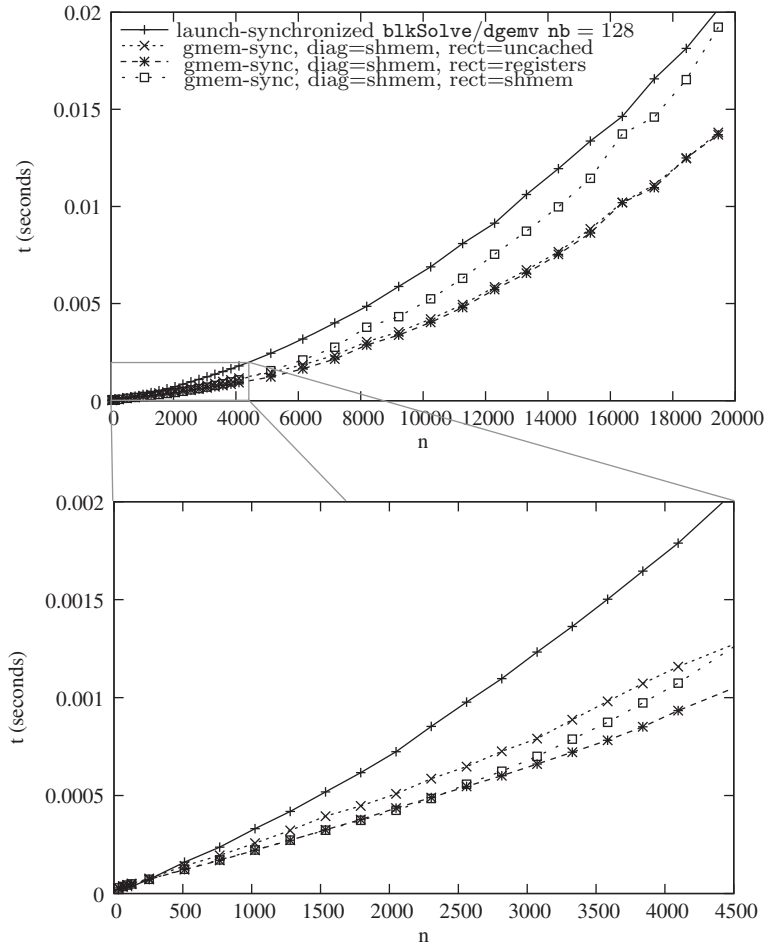

FIG. 6. *Performance of global memory-synchronized kernel using different precaching techniques. The launch-synchronized method with* `nb` $= 128$ *is shown for comparison. Times are measured using* `clock_gettime()`*. Lower picture shows small n detail.*

However, it differs from whole matrix inversion approaches such as those described by Ries et al. [8], where the entire matrix is inverted. A whole matrix inversion increases the main memory traffic significantly and often converts the problem from being memory to compute bound.

Following the componentwise backward stable divide and conquer approach laid out by Higham [2], 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.

