Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200030

High Performance Eigenvalue Solver for Hubbard Model: Tuning Strategies for LOBPCG Method on CUDA GPU

Susumu YAMADA [a,1], Masahiko MACHIDA [a] and Toshiyuki IMAMURA [b]
[a] *Center for Computational Science & e-Systems, Japan Atomic Energy Agency*
[b] *RIKEN Center for Computational Science*

Abstract. The exact diagonalization is the most accurate approach for solving the Hubbard model. The approach calculates the ground state of the Hamiltonian derived exactly from the model. Since the Hamiltonian is a large sparse symmetric matrix, we usually utilize an iteration method. It has been reported that the LOBPCG method is one of the most effectual solvers for this problem. Since most operations of the method are linear operations, the method can be executed on CUDA GPU, which is one of the mainstream processors, by using cuBLAS and cuSPARSE libraries straightforwardly. However, since the routines are executed one after the other, cached data can not be reused among other routines. In this research, we tune the routines by fusing some of their loop operations in order to reuse cached data. Moreover, we propose the tuning strategies for the Hamiltonian-vector multiplication with shared memory system in consideration of the character of the Hamiltonian. The numerical test on NVIDIA Tesla P100 shows that the tuned LOBPCG code is about 1.5 times faster than the code with cuBLAS and cuSPARSE routines.

Keywords. LOBPCG method, CUDA GPU, CUDA Fortran, cuSPARSE, cuBLAS, Hubbard model, quantum lattice systems

1. Introduction

The Hubbard model[1][2] has attracted a tremendous number of physicists since the model exhibits a lot of interesting phenomenon such as High-T_c superconductivity. When we solve the ground state (the smallest eigenvalue and the corresponding eigenvector) of the Hamiltonian derived from the model, we can understand its properties. Therefore, a lot of computational methods for solving this problem have been proposed. The most accurate one is the exact diagonalization which directly solves the ground state of the Hamiltonian derived exactly from the model. Since the Hamiltonian is a huge sparse symmetric matrix, we usually solve the eigenvalue problem with an iteration method, such as the Lanczos method[3], the LOBPCG method[4][5], and so on.

[1] Corresponding Author: Japan Atomic Energy Agency, 178-4 Wakashiba, Kashiwa, Chiba, 277-0871, Japan; E-mail: yamada.susumu@jaea.go.jp.

The graphics processing unit (GPU), which is one of the mainstream processors, achieves an excellent performance with regular data access pattern. Since most of operations of the LOBPCG method are linear ones, the method can be executed on CUDA GPU by using cuBLAS routines[6]. Moreover, since the Hamiltonian can be decomposed into three matrices, whose non-zero elements are arranged regularly, by considering its physical property[7][8], we have proposed the code for the Hamiltonian-vector multiplication using the non-zero patterns in the three matrices. We reported in [8] that the code was faster than cuSPARSE routines[9] on CUDA 4.0. However, cuSPARSE routines have been tuned, and then, nowadays they are faster than our proposed codes (see Table 1).

In this research, we focus on the shared memory whose access speed is much faster than the local global memory's. And then, we propose new strategy to store the matrix data in the shared memory when performing Hamiltonian-vector multiplication. Moreover, we fuse some linear operations, which can be calculated using cuBLAS routines, to improve the cache performance.

The rest of the paper of structured as follows. In Section 2, we briefly introduce the algorithm of the Hamiltonian-vector multiplication and its conventional calculation strategy. And we propose the tuning strategies for the multiplication using the shared memory system on CUDA GPU. Section 3 presents the tuning strategies for other operations of the LOBPCG method. Section 4 shows the result of numerical test on NVIDIA Tesla P100. A summary and conclusion are given in Section 5.

2. Tuning strategy for Hamiltonian-vector multiplication

2.1. Hamiltonian-vector multiplication

The Hamiltonian of the Hubbard model is given as

$$H = -t \sum_{i,j,\sigma} c^{\dagger}_{j\sigma} c_{i\sigma} + \sum_{i} U_i n_{i\uparrow} n_{i\downarrow}, \tag{1}$$

where t is the hopping parameter from a site to another one and U_i is the repulsive energy for one-site double occupation of two fermion the i-th site. Quantities $c_{i,\sigma}$, $c^{\dagger}_{i,\sigma}$ and $n_{i,\sigma}$ are the annihilation, the creation, and the number operator of a fermion with pseudo-spin σ on the i-th site, respectively. When we solve the ground state of the Hamiltonian, we can understand the property of the model.

