# A High Performance SYMV Kernel on a Fermi-core GPU

Toshiyuki Imamura[1,2,4], Susumu Yamada[3,4], and Masahiko Machida[3,4]

[1] University of Electro-Communications, Chofu-shi, Tokyo 182-8585, Japan
[2] RIKEN Advanced Institute for Computational Science, Kobe-shi, Hyogo 650-0047, Japan
[3] CCSE, Japan Atomic Energy Agency, Kashiwa-shi, Chiba 277-8587, Japan
[4] CREST, Japan Science and Technology Agency

**Abstract.** A high-performance SYMV kernel is implemented on Fermi-core GPUs using an atomic-operation based algorithm. The algorithm is effective for the memory bandwidth and reduced memory usage. On a Tesla C2050, sustained double-precision and single-precision performances of approximately 43 GFLOPS and 78 GFLOPS, respectively, were achieved. The proposed SYMV kernel also performs on a GeForce GTX580 with 72 GFLOPS and 128 GFLOPS in the double-precision and single-precision modes, respectively. The proposed SYMV kernel outperforms major CUDA BLAS kernels, CUBLAS, MAGMABLAS, and CULA-BLAS. This performance improvement has a significant impact when the SYMV kernel is plugged into user codes.

## 1 Introduction

In the development of our eigensolver [1], it was very difficult to speed up Householder tridiagonalization. Detailed cost analysis revealed the cost of the kernel SYMV to be extremely high, and cost reduction is a very important problem. With emerging GPGPU, costly kernels such as SYMV and SYR2K in Householder tridiagonalization can be accelerated. Nath et al. [2] reported the optimization of the SYMV kernel in their MAGMABLAS library, which had faster performance than the CUBLAS library. Consequently, their Householder tridiagonalization routine, magma_dsytrd, performs very well. In the present paper, we present a new implementation of the SYMV kernel in order to realize a new eigenvalue solver, and we demonstrate the performance of this kernel on a single Fermi core GPU, such as an NVIDIA Tesla C2050 or an NVIDIA GeForce GTX580.

## 2 SYMV Kernel

The SYMV is a kernel function for a symmetric-matrix vector product and is categorized in Level 2 BLAS. Since the cost of operation and the amount of data in SYMV are each $O(n^2)$, performance bounds are based on memory bandwidth.

M. Daydé, O. Marques, and K. Nakajima (Eds.): VECPAR 2012, LNCS 7851, pp. 59–71, 2013.

It is difficult to achieve sufficient performance on a general CPU, on which the memory bandwidth is limited to at most 30 or 40 [GB/s]. Therefore, better performance is expected to contribute to a wider global memory bandwidth of the GPU (140 [GB/s] on a GTX280, for example). Compared with other Level 2 kernels (GEMV, for example), the symmetry of the matrix helps to reduce data accesses between memory and processor. In other words, improving the algorithm is expected to provide higher performance.

### 2.1 SYMV Algorithms

Taking symmetry into account, the Fortran code can be written as follows:

```
w(1:n)=0
do i=1,n
   y0=0
   do j=1,i-1
      y0  =y0  +a(j,i)*x(j)
      w(j)=w(j)+a(j,i)*x(i)
   enddo
   y(i)=y0+a(i,i)*x(i)
enddo
y(1:n)=y(1:n)+w(1:n)
```

The vector $w$, which can be replaced by $y$, is introduced in order to clarify the CUDA algorithm. Based on the source lines shown above, the framework of the SYMV algorithm specified with the 'U' option for a CUDA environment is presented in Figure 1. The algorithm is divided primarily into three kernels: pre-processing (corresponding to w(1:n)=0)), post-processing (corresponding to y(1:n)+=w(1:n)), and the main process. The main kernel consists of three parts, calculation on non-diagonal blocks, calculation on a diagonal block, and sumup of the registers on each thread. In Figure 1, the outer-most loop represented by counter `i` is expanded UX times, and the thread-block is organized into a one-dimensional array of threads, where BLOCK_SIZE is the number of threads issued.

Since Figure 2 shows the data access pattern for a specific thread-block represented in block.id, data updating of vector $w$ is performed by multiple thread-blocks. In order to secure updating of vector $w$, we need an exclusive control mechanism. There are several variations of implementations in which vector $w$ is multiplexed and exclusive (or mutex) control is fully obligated. In the remainder of this section, we would like to explain three algorithms with regard to an exclusive control mechanism on the SYMV kernel.