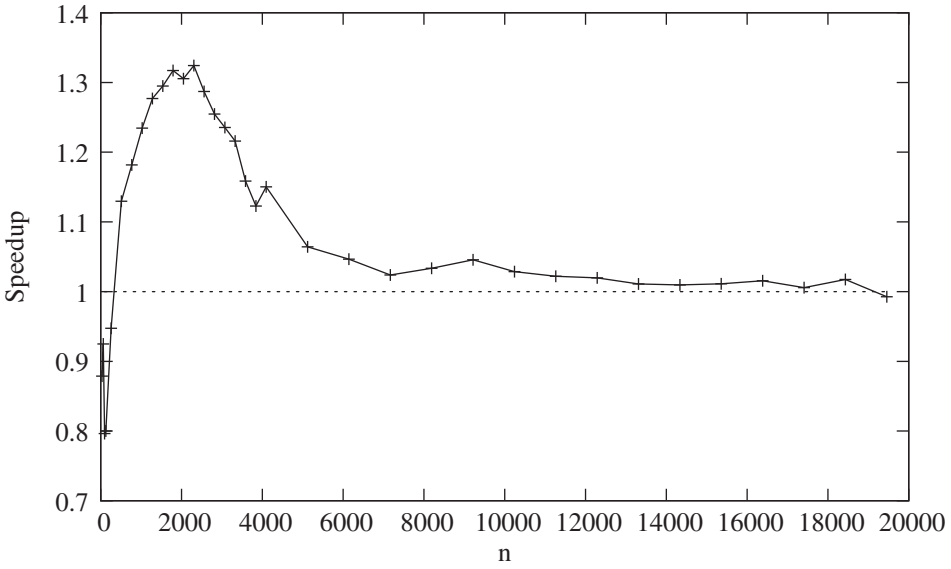

FIG. 7. *Speedup from using explicit inversion.*

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 7 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 that 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. Simple performance model. We build a simple performance model for the $Lx = b$ case by observing that the execution time of the entire algorithm is determined by the finish time of the last block row. This in turn is dependant on when the last block row begins. Our model assumes that if there are k blocks per SM and p SMs, then the final block b is preceded in the same critical "slot" by $b-kp, b-2kp, b-3kp, \ldots$ and so forth until all blocks have been allocated. (This behavior has been observed in practice.)

We observe experimentally that on the C2050, the termination time of blocks 1 to 78 is latency bound (i.e., Mode 1), with all remaining blocks being bound on available bandwidth (i.e., Mode 2). Using this, for each value n, we calculate the number of latency-bound blocks $nblk_{latency}$, the number of bandwidth-bound blocks $nblk_{bandwidth}$, and the number of bandwidth-bound row initializations $nrow_{init}$. (For latency-bound rows the initialization cost is hidden.)

For example, consider the case $n = 8192$. There are 256 block rows of size 32, and the critical slot executes rows 32, 88, 144, 200, and 256. This is modelled as 4 block initializations, 32 latency-bound blocks, and 688 bandwidth-bound blocks, plus

TABLE 5.1
Breakdown of performance model for $n = 8192$.

Task	$nrow$	$nblk_{latency}$	$nblk_{bandwidth}$
Row 32		32	
Row 88	1		88
Row 144	1		144
Row 200	1		200
Row 256	1		256
Total	4	32	688

a constant for the latency involved in transferring x to and from device memory and kernel launch overhead. These figures are broken down in Table 5.1.

The estimated run-time is based on the formula

$$t_{predict} = t_{setup} + nrow_{init} \times t_{init} + nblk_{latency} \times t_{latency} + nblk_{bandwidth} \times t_{bandwidth},$$

where, experimentally,

$$\begin{aligned} t_{setup} &= 6.65 \times 10^{-5}\text{seconds}, \\ t_{init} &= 4.50 \times 10^{-5}\text{seconds}, \\ t_{latency} &= 3.75 \times 10^{-6}\text{seconds}, \end{aligned}$$

and $t_{bandwidth}$ is calculated theoretically by dividing the amount of data per block by the available bandwidth. To this we add an experimentally determined 250 additional clock ticks to allow for bookkeeping operations and the load latency x from (hopefully) L2 cache:

$$\begin{aligned} t_{bandwidth} &= (\texttt{sizeof(double)} \times 32^2 \text{ bytes})/(2.58 \times 10^9 \text{ bytes/sec}) \\ &\quad + (250 \text{ clocks})/(1.15 \times 10^9 \text{ Hz}) \\ &= 3.39 \times 10^{-6}\text{sec}. \end{aligned}$$

Figure 8 compares this model to the actual results. It successfully predicts performance within $\pm 10\%$. The departure from the purely latency-bound curve occurs at $n = 2304$. Thereafter the model underpredicts actual performance except for the largest matrices. This underprediction is due to the simple model that assumes all blocks receive (on average) the same bandwidth. However, in reality, toward the end of the computation fewer blocks are executing per SM, so they each receive higher bandwidth. This is demonstrated in Figure 9 and affects a greater portion of the calculation for smaller n. The overprediction for the largest matrices is due to the false assumption that peak bandwidth can be sustained: even the simplest kernels, such as `dcopy`, only manage about 126 GB/s.

