# High-Performance GEMV and SYMV with Auto-Tuning for Performance Stabilization on Multiple GPU Generations

Large-scale Parallel Numerical Computing Technology Research Team
Advanced Institute of Computational Sciences (AICS), RIKEN
- 7-1-26 Minato-jima Minami-machi, Chuo-ku, Kobe, Hyogo, 650-0047, JAPAN
- Poster Presenter: Daichi Mukunoki (daichi.mukunoki@riken.jp)

❖ GPU Technology Conference (GTC2015), March 17-20, 2015

**Overview:**
We are developing high performance level-2 BLAS routines (GEMV and SYMV) with auto-tuning for NVIDIA GPUs. Our goal is to achieve best performance at any matrix size, on any given configuration (precision, real/complex, trans, uplo), and on any GPU architecture/product. Our GEMV and SYMV are implemented with different auto-tuning strategies. Both routines exhibit competitive performance in terms of both throughput and performance stability with respect to the matrix size on multiple GPU generations when compared to latest major implementations.

**Downloads:**
"ASPEN.K2: Automatic Stabilization and PErformance tuNing of level 2 BLAS Kernels"
http://www.aics.riken.jp/labs/lpnctrt/ASPENK2_e.html
❖ ASPEN.K2 is an open source software.

**Performance Evaluation:**
- Binaries of our routines are generated for all GPU architecture using "-gencode arch=compute_XX, code=sm_XX" (XX is the compute capability) and "–O3" option.
- We compared the performance with CUBLAS 6.5, KBLAS 1.2, and MAGMA 1.6.0. However, KBLAS and MAGMA do not support Maxwell. We built them on Maxwell with the configuration for Kepler.
- ❖ MAGMA: http://icl.cs.utk.edu/magma/, KBLAS: http://ecrc.kaust.edu.sa/Pages/Res-kblas.aspx

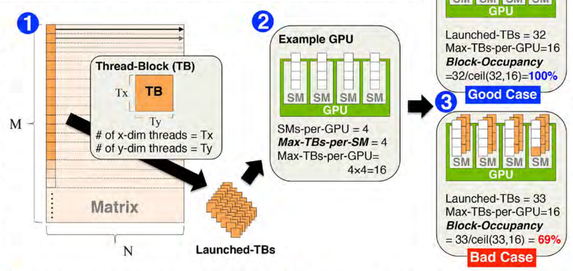
| | Host Machine (common) |
|---|---|
| CPU | Intel Core i7-4790 |
| OS | CentOS 7.0.1406 (3.10.0-123.13.1.el7.x86_64) |
| CUDA | Toolkit: 6.5.19 Driver: 343.19 |

| | GeForce GTX 980 | Tesla K20c | GeForce GTX 580 |
|---|---|---|---|
| Architecture | Maxwell (GM204, CC5.2) | Kepler (GK110, CC3.5) | Fermi (GF110, CC2.0) |
| Cores & SMs | 128 Cores × 16 SMs | 192 Cores × 13 SMs | 32 Cores × 16 SMs |
| Max TBs / SM | 32 | 16 | 8 |
| Bandwidth | 224 GB/s | 208 GB/s | 192.4 GB/s |

## GEMV (General Matrix-Vector Multiplication)

■ Authors: Daichi Mukunoki (RIKEN AICS), Toshiyuki Imamura (RIKEN AICS), Daisuke Takahashi (Univ. of Tsukuba)

**Overview:**
- Our GEMV routines are optimized with shared-memory blocking and register blocking along with 128-bit vector load/store instructions. Even though the implementation could approach actual peak memory throughput at some matrix sizes, performance fluctuates periodically depending on the matrix size.
- We could explain the reason for the performance fluctuations by introducing "*Block-Occupancy*" as a performance model, and then created an automatic thread-block size adjustment mechanism to avoid the performance fluctuations.
- Our method automatically determines the optimal thread-block size based on the theoretical performance model before launching kernel. It eliminates the need for trial executions to determine optimal parameters.




### 3 Block-Occupancy

$$\textit{Block-Occupancy} = \frac{\textit{Launched-TBs}}{\text{ceil}\ (\textit{Launched-TBs, Max-TBs-per-GPU})}$$

*ceil(x,y) rounds x up to the nearest multiple of y

- *Block-Occupancy* is occupancy of launched TBs to chunk of TB execution on GPU.
- We found low *Block-Occupancy* leads to performance degradation on the GEMV kernel (see "Actual Case" of SGEMV-N on Tesla K20c on left below)
- *Launched-TBs* depends on *M* (the number of rows of a matrix). *Max-TBs-per-GPU* depends on *SMs-per-GPU* and *Max-TBs-per-SM*. Thus, *Block-Occupancy* depends on hardware, kernel, and matrix size.