### 2.2 Atomic Algorithm

The algorithm shown in Figure 3 (top) uses atomic operations or a mutex mechanism, and so is referred to as the *Atomic algorithm*. Since CUDA 4.x does not

```
<01> // variable n refers to the dimension of the matrix A.
<02> // variables y_{*} and w_{0} refer to registers.
<03> // array s is on shared memory.
<04> // assume blockDim>=UX.
<05> // sumup adds up the value of the specified register in a block.
<06> kernel kernel_preprocess
<07>   set j := thread.id + block.id * block.Dim.
<08>   if j < n then
<09>     w[j] := 0.
<10>   endif
<11> endkernel
<12> kernel kernel_main
<13>   define j ≡ j̃ + thread.id.
<14>   foreach (thread, block) do
<15>     set i:=UX*block.id.
<16>     y_{0} := ... := y_{UX-1} := 0.
<17>     // part one / sweep along the column block
<18>     CORE of either the Algorithm Atomic, Blocked, or Ticket in Fig. 3
is called here.
<19>     // part two / calculation on a diagonal part
<20>     for j̃:=thread.id to UX-1 do
<21>       s(thread.id, j) := s(j, thread.id) := a(i+thread.id, i + j).
<22>     endfor
<23>     sync threads in a block
<24>     if thread.id<UX then
<25>       y_{k} += s(thread.id, k) * x[i + k]  for k ∈ [0, UX).
<26>     endif
<27>     // part three
<28>     s(k, thread.id) := sumup(y_{k})  for k ∈ [0, UX).
<29>     if thread.id<UX then
<30>       y[i + thread.id] += s(thread.id, thread.id).
<31>     endif
<32>   endfor
<33> endkernel
<34> kernel kernel_postprocess
<35>   set j := thread.id + block.id * block.Dim.
<36>   if j < n then
<37>     y[j] += w[j], or y[j] += ∑ w(j, :).
<38>   endif
<39> endkernel
```

**Fig. 1.** Framework of the SYMV Algorithm

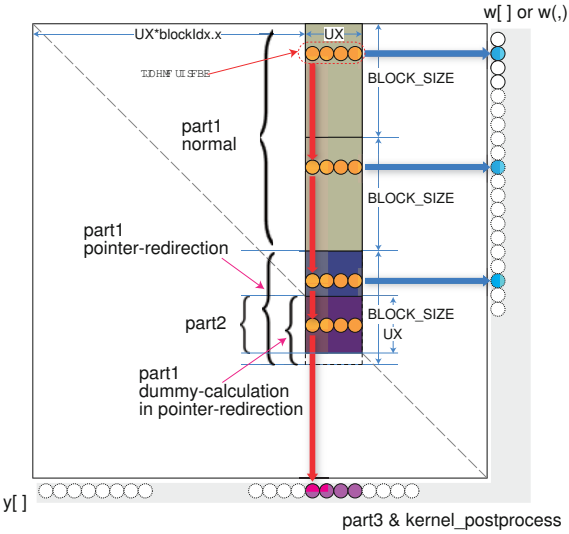


**Fig. 2.** Schematic diagram of the data access pattern on the SYMV kernel

support atomicAdd for double precision, critical section controls (mutex lock and unlock) are implemented using `atomicCAS` and `atomicExch` functions.

In our implementation, the atomic functions are issued on the master threads to avoid incurring the serialization of all working threads. Thus, loss of thread serialization can be minimized. Following the CUDA thread model, it is reasonable to secure the update of vector $w$, even if the operations to be in the Atomic algorithm are relatively small. Consequently, we expect that the total cost on exclusive controls is reduced. This lock-and-unlock mechanism is a general model and can be applied to other BLAS kernel implementations, such as GEMV and spMV, to reduce memory usage on working buffers.

### 2.3 Blocked Algorithm

Figure 3 (middle) shows the variation for exclusive control of the Atomic algorithm. This algorithm adds a second index to the variable $w$ and requires no exclusive control. Thus, data access to $w$ is implicitly blocked, and we refer to this algorithm as the *Blocked algorithm*. This is also known as the 'scatter and gather technique'. From the viewpoint of the multiplicity of vector $w$, this is similar to the algorithm adopted in MAGMABLAS when UX=1.