Given this model, it is apparent that exactly how many blocks run in a given SM slot before the final block affects the performance. Running more blocks per SM would reduce the number of blocks in the critical slot, at the expense of lowering the bandwidth. Conversely, running with fewer blocks increases bandwidth at the cost of increasing the number of blocks to be processed in each slot. In our experiments, using 5 blocks led to increased register spill into local memory and a consequent decrease in performance that was not offset by the expected gains. Runs using 3 blocks per SM did indeed demonstrate a 1.5 GB/s improvement for $n = 19456$; however, performance drops for smaller n prevented adoption as a universal default.

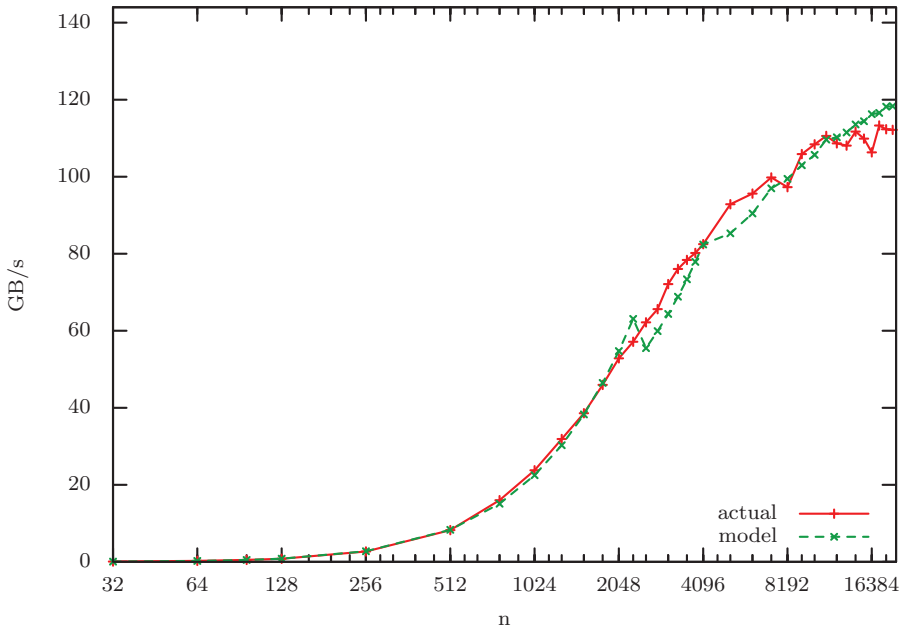

FIG. 8. *Comparison of the performance model with the actual results for solving* $Lx = b$.