Here, the Hamiltonian is a huge sparse symmetric matrix, therefore, we usually utilize an iteration method, such as the Lanczos method, the LOBPCG method, and so on. Since the most time-consuming operation of the solvers is the matrix-vector multiplication, it is crucial to tune the Hamiltonian-vector multiplication. Therefore, the storage formats of the Hamiltonian and the vector are crucial for high performance computing. Here, the multiplication Hv can be split as

$$Hv = Dv + (I_{\downarrow} \otimes A_{\uparrow})v + (A_{\downarrow} \otimes I_{\uparrow})v, \tag{2}$$

where $I_{\uparrow(\downarrow)}$, $A_{\uparrow(\downarrow)}$, and D are the identity matrix, a sparse symmetric matrix derived from the hopping of an up-spin (a down-spin), and a diagonal matrix from the repulsive energy, respectively[10]. The multiplication (2) can be represented as

```
      i=(blockIdx%x-1)*blockDim%x+threadIdx%x
      j=(blockIdx%y-1)*blockDim%y+threadIdx%y
      ix=threadIdx%x; iy=threadIdx%y
      V_s(ix,iy)=V_new(i,j)
      call syncthreads()
!!
      i=(blockIdx%y-1)*blockDim%x+threadIdx%x
      j=(blockIdx%x-1)*blockDim%y+threadIdx%y
      ix=threadIdx%x; iy=threadIdx%y
      do k=iru(i), iru(i+1)-1
        V_s(iy,ix)=V_s(iy,ix)+Au(k)*V_r(j,icu(k))
      enddo
      call syncthreads()
!!
      i=(blockIdx%x-1)*blockDim%x+threadIdx%x
      j=(blockIdx%y-1)*blockDim%y+threadIdx%y
      ix=threadIdx%x; iy=threadIdx%y
      V_new(i,j)=V_s(ix,iy)
```

(a) $A_{\uparrow}V$

```
      i=(blockIdx%y-1)*blockDim%x+threadIdx%x
      j=(blockIdx%x-1)*blockDim%y+threadIdx%y
      do k=ird(j), ird(j+1)-1
        V_new(i,j)=V_new(i,j)+V(i,icd(k))*Ad(k)
      enddo
```

(b) $VA_{\downarrow}^{T}$

Figure 1. Schematic CUDA Fortran code of $A_{\uparrow}V$ and $VA_{\downarrow}^{T}$. The data of V and `V_r` are stored in column-major order and row-major one, respectively. Here, `V_s` is the shared memory array. The non-zero elements of the matrices $A_{\uparrow}$ and $A_{\downarrow}$ are stored in the CRS format, that is, the vectors `A*`, `ic*`, and `ir*` store the the values of non-zero elements, the column indexes of the elements, and the indexes where each row starts.

$$V_{i,j}^{new} = \bar{D}_{i,j}V_{i,j} + \sum_{k=1}^{m} A_{\uparrow i,k}V_{k,j} + \sum_{k=1}^{n} V_{i,k}A_{\downarrow k,j} \tag{3}$$

where the subscript i, j of the matrix is represented as the (i, j)-th element and V and $\bar{D}$ are constructed from the elements of the vector v and the diagonal elements of the matrix D in consideration of the physical property of the Hubbard model, respectively[10].