### 2.4 Ticket Algorithm

On the other hand, we can generate another intermediate algorithm between the Atomic algorithm and the Blocked algorithm. In Figure 3 (bottom), vector $w$

```
// CORE of the Atomic Algorithm
    for j̃:=0 to i − 1 − thread.id step BLOCK_SIZE do
      y_{k} += a(j, i + k) ∗ x[j]  for k ∈ [0, UX).
      w_{0} := Σ_{k∈[0,UX)} a(j, i + k) ∗ x[i + k].
      // equivalent to atomic (w[j] += w_{0}) in the CUDA semantics.
        mutex_lock @ the masterthread and syncthreads.
        w[j] += w_{0}.
        syncthreads and mutex_unlock @ the masterthread.
    endfor


// CORE of the Blocked Algorithm
  for j̃:=0 to i − 1 − thread.id step BLOCK_SIZE do
    y_{k} += a(j, i + k) ∗ x[j]  for k ∈ [0, UX).
    w_{0} := Σ_{k∈[0,UX)} a(j, i + k) ∗ x[i + k].
    w(j, block.id) += w_{0}.
  endfor


// CORE of the Ticket Algorithm
  ticket_id = get_ticket_id( ).
  for j̃:=0 to i − 1 − thread.id step BLOCK_SIZE do
    y_{k} += a(j, i + k) ∗ x[j]  for k ∈ [0, UX).
    w_{0} := Σ_{k∈[0,UX)} a(j, i + k) ∗ x[i + k].
    w(j, ticket_id) += w_{0}.
  endfor
// 'int Ticket_id_master' is initialized at the preprocessing step.
__device__ int function get_ticket_id( )
  __shared__ int shred_buff;
  if is_master_thread then
    shared_buff := atomicInc( &Tickect_id_master, gridDim.x );
  endif
  syncthreads; block_id := mod(shared_buff, M); syncthreads
  return block_id
end function
```

**Fig. 3.** Core Algorithms (Top: Atomic, Middle: Blocked, Bottom: Ticket)

is multiplexed $M$ times ($M$ should be a multiple of the number of SM's). This limits mutex control among activated thread-blocks and consequently reduces the number of atomic operations. In this case, since exclusive control is open not only for a single thread-block but also for multiple thread-blocks, this algorithm is similar to a seat reservation model, such as for reserving train tickets. Therefore, this algorithm is referred to as the *Ticket algorithm*.

### 2.5 Another Kernel Implementation (L+U Algorithm)

There is another implementation of SYMV. A symmetric matrix $A$ is represented by the sum of upper and lower triangle matrices by taking into account the symmetry of the matrix

$$A = L + D + U = \tilde{L}(= L + D) + U(= L^t).$$

Using the decomposition, SYMV can be calculated by $Ax = \tilde{L}x + Ux$. We refer to the above expression as the *L+U algorithm*, which can be easily implemented using GEMV kernels modified for upper and lower triangle matrices. In the present paper, the GEMV kernel codes presented in [3] are modified and used for the SYMV kernel. Since this algorithm requires two kernels, the upper triangular part ($Ux$) and the lower triangular part ($\tilde{L}x$), the overhead to start up GPU kernels is quite small. Therefore, the L+U algorithm offers better performance when the matrix dimension is small.

### 2.6 Pointer Redirection Optimization

Pointer redirection [2] is an optimization technique for CUDA programming that produces coherent running threads on a specific loop. When all of the threads in a thread-block proceed with the loop and a number of these threads access out of bound on array accesses, invalid pointers are modified in order to access proper addresses. In the present case, invalid accesses to matrix $a$ and vector $x$ are adjusted to access their top element. In the case of vector $w$, an invalid pointer is redirected to a dummy variable on global memory in order to protect $w$ from an invalid overwrite.

## 3 Experiment Results

The primary experiments were conducted on an NVIDIA Tesla C2050 GPU and an NVIDIA GeForce GTX580, and we use an NVIDIA GeForce GTX280 for a preliminary test. Their specifications and software environment are summarized in Table 1. All of the performance tests include only kernel execution without host-device data transfer. In other words, CUDA BLAS kernels operate in the non-thunking mode. The performance parameters (BLOCK_SIZE, UX) are chosen to be (256, 16) and (128, 26) for the DP and SP modes, respectively.