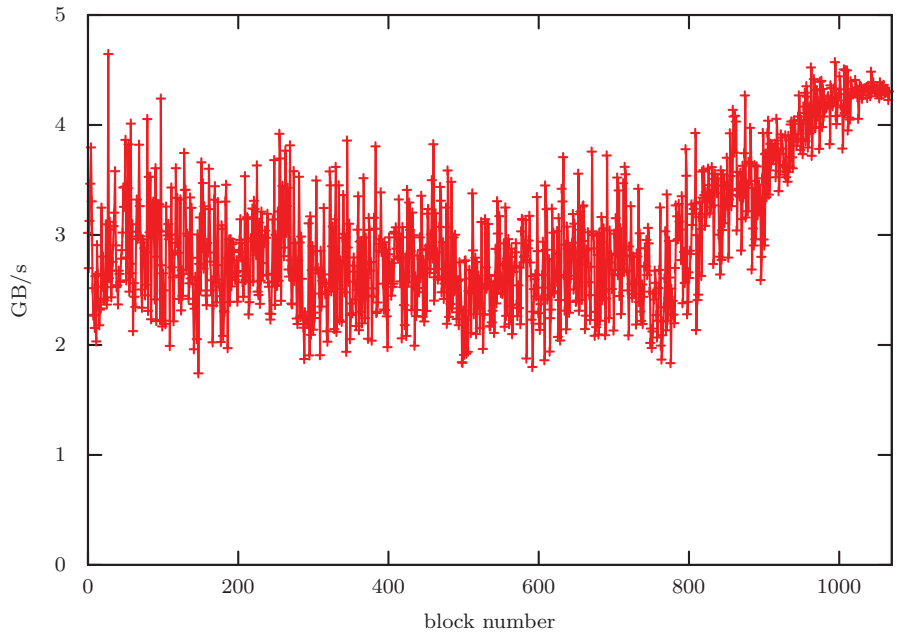

FIG. 9. *Per-block bandwidth for blocks in the critical slot with* $n = 10240$ *on a C2050 for* $Lx = b$.

6. Results on other GPUs. To demonstrate performance of the algorithm on other cards, in Figure 10 we present results on an M2090, which offers a higher maximum bandwidth than the C2050 (177 rather than 144 GB/s). We see that the maximum achieved bandwidth is 135.2 GB/s, giving the same 76% of peak as seen on the C2050.

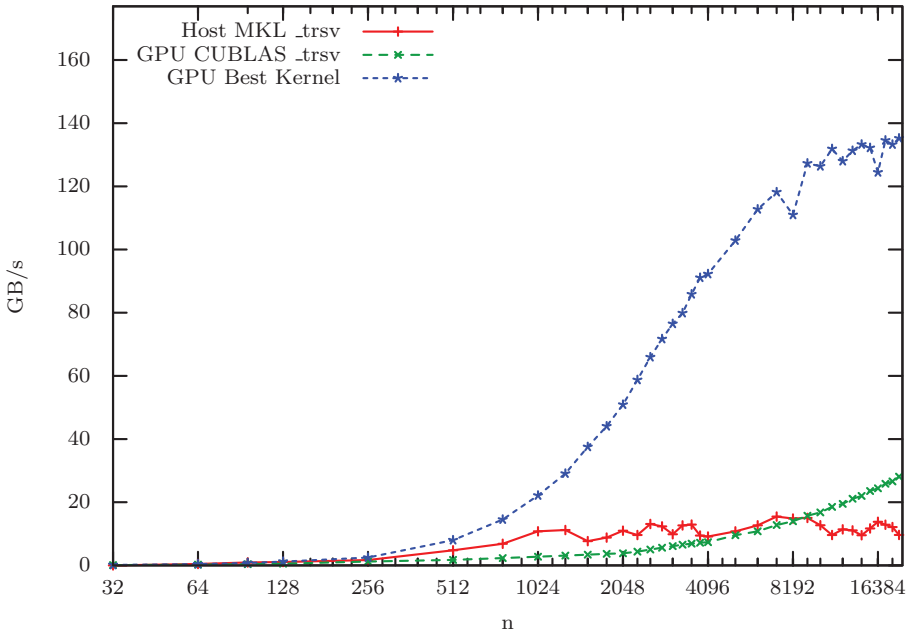

FIG. 10. *Performance of* `dtrsv` *on M*2090 *(theoretical peak bandwidth* 177 *GB/s).*

We also present results on the new Kepler architecture using a GTX680 card featuring a GK104 chip. As this chip has very limited double precision performance (compared to the GK110 used in the newest K20 cards) we present results for single precision only. Doing so has a number of consequences for performance. The first is that data takes half the space, and we can fit 9 blocks per SM rather than 4. However, the GK104 only has 8 SMs, rather than 14 on the C2050 (16 on the M2090), so this mostly balances out. Second, the bandwidth required for a block is halved, while the latency is unchanged. The upshot is that the first 4 blocks running in each slot are latency bound, rather than merely the first 1 or 2 as seen on Fermi. This leads to the latency-bound blocks dominating the performance costs in our models.

Two lines are plotted in Figure 11: the first shows results for a straightforward port of the memory-sychronized kernel, while the second shows performance when the `__threadfence()` instruction on line 38 of Listing 3 is omitted. This demonstrates a major difference between the Fermi and Kepler cards: this instruction incurs a much higher performance penalty on the latter.[1] However, this instruction is necessary for the program to guarantee calculating the correct answer (though we never observed such a breakdown in our tests). We break down the latency cost per block on Fermi and Kepler in Table 6.1.