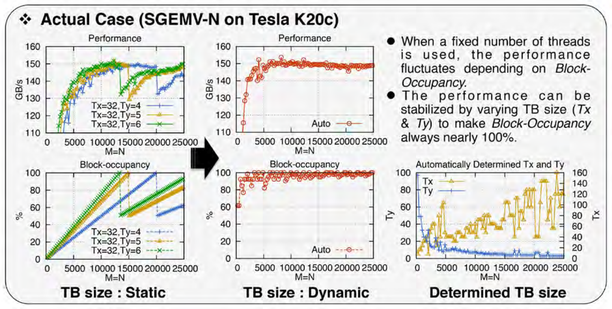
**References:**
1. D. Mukunoki, T. Imamura and D. Takahashi, "Fast Implementation of General Matrix-Vector Multiplication (GEMV) on Kepler GPUs," 23rd Euromicro International Conference on Parallel, Distributed and Network- based Processing (PDP 2015), 2015 (accepted).

**Acknowledgment:**
❖ This work was supported by JSPS Grant-in-Aid for JSPS Fellows No. 251290.

### 1 Computation with multiple TBs
- A matrix is computed by multiple two-dimensional thread-blocks (TBs) with *Tx*×*Ty* threads.
- The number of *Launched-TBs* increases in proportion to the number of rows of the matrix (*M*).

### 2 Multiple TBs are executed on multiple SMs
- An NVIDIA GPU consists of multiple streaming multiprocessors (SMs). Each SM is capable of executing multiple TBs simultaneously. (We call the number "*Max-TBs-per-SM*", it's also called "*active Thread-Blocks per Multiprocessor*" in CUDA Occupancy Calculator)
- The physical maximum of *Max-TBs-per-SM* is 8 on Fermi, 16 on Kepler, and 32 on Maxwell. However, they may be smaller depending on kernel attributes: TB size (*Tx* & *Ty*), shared-memory usage per TB, and register usage per thread.

### ❖ Actual Case (SGEMV-N on Tesla K20c)




- When a fixed number of threads is used, the performance fluctuates depending on *Block-Occupancy*.
- The performance can be stabilized by varying TB size (*Tx* & *Ty*) to make *Block-Occupancy* always nearly 100%.

TB size : Static | TB size : Dynamic | Determined TB size

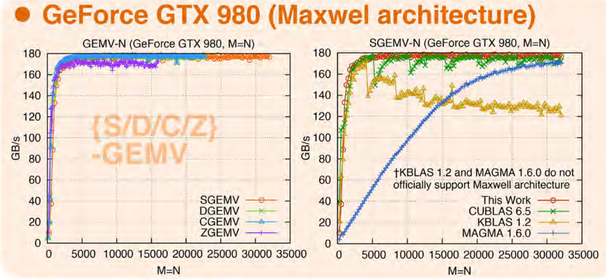
### 4 Rules for determining TB size (Tx and Ty)

| Rules (in order of priority) | | |
|---|---|---|
| 1. Block-Occupancy | As large as possible | |
| 2. Warp-Occupancy | As large as possible | |
| 3. Tx (x-dim threads per TB) | As large as possible | |
| 4. Ty (y-dim threads per TB) | As small as possible | |
| **Restrictions** | | |
| 128 ≦ Tx × Ty ≦ 1024 | | |
| Tx-min ≦ Tx ≦ 1024, Tx-min = 32†, Tx % Tx-step = 0<br>†Tx-min=Tx-step when launched-TBs < Max-TBs-per-GPU with Min-TBs-per-SM=1 | | |
| Max-TBs-per-SM ≧ Min-TBs-per-SM | | |
| **Architecture-dependent** | **Fermi** | **Kepler** / **Maxwell** |
| Tx-step | 16 (half-warp) | 128/sizeof(value)×(#of rows per thread) (128 is cache line size (128-bytes)) |
| Min-TBs-per-SM | 1 | 3‡ / 1 |