2.2. Conventional multiplication strategy

When the data of the matrix V are stored in column-major order, we can execute the multiplication (2) with contiguous memory access on CUDA Fortran by the following:

1. $V_{new} \leftarrow$ elementwise product of $\bar{D}$ and V,
2. $V_r \leftarrow V$ (row-major $\leftarrow$ column-major (transpose)),
3. $V_{new} \leftarrow V_{new} + A_{\uparrow}V_r$, (see Fig.1 (a))
4. $V_{new} \leftarrow V_{new} + VA_{\downarrow}^{T}$. (see Fig.1 (b))

On the other hand, we can execute the multiplication (2) with cuSPARSE routines as follows:

1. $V_1 \leftarrow$ elementwise product $\bar{D}$ and V

```
      real(8), shared :: au_s(ndim)
      integer, shared :: icu_s(ndim)
!
      i =(blockIdx%x-1)*blockDim%x + threadIdx%y
      i0=(blockIdx%x-1)*blockDim%x + 1
      k =iru(i)-1
      k0=k-iru(i0)
      k1=iru(i+1)-iru(i)
      do l=0,k1-1,blockDim%x
        if (threadIdx%x+l.le.k1) then
          au_s(k0+l+threadIdx%x)=Au(k+l+threadIdx%x)
          icu_s(k0+l+threadIdx%x)=icu(k+l+threadIdx%x)
        endif
      enddo
!
      call syncthreads()
```

Figure 2. Schematic CUDA Fortran code for storing the data of the matrix $A_\uparrow$ the shared memory. Since we store the matrix using the CRS format, therefore, the target vectors are `Au` and `icu`, which store the values of non-zero elements and the column indexes of the elements. In this code, the built-in variable `blockDim%y` should be equal to `blockDim%x`. Moreover, we set the value `blockDim%x` so that the coalescing access can be realized for not only this operation but also the matrix-vector multiplication $A_\uparrow V$.

2. $V_1 \leftarrow V_1 + A_\uparrow V$ (using "cusparseDcsrmm"),
3. $V_2 \leftarrow A_\downarrow V^T$ (using "cusparseDcsrmm2"),
4. $V_{new} \leftarrow V_1 + (V_2)^T$ (transpose and addition).

It was reported in [8] that the former algorithm (our algorithm) was faster than the latter (cuSPARSE) on CUDA 4.0. However, the cuSPARSE routines have been tuned, consequently, they are nowadays faster than our conventional method (see Table 1).

2.3. Tuning strategies by considering memory access

When the codes shown in Fig. 1 are executed on GPU, all threads in a block requires the same data of the matrices $A_\uparrow$ and $A_\downarrow$. In the conventional strategy, since the data are stored in the global memory, each thread has to access them individually. On the other hand, GPU has the shared memory system which can be accessed by all threads even faster than the global one. And, the target data can be stored in the shared memory just by accessing the data stored in the global memory by any one thread, not all threads. Therefore, it is expected that the performance improves by storing data of the matrices on the shared memory. However, since the size of the shared memory on GPU is very small, all data can not be stored. Then, when executing a thread block, we store only data required for the calculation in the shared memory (see Fig. 2).

Moreover, we fuse a do-loop of the elementwise product of $\bar{D}$ and V with that of $VA_\downarrow^T$ to improve the cache performance. Table 1 shows the elapsed time for the multiplication on GPU system in Japan Atomic Energy Agency (see Table 2). The result indicates that the tuned code is about 1.3 times faster than cuSPARSE routines on a recent GPU system.

Table 1. Elapsed time of Hamiltonian-vector multiplication on GPU system in Japan Atomic Energy Agency (see Table 2). The target Hamiltonian is derived from 2-dimensional (4×4-site) Hubbard model with 7 up-spins and 7 down-ones. In the tuned code, the elementwise product of $\bar{D}$ and V is fused with the multiplication $VA_{\downarrow}^{T}$, therefore the table indicates the sum of their elapsed times.

	Elapsed time (msec)		
	Conventional	cuSPARSE	Tuned
Elementwise product of $\bar{D}$ and V	5.66	5.66	13.44
$VA_{\downarrow}^{T}$	28.85	9.06	
$A_{\uparrow}V$	33.70	19.07	13.11
Transpose (and addition)	4.50	7.93	4.50
Total	72.71	41.72	31.05

Dimension of $A_{\uparrow}(A_{\downarrow})$: 11440
Number of non-zero elements of $A_{\uparrow}(A_{\downarrow})$: 144144
Dimension of Hamiltonian: 130873600

Table 2. Details of GPU system in Japan Atomic Energy Agency.