7. Transpose problem. The transpose problem can be approached in a similar manner to the nontranspose case so far treated. However, to achieve maximal memory bandwidth it is necessary for threads of a warp to read entries in a rowwise rather than columnwise fashion (see Figure 12).

Using this access pattern means that instead of the simple $32 \times 4 \rightarrow 32 \times 1$ reduction operation (where each thread of warp 0 sums 4 numbers), a $32 \times 32 \rightarrow 32 \times 1$

[1] NVIDIA have indicated that their tests show this only affects the GK104 (primarily aimed at the consumer market) and not the GK110 chip (intended for HPC).

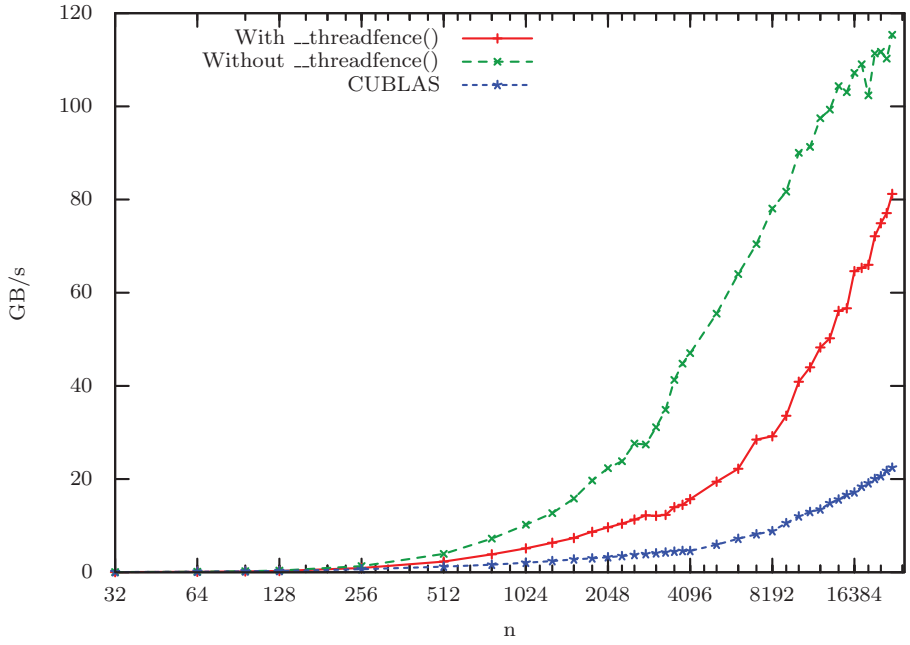

FIG. 11. *Performance of* `strsv` *on GTX*680 *(theoretical peak bandwidth* 192.2*) with and without* `__threadfence()`.

TABLE 6.1

*Derivation of latency costs on Fermi (C*2050*) and Kepler (GTX* 680*), single precision. Line numbers refer to Listing* 3*. Times are in clocks unless otherwise indicated (seconds = clocks/frequency, where frequency* $= 1.15 \times 10^9$ *for Fermi and* 1.00×10^9 *for Kepler).*

Task	Fermi	Kepler
Global memory latency	800–2000	250–500
Synchronization (line 28)	90–105	50–60
Subdiagonal block + shmem store (lines 29–33)	800-1100	550-750
Reduction from shared memory (line 35)	160–190	150–175
Multiply with inverse of diagonal block (line 36)	800-1000	600–900
Global memory store + __threadfence() (lines 37–38)	150–200	7000–15500
Increment sync[0] (line 40)	300-350	110–130
Total (clocks)	3100–4945	8710–18015
Total (seconds)	3.5×10^{-6}	1.3×10^{-5}