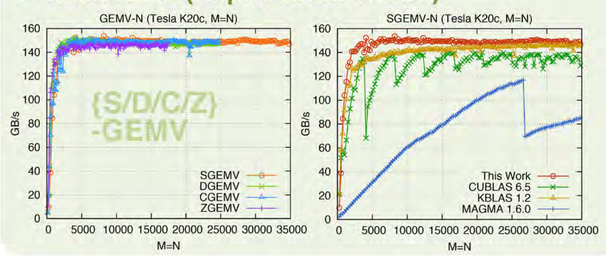
‡Min-TBs-per-SM = 2 or 1 when launched-TBs < Max-TBs-per-GPU

### 4 Automatic TB size adjustment for avoiding low Block-Occupancy
- In order to avoid performance degradation due to low *Block-Occupancy*, our routines adjust TB size (*Tx* & *Ty*) automatically under the rules (left table).
- *Tx* should be large to improve blocking efficacy. *Ty* should be small to decrease reduction cost. *Tx-step* is decided on the basis of the number of threads concurrently accessing memory.
- The process to determine the optimal TB size is performed in the GEMV routine every time before launching kernel.

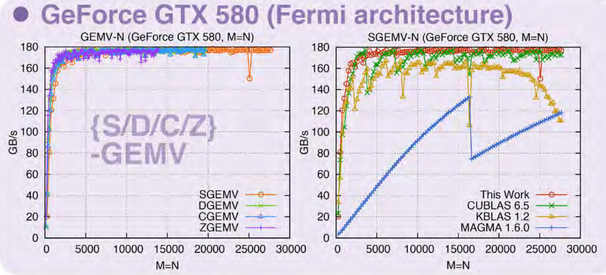
### ● GeForce GTX 980 (Maxwel architecture)



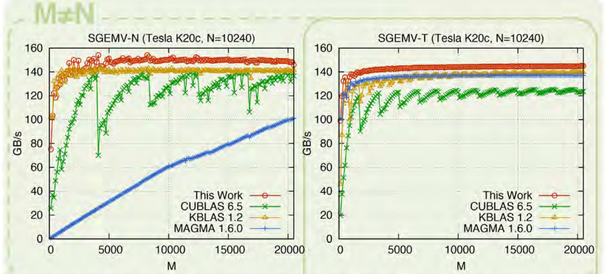

### ● Tesla K20c (Kepler architecture)

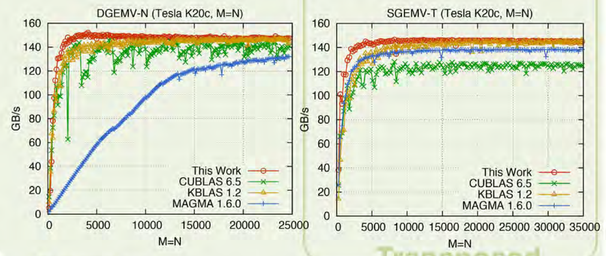



### ● GeForce GTX 580 (Fermi architecture)



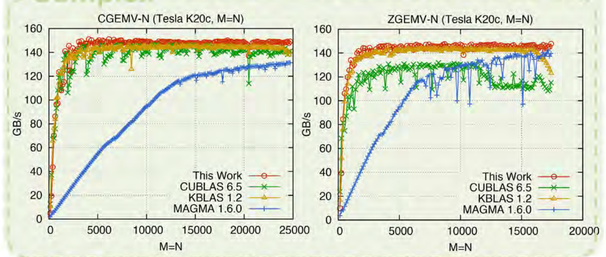

### M≠N




Transposed

### Complex

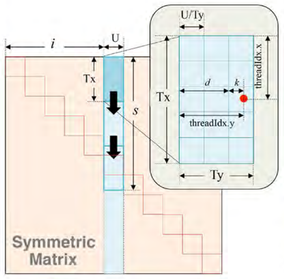



## SYMV (Symmetric Matrix-Vector Multiplication)

■ Author: Toshiyuki Imamura (RIKEN AICS)

**Overview:**
- Our SYMV routines are optimized by taking into account the symmetry of matrix with atomic operations and auto-tuning.
- Although KBLAS and CUBLAS (atomics mode) also exhibit high performance with atomic operations, they do not guarantee bit-wise reproducible result. Our implementation resolved this issue by implementing mutex with atomic operations.
- Tuning parameters are automatically determined by performing multiple trial executions during the building process. However, the process requires long hours. We decreased the tuning time for trial executions by estimating optimal tuning parameters based on the d-spline [2] fitting function.