**Table 1.** Hardware and software specifications of GPU's used in the present study (*Theoretical peak performance is referred from [5])

| | Tesla C2050 | GTX580 | GTX280 |
|---|---|---|---|
| The core architecture | Fermi | Fermi | GT200 |
| The number of CUDA cores | 448 | 512 | 240 |
| Processor core clock [GHz] | 1.15 | 1.544 | 1.296 |
| Peak performance [DP/SP GFLOPS]* | 515/1030 | 393/1573 | 78/933 |
| Memory capacity [GB] | 3 | 1.5 | 1 |
| Memory bandwidth [GB/s] | 144 | 192.4 | 141.7 |
| Host CPU | Core i7-860 | Core i7-2600K | Phenom 9750 |
| Frequency [GHz] | 2.8 | 3.4 | 2.4 |
| CUDA Compute Capability | 2.0 | 2.0 | 1.3 |
| CUDA version | 4.0 | 4.1 | 4.1 |
| NVIDIA Linux driver | 275.09.07 | 295.71 | 295.71 |
| GNU gcc version | 4.4.5 | 4.5.3 | 4.5.3 |

### 3.1 Preliminary Performance Prediction

We first theoretically discuss the performance bound for the SYMV. As a result of symmetry, memory access to an element of matrix $A$ corresponds to two multiply-add operations. Thus, the memory requirement per DP operation is computed by 'BF' := sizeof(element)[Byte]/4[flop] = 2 [Byte/flop]. Based on the benchmark reports, e.g., [4], the sustained memory bandwidth of a C2050 with ECC switched on is calculated to be approximately 99 [GB/s]. The optimal SYMV performance is 99/BF [GFLOPS]. Therefore, the SYMV kernel is bounded by 49.5 and 99 [GFLOPS] in cases of the DP and SP modes, respectively.

### 3.2 Comparison of Four Algorithms

The Atomic, Ticket, and Blocked algorithms have no difference in the core computation part, except for exclusive operation and sumup of vector $w$. Table 2 summarizes the differences in parts 1 and 3 of these algorithms. Table 2 indicates that the Atomic algorithm has a significant advantage with respect to the cost of $w$, the computational cost, and memory usage. In contrast, the Atomic algorithm requires far more atomic operations than other algorithms.

Here, we should have two scenarios according to the cost of the atomic operations. If the cost of atomic operations is too high, we expect *Atomic* > *Ticket* < *Blocked*. Thus, the Ticket algorithm has an advantage in such cases. On the other hand, if the cost of atomic operations is small, we expect *Atomic* < *Ticket* < *Blocked*, which leads us to select the Atomic algorithm.

Table 3 shows the elapsed time and the overhead cost for the above three algorithms as well as the L+U algorithm on an NVIDIA Tesla C2050. Tables 2 and 3 suggest that the overhead cost is proportional to the memory usage of $w$. This tendency was reported by Nath et al. [2], and the results of the present

**Table 2.** Complexity analysis for three algorithms ($L \equiv \lceil N/\text{BLOCK\_SIZE} \rceil$)

| | words of extra data | cost of sumup $w$ | max(atomic ops. per thread-block) |
|---|---|---|---|
| Atomic | $N + L$ | $N$ | $2L$ |
| Ticket | $NM + M$ | $NM$ | 2 |
| Blocked | $N\lceil N/\text{UX} \rceil$ | $N\lceil N/\text{UX} \rceil$ | 0 |

**Table 3.** Analysis of SYMV calculation for four algorithms in the DP mode on a Tesla C2050 (top: total time [s], bottom: overhead time [s])

| | matrix dimension | | | | |
|---|---|---|---|---|---|
| | 1,000 | 2,000 | 4,000 | 6,000 | 8,000 |
| Atomic | .252E-3 | **.395E-3** | **.968E-3** | **.189E-2** | **.319E-2** |
| | .118E-3 | .126E-3 | .127E-3 | .119E-3 | .124E-3 |
| Ticket | .429E-3 | .611E-3 | .127E-2 | .234E-2 | .380E-2 |
| | .306E-3 | .343E-3 | .383E-3 | .407E-3 | .450E-3 |
| Blocked | .426E-3 | .632E-3 | .160E-2 | .294E-2 | .492E-2 |
| | .266E-3 | .376E-3 | .694E-3 | .103E-2 | .159E-2 |
| L+U | **.139E-3** | .406E-3 | .147E-2 | .321E-2 | .566E-2 |