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

Nontranspose 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

FIG. 12. *Nontranspose 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.*

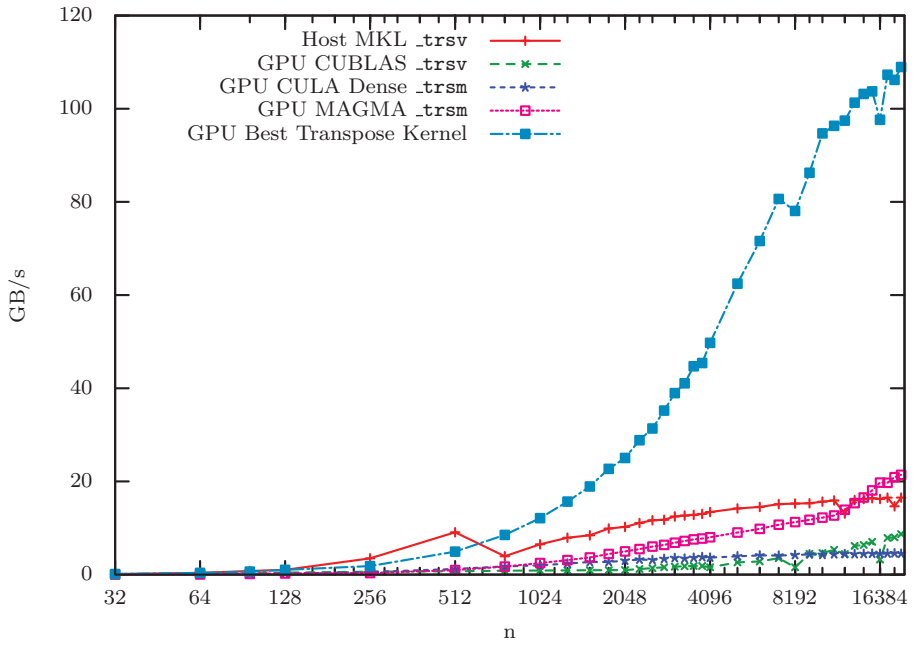

FIG. 13. *Performance of best kernels adapted for transpose solve. Times are measured using* `clock_gettime()`.

reduction is required. To maintain the same occupancy level (4 blocks per SM) we must limit ourselves to a 32×4 block of shared memory in which to perform this operation. Given these constraints, the most efficient way we found was to perform 8 separate reduction operations each of which is a $4 \times 32 \to 4 \times 1$ reduction.

As this operation is considerably slower than the equivalent step in the nontranspose case, we removed it from the critical path. This was achieved by performing a transpose operation on the blocks belonging to the critical path as part of the precaching process, and proceeding as per the nontranspose case. This yields the code of Listing 4, which should be compared with that of Listing 3.

The results of this approach are shown in Figure 13. Observe that the performance is slightly lower than for the nontranspose case, especially for smaller matrices, due to the additional overhead of the extra reduction. However, the code still significantly outperforms existing CUDA implementations.

8. Conclusions. By selecting the best algorithm developed in this paper for relevant domains of n values, a high performance `_trsv` implementation has been developed. Figures 14 and 15 demonstrate performance comparisons against other notable implementations on the C2050.

By analyzing 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 CUBLAS v4.1 implementation. Explicit prefetching 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 utilization for large matrices, allowing over 75% of peak to be achieved.

In future work we plan to integrate the use of this code into the development of a GPU-hosted sparse direct solver. To do so, we will modify the code to provide