**References:**
1. T. Imamura, S. Yamada and M. Machida, "A High Performance SYMV Kernel on a Fermi-core GPU," High Performance Computing for Computational Science - VECPAR 2012, Lecture Note in Computer Science (LNCS) 7851, pp. 59-71, 2013.
2. T. Tanaka, "An Incremental Performance Parameter Estimation Method Applied to Software Automated Performance Tuning for Numerical Libraries," PhD thesis, the University of Electro-Communications, 2007 (in Japanese).




1 Kernel design

```
v(1:n)=0;
y(1:n)*=beta

do j=1,n
  t=0
  do i=1,j-1
    Aij=A(i,j)
    v(i)+=Aij*x(j)
    t+=Aij*x(i)
  enddo
  y(j)+=alpha*t
enddo

do i=1,n
  y(i)+=alpha*A(i,i)*x(i)
enddo
y(1:n)+=alpha*v(1:n)
```

Sequential SYMV algorithm

```
kernel symv preprocess
  j := get threadIID().
  if j < n then
    v(j) := 0, and y(j) *= beta.
  endif
  if j < MAX blkID then
    ticket(j) := MAX blkID.
  endif
  if j = 0 then
    atomicExch( &Master blkID, 0 ).
  endif
endkernel

kernel symv main <Tx, Ty, U, M>
  define j = j+threadIdx.x.
  thID := get localID(), and
  blkID := get blockID().
  d := (U/Ty)*threadIdx.y, i := U*blkID, and
  s := ceil(i−1, Tx).
  Ticket := ticket.
  yreg[0] := ... := yreg[U/Ty−1] := 0.

  for j:=0 to s−1 step Tx
    if j < i−1 then
      areg[k] := A(j, i+k +d),
      yreg[k] += areg[k]*x(j), and
      wreg := ∑kareg[k]*x(i+k) for k ∈ [0, U/Ty).
    endif
    get Ticket( Ticket )
    wreg := sumup wreg through Ty.
    if j < i−1 then
      v(j) += wreg.
    endif
    release Ticket( Ticket ), and Ticket++.
  endfor

  for j:= thID to U do
    shm[thID][j] := shm[j][thID] := A(i+thID, i+j).
  endfor
  synchthreads
  if thID < U then
    yreg[k] := shm[thID][k]*x(i+k) for k ∈ [0, U/Ty).
  endif
  shm[k][thID] := sumup yreg[k]
    through Tx for k ∈ [0, U/Ty).
  if thID < U then
    y(i+thID) += alpha*shm[thID][thID].
  endif
  check the passage of all thread-blocks
  j := get threadIID().
  if j < n then
    y(j) += alpha*v(j).
  endif
endkernel
```

2 CUDA implementation

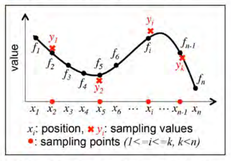
```
function get blockID
  if isMasterThread() then
    c := atomicInc( &Master blkID ).
  endif
  broadcast c of MasterThread.
  return MAX blkID−c.
endfunction

function get threadID
  return threadIdx.x+blockIdx.x*blockDim.x.
endfunction

function get localID
  return threadIdx.x+threadIdx.y*blockDim.x.
endfunction

procedure get Ticket( int *Ticket )
  if isMasterThread() then
    while (TRUE)
      c := atomicCAS( Ticket, blkID, −1 ).
      if c = blkID break
    endwhile
  endif
  syncthreads
endprocedure

procedure relase Ticket( int *Ticket )
  syncthreads
  if isMasterThread() then
    atomicExch( Ticket, blkID−1 ).
  endif
endprocedure
```

Functions for mutex

### 3 Tuning parameters

| Search range of tuning parameters | | | |
|---|---|---|---|
| Tx (x-dim threads per TB) | {32,64,…,Tx-max} | | |
| Ty (y-dim threads per TB) | {1,2,…,8} | | |
| U (panel width per TB) | U/Ty ∈ {3,4,…,32} | | |
| M (stream order) | {1,2,…,10} | | |
| **Restrictions** | | | |
| 96 (3×WarpSize) ≦ Tx × Ty ≦ Tx-max | | | |
| Tx-max := {288 (DSYMV), 320 (SSYMV)} | | | |
| U ≦ Tx | | | |
| **Architecture-dependent** | **Fermi** | **Kepler** | **Maxwell** |
| USE_VOLATILE (use of volatile) | Yes | Yes | Yes |
| USE_TEXTURE (use of texture memory) | Yes | No | No |
| USE_RESTRICT (use of read only caceh with "const TYPE __restrict__") | No | CC3.0=Yes CC3.5=No | No |
| USE_LDG (use of read only cache with "ldg()") | No | CC3.0=No CC3.5=Yes | Yes |