**Table 4.** Analysis of SYMV calculation for four algorithms in the DP mode on a GeForce GTX280 (top: total time [s], bottom: overhead time [s])

| | matrix dimension | | | | |
|---|---|---|---|---|---|
| | 1,000 | 2,000 | 4,000 | 6,000 | 8,000 |
| Atomic | .275E-2 | .514E-2 | .914E-2 | .110E-1 | .149E-1 |
| | .387E-4 | .364E-4 | .399E-4 | .405E-4 | .400E-4 |
| Ticket | .398E-3 | .579E-3 | **.119E-2** | **.211E-2** | **.347E-2** |
| | .236E-3 | .264E-3 | .305E-3 | .360E-3 | .415E-3 |
| Blocked | .346E-3 | .556E-3 | .142E-2 | .270E-2 | .454E-2 |
| | .236E-3 | .289E-3 | .503E-3 | .855E-2 | .139E-2 |
| L+U | **.167E-3** | **.470E-3** | .163E-2 | .361E-2 | .624E-2 |

study are similar. Furthermore, the cost of atomic operations is not significant compared to the total computational time. Table 4 also shows the elapsed time and the overhead cost for an NVIDIA GeForce GTX280, which is prior to the Fermi GPU core architecture. Since the costs of the Ticket algorithm and the Blocked algorithm on a GeForce GTX280 are almost equivalent to those on a Tesla C2050, the large performance difference between a Tesla C2050 and a GeForce GTX280 stems from the cost of the atomic operations. A technical paper by NVIDIA reported that a Fermi core adopts fast and greatly improved atomic memory operations, compared to GT200 cores [6], and that the cost of atomic operations on a GTX280 is quite high. The Ticket algorithm is a better algorithm in such a case. Thus, the Atomic algorithm has better overall

performance than the Ticket and Blocked algorithms in the case of the current Fermi core architecture. However, if the cost imbalance between calculation and atomic operations is severe in a future architecture, the selected algorithm will differ.

Furthermore, Tables 3 and 4 suggest that the L+U algorithm performs with a tiny overhead. In fact, the L+U algorithm has better performance when the matrix dimension is smaller ($N = 1,000$). Therefore, on a Tesla C2050, we switch the algorithm in the DP mode, using the L+U algorithm when $N < 2,020$ and the Atomic algorithm when $N \geq 2,020$. In the case of the SP mode, the border between the L+U algorithm and the Atomic algorithm is $N = 2,990$.

### 3.3 Performance Test

In the present paper, we measured the performance of SYMV kernels for the present implementation (hybrid of the Atomic and L+U algorithms) and three major BLAS implementations: CUBLAS [7], MAGMABLAS [8] (run in the L-mode), and BLAS in CULA [9]. Here, the performance is calculated by $2N^2$/ *'elapsed time'*.

**Tesla C2050.** The performance on a single NVIDIA Tesla C2050, which is a high-end Fermi-core GPU card, is presented in Figure 4. The release versions of the CUDA BLAS's are as follows: CUBLAS R4.0, MAGMABLAS R1.2.1 (run in the L-mode), and CULABLAS R12. In [2], the performance on the SP mode for MAGMABLAS on a C2050 exceeds 80 GFLOPS; however, we could not obtain that performance in the present C2050 environment. In order to compare the throughput from the FLOPS rates fairly, the results on a C2050 are based on the performance measurement in the present paper.

In the DP and SP modes, the performance of the proposed SYMV kernel reaches 43 and 78 GFLOPS, respectively, when the matrix dimension is 18,000. These values represent speed-ups of approximately 2.5 times and 4.0 times, respectively, when CUBLAS is set as a baseline. Furthermore, performances of 86% and 78% of the upper bound, respectively, are achieved.

**GeForce GTX580.** The performance on a single NVIDIA GeForce GTX580, which is also a Fermi-core GPU, is presented in Figure 5. The release versions of the CUDA BLAS's are as follows: CUBLAS R4.1, MAGMABLAS R1.2.1 (run in the L-mode), and CULABLAS R14. In the DP and SP modes, the performance of the proposed SYMV kernel achieves 71 and 128 GFLOPS, respectively, when the matrix dimension is 12,000. These values represent speed-ups of approximately 2.8 times and 3.3 times, respectively, in comparison to CUBLAS.