LISTING 4
Outline code for transpose global-memory synchronized _trsv.

```
1  /* Performs _trsv for Transposed Lower−triangular Unit matrices
2   * Requires trsv_init() to be called first to initialize sync[].
3   * Best performance on C2050 with 8 warps of 32 threads each. */
4  Subroutine trsv_trans_exec(L, x, sync[]):
5     /* Symmetrically partition L into nblk rows and nblk columns */
6
7     Warp 0, Thread 0:
8        row = atomicIncrement(&sync[1])
9     __syncthreads();
10
11    All Warps:
12       Precache transposed L(row,row) into shared memory
13       Precache transposed L(row+1,row) into registers
14
15    Warp 0 only:
16       Local register val = x[threadIdx.x]
17    Other Warps:
18       Local register val = 0
19
20    All warps:
21       Local register array part_sum[] = 0 /* of size 32/blockDim.y */
22
23    For i = nblk:row+1
24       Warp 0, Thread 0:
25          Wait until sync[0] >= i
26       __syncthreads()
27       All Warps:
28          Perform local matrix−vector multiply part_sum[] += L(i,row)*x(i).
29    End For
30    For i = 0:32/blockDim.y−1
31       Copy part_sum[i] into shared memory
32       __syncthreads()
33       Warp 0, Threads i:i+3
34          Reduce 32 entries from shared memory into val
35       __syncthreads()
36    End For
37    Warp 0, Thread 0:
38       Wait until sync[0] >= i
39    __syncthreads()
40    Perform local matrix−vector multiply val += L(row+1,row)*x(i)
41    Store val into shared memory work array
42    __syncthreads()
43    Warp 0:
44       Perform reduction across shared memory work array
45       dblkSolve() on diagonal block L(row, row)
46       Store x(row) to global memory
47       __threadfence()
48       Thread 0:
49          sync[0]++ // Notify other thread blocks we have finished
50 End Subroutine trsv_trans_exec
```

a combined `_trsv` and `_trsm` functionality corresponding to the omission of one or more columns from the triangular solve algorithm. Further, to cope with the available parallelism within a sparse solve, the ability to run multiple such kernels in parallel to maximize SM occupancy will be explored.

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/.

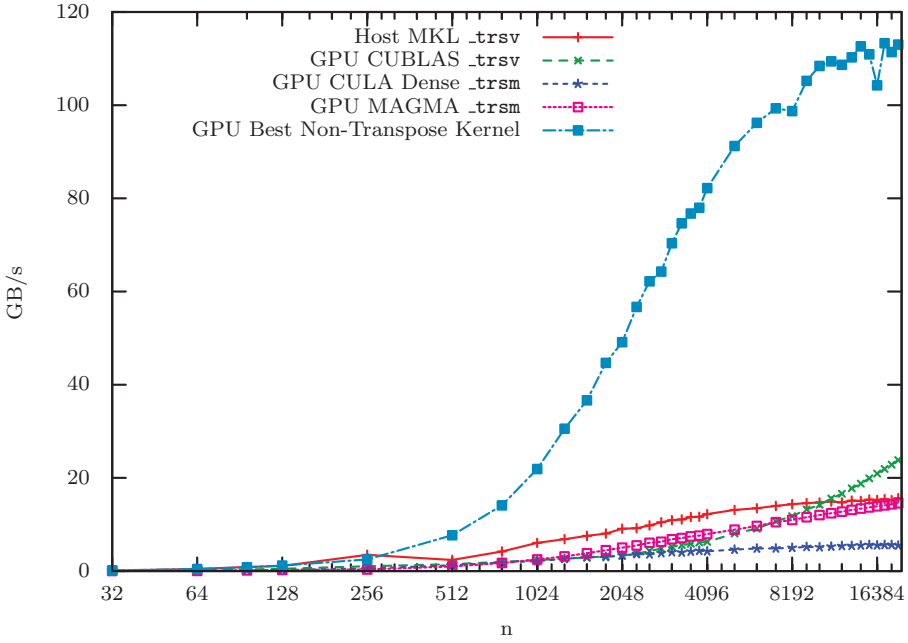

FIG. 14. *Comparison of bandwidth achieved using the best kernels from this paper against Host MKL* `_trsv` *and GPU CUBLAS* `_trsv`*, GPU CULA Dense* `_trsm`*, and GPU MAGMA* `_trsm` *implementations.*