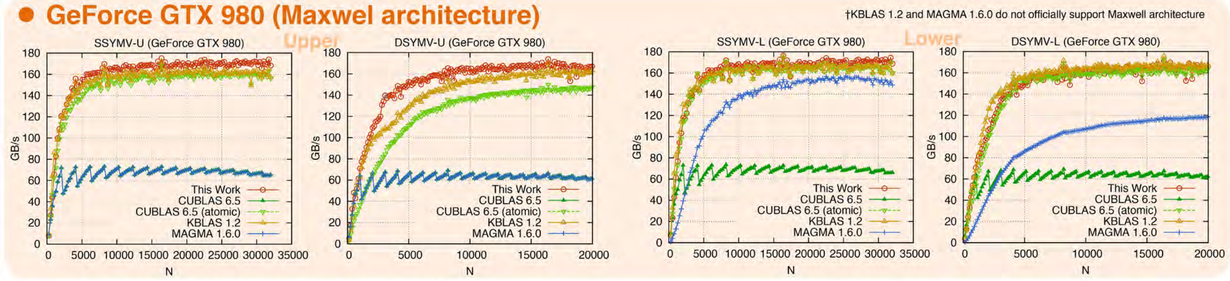
### 4 "d-spline" based parameter estimation



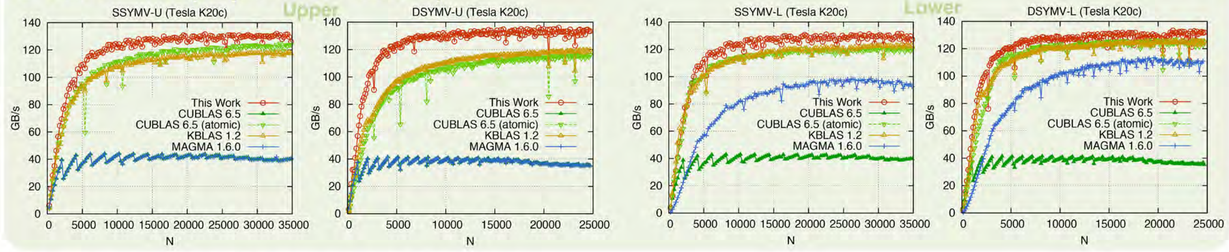

- "d-spline" is a spline approximation for discrete valued data. It generates fitting results $f=(f_1,f_2,\ldots,f_n)^T$ to minimize $||y-Ef||^2+\alpha^2||Df||^2$ with sampling data $y=(y_1,y_2,\ldots,y_k)^T$ and their position data $x=(x_1,x_2,\ldots,x_k)^T$. $E$ and $D$ are coefficient matrices. $\alpha$ is a parameter to determine smoothness.
- Tuning parameters for all matrix sizes are determined by d-spline with optimal parameters at some sampling points (matrix sizes).

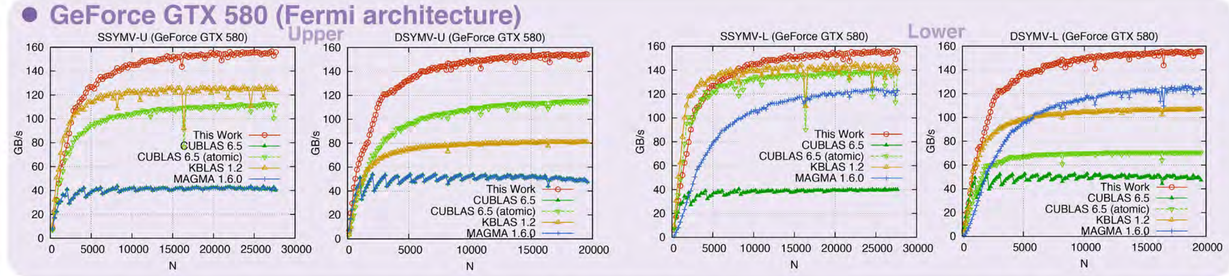
†KBLAS 1.2 and MAGMA 1.6.0 do not officially support Maxwell architecture

### ● GeForce GTX 980 (Maxwel architecture)




### ● Tesla K20c (Kepler architecture)




### ● GeForce GTX 580 (Fermi architecture)