### 3.4 Discussion and Related Research

Compared with other implementations of BLAS, small fluctuations in the SYMV appear upon varying the matrix dimension, whereas the performance of the

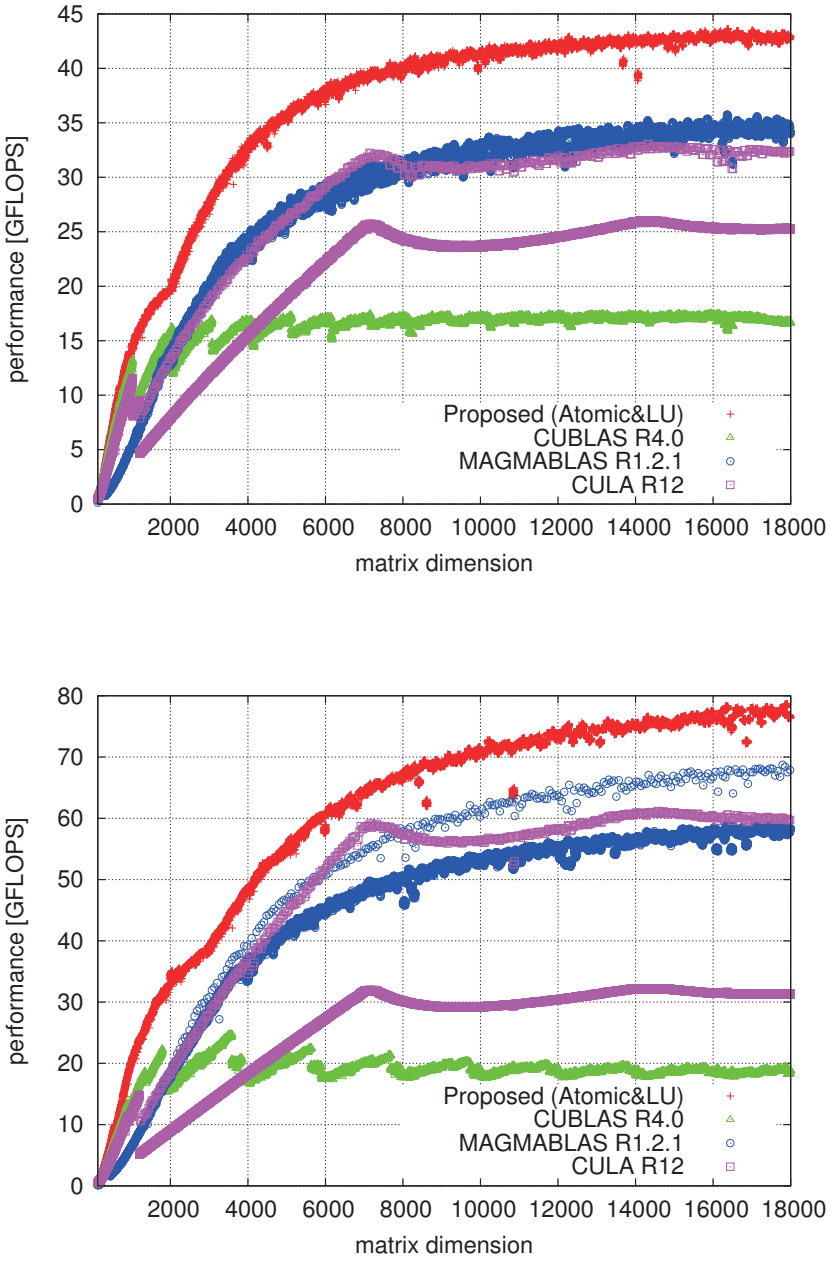


**Fig. 4.** Performance results on a Tesla C2050 in non-thunking mode (Top: DSYMV = DP mode, bottom: SSYMV = SP mode)

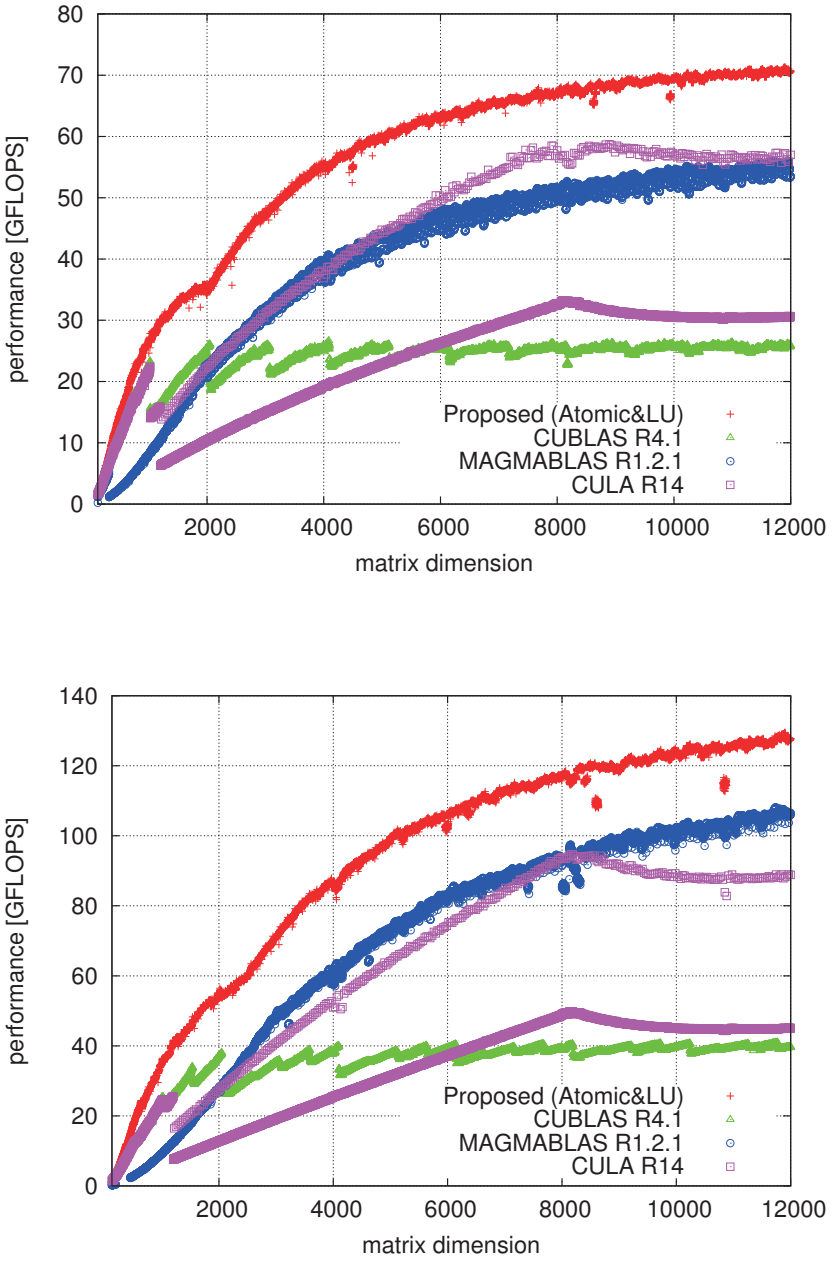


**Fig. 5.** Performance results on a GeForce GTX580 in non-thunking mode (Top: DSYMV = DP mode, bottom: SSYMV = SP mode)

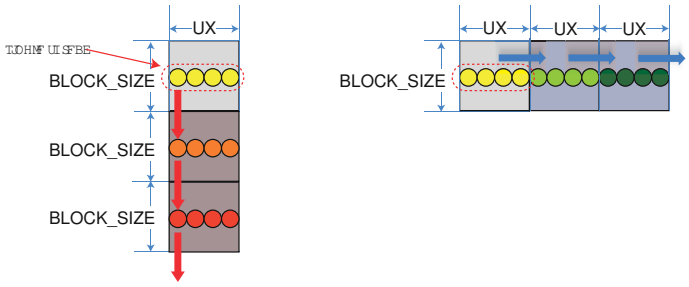


**Fig. 6.** Access patterns in GEMV-T (left) and GEMV-N (right)

present implementation is the most stable. CUBLAS exhibits performance with a sawtooth profile, and CULABLAS behaves irregularly for multiples of 32 or 64 dimensions, which is also true for MAGMABLAS on a C2050 in the SP mode. The proposed SYMV kernel also has a small fluctuation, especially in the SP mode. Since the period is observed to be equivalent to the size of UX, we recognize that load imbalance among thread-blocks affects this phenomena. The fluctuation reaches approximately 3% and is negligible.