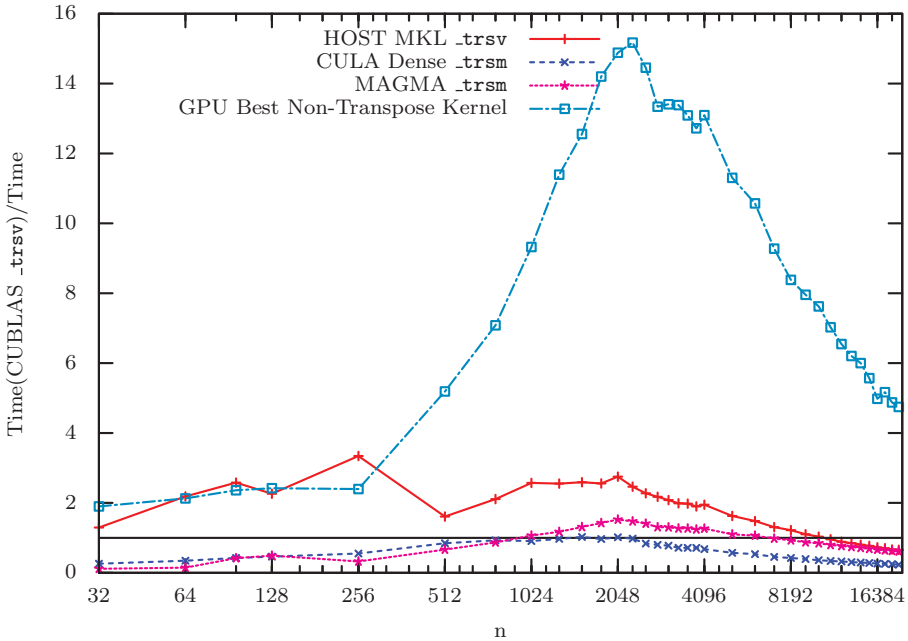

FIG. 15. *Comparison of speedup against GPU CUBLAS* `_trsv` *for best kernels from this paper, Host MKL* `_trsv`*, GPU CULA Dense, and GPU MAGMA* `_trsm`*.*

Acknowledgments.

- Mike Giles and Jennifer Scott provided comments on drafts of this paper. Further credit goes to Mike for the observation that registers can be used to precache the rectangular block on the critical path for the global memory version, allowing high occupancy.
- Philippe Vandermersch of NVIDIA engaged in useful exchanges regarding the code and some performance tweaks.
- Oxford Supercomputing Centre and the e-Infrastructure South Centre for Innovation provided access to their facilities, including their EMERALD HPC facility (EPSRC Grant EP/K000144/1, EP/K000136/1).
- Two anonymous referees provided helpful comments and suggestions.

REFERENCES

[1] EM Photonics, *CULA Dense*, http://www.culatools.com/dense/ (December 2012).
[2] N. J. Higham, *Stability of parallel triangular system solvers*, SIAM J. Sci. Comput., 16 (1995), pp. 400–413.
[3] J. D. Hogg and J. A. Scott, *A note on the solve phase of a multicore solver*, Technical Report RAL-TR-2010-007, Rutherford Appleton Laboratory, Chilton, Oxfordshire, England, 2010.
[4] X. S. Li, *Evaluation of SuperLU on multicore architectures*, J. Phys. Conf. Ser., 125 (2008), 12079.
[5] 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 Science—VECPAR 2010, J. Palma, M. Daydé, O. Marques, and J. Lopes, eds., Lecture Notes in Comput. Sci. 6449, Springer Berlin, 2011, pp. 93–101.
[6] R. Nath, S. Tomov, and J. Dongarra, *BLAS for GPUs*, in Scientific Computing with Multicore and Accelerators, J. Kurzak, D. A. Bader, and J. Dongarra, eds., CRC Press, Boca Raton, FL, 2010.
[7] NVIDIA, *CUDA Toolkit* 4.1 *CUBLAS Library*, https://developer.nvidia.com/cuda-toolkit-41-archive (January 2012).
[8] 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, 2009, ACM, pp. 9:1–9:10.
[9] H. H. B. Sørensen, *Auto-tuning of level* 1 *and level* 2 *BLAS for GPUs*, Concurrency and Computation: Practice and Experience, 25 (2013), pp. 1183–1198