Investigating the difference in the performance profile by reading the source code of MAGMABLAS reveals that MAGMABLAS is realized by the GEMV-N-based algorithm (access through the same row direction). On the other hand, the GEMV-T algorithm (access through the same column direction) used in the present study is the enhanced version presented in [3]. Their typical access patters are shown in Figure 6. This also causes a difference in performance between MAGMABLAS and the proposed SYMV kernel.

GLAS by Sørensen [10,11] is another CUDA BLAS implementation that uses atomic operations. GLAS adopts atomic operations not in a SYMV kernel but rather in a GEMV-N kernel at the current release, wherein `atomicAdd` or equivalent functionality is emulated. GLAS's GEMV implementation on a Tesla C2050 is optimized by an automatic tuning technique and achieved approximately 90% of the memory bandwidth (performance upper bound) in the SP mode.

For the Tesla C2050 and the GeForce GTX580, the implementation of the proposed SYMV kernel is sufficiently optimized. We conclude that the proposed SYMV algorithm is, overall, the most stable and fastest. The Atomic algorithm used in the current implementation is a powerful technique in CUDA GPGPU programming for the Fermi generation GPU architecture.

## 4 Conclusion

We have presented an optimal implementation of the GPU kernel for symmetric-matrix vector multiplication, referred to as the SYMV kernel. The proposed SYMV kernel uses the Atomic algorithm, which requires very little extra working memory. In the DP and SP modes, the proposed SYMV kernel performs at 43 and 78 GFLOPS on a Tesla C2050, respectively. The implementation of the SYMV

kernel is herein demonstrated to provide remarkable memory consumption and performance. Since a generic processor performs at from 2 to 5 GFLOPS on DSYMV, the impact of the proposed SYMV kernel is enormous.

In the future, we would like to examine the proposed SYMV kernel on Kepler, which is a new GPU core architecture. Furthermore, we would like to apply the SYMV kernel to a previously proposed eigensolver [1].

**Acknowledgements.** The present study was supported in part by the Ministry of Education, Science, Sports and Culture through a Grant-in-Aid for Scientific Research (B), 21300013, and through a Grant-in-Aid for Scientific Research on Innovative Areas, 22104003.

## References

1. Imamura, T., Yamada, S., Machida, M.: Development of a High Performance Eigensolver on the Peta-Scale Next Generation Supercomputer System, the Atomic Energy Society of Japan. Progress in Nuclear Science and Technology 2, 643–650 (2011)
2. Nath, R., Tomov, S., et al.: Optimizing symmetric dense matrix-vector multiplication on GPUs. In: Proc. of the Intl. Conf. High Performance Computing, Networking, Storage and Analysis, SC 2011 (2011)
3. Imamura, T.: Performance-stabilization and automatic performance tuning for DGEMV routines on a CUDA environment. IPSJ Journal, Transaction of Advanced Computing Systems, ACS 4(4), 158–168 (2011) (in Japanese)
4. Schäfer, A., Fey, D.: High Performance Stencil Code Algorithm for GPGPUs. In: Proc. of ICCS 2011, Procedia Computer Science, vol. 4, pp. 2077–2036 (2011)
5. Hwu, W.W. (ed.): GPU Computing Gems Jade Edition (Applications of GPU Computing Series). Morgan Kaufmann (2011)
6. NVIDIA: whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi, `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf`
7. NVIDIA: CUDA CUBLAS Library, `http://developer.download.nvidia.com`
8. Agullo, E., Demmel, J., et al.: Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. J. of Physics: Conference Series 180 (2009)
9. Humphrey, J.R., Price, D.K., et al.: CULA: Hybrid GPU Accelerated Linear Algebra Routines. In: SPIE Defense and Security Symposium (DSS) (2010)
10. Sørensen, H.H.B.: Auto-tuning Dense Vector and Matrix-Vector Operations for Fermi GPUs. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) PPAM 2011, Part I. LNCS, vol. 7203, pp. 619–629. Springer, Heidelberg (2012)
11. GPUlab: GLAS library version 0.0.2, `http://gpulab.imm.dtu.dk/docs/glas_v0.0.2_C2050_cuda_4.0_linux.tar.gz`