Processor	Intel Xeon E5-2680 v4
GPU	NVIDIA Tesla P100
Fortran Compiler	pgfortran 17.1-0
CUDA Version	8.0
Compile option	`-O3 -Mcuda=6.0 -lcublas -lcusparse -llapack -lblas`

3. Tuning strategies for other operations of LOBPCG method

In this section, we propose the tuning strategies for operations other than the matrix-vector multiplication of the LOBPCG method shown in Fig. 3.

First, we focus on the two 3×3-dimensional symmetric matrices (S_A and S_b in Fig. 3). In order to construct them, we have to calculate ten inner product operations using six vectors[2]. These operations can be realized by executing cuBLAS routine `cublasddot` ten times. However, since these operations are executed one after the other, we can not reuse cached data which were used in other operations. Therefore, we fuse ten operations and store the data in the shared memory to improve cache performance. The most important operation in an inner product on GPU is sum-reduction. When the shared memory system is used appropriately, the operation can be executed efficiently on GPU[11]. However, the shared memory is too small to store all data. Therefore, we decompose the vectors so that we can store the decomposed data in the shared memory, and we calculate partial sums of ten inner products using the decomposed vectors (see Fig. 4). After that, we calculate the global sums of partial sums using the shared memory system. The elapsed time using cuBLAS and our proposed code for the inner product operations on NVIDIA Tesla P100 for the same problem in Table 1 are 33.29 msec and 25.53 msec, respectively.

Moreover, Fig. 5 shows the operations to update the vectors x, p, X, P, and w in the LOBPCG method. Each operation can be realized using a cuBLAS routine. On the other

[2]Twelve inner products are required to construct the two matrices. However, since two vectors w and p are normalized, there is no need to calculate the two inner products (w,w) and (p,p).

$\boldsymbol{x}_0$:= an initial guess, $\boldsymbol{p}_0 := 0$
$\boldsymbol{x}_0 := \boldsymbol{x}_0/\|\boldsymbol{x}_0\|$, $X_0 := A\boldsymbol{x}_0$, $P_0 := 0$, $\mu_{-1} := (\boldsymbol{x}_0, X_0)$, $\boldsymbol{w}_0 := X_0 - \mu_{-1}\boldsymbol{x}_0$
do k=0, ... until convergence
$W_k := A\boldsymbol{w}_k$
$S_A := \{\boldsymbol{w}_k, \boldsymbol{x}_k, \boldsymbol{p}_k\}^T\{W_k, X_k, P_k\}$
$S_B := \{\boldsymbol{w}_k, \boldsymbol{x}_k, \boldsymbol{p}_k\}^T\{\boldsymbol{w}_k, \boldsymbol{x}_k, \boldsymbol{p}_k\}$
Solve the smallest eigenvalue μ and the corresponding vector $\boldsymbol{v}$,
$S_A\boldsymbol{v} = \mu S_B\boldsymbol{v}$, $\boldsymbol{v} = (\alpha, \beta, \gamma)^T$.
$\mu_k := (\mu + (\boldsymbol{x}_k, X_k))/2$
$\boldsymbol{x}_{k+1} := \alpha\boldsymbol{w}_k + \beta\boldsymbol{x}_k + \gamma\boldsymbol{p}_k$, $\boldsymbol{x}_{k+1} := \boldsymbol{x}_{k+1}/\|\boldsymbol{x}_{k+1}\|$, $\boldsymbol{p}_{k+1} := \alpha\boldsymbol{w}_k + \gamma\boldsymbol{p}_k$, $\boldsymbol{p}_{k+1} := \boldsymbol{p}_{k+1}/\|\boldsymbol{p}_{k+1}\|$
$X_{k+1} := \alpha W_k + \beta X_k + \gamma P_k$, $X_{k+1} := X_{k+1}/\|\boldsymbol{x}_{k+1}\|$, $P_{k+1} := \alpha W_k + \gamma P_k$, $P_{k+1} := P_{k+1}/\|\boldsymbol{p}_{k+1}\|$
$\boldsymbol{w}_{k+1} := X_{k+1} - \mu_k\boldsymbol{x}_{k+1}$, $\boldsymbol{w}_{k+1} := \boldsymbol{w}_{k+1}/\|\boldsymbol{w}_{k+1}\|$
enddo

Figure 3. Algorithms of LOBPCG method for the matrix A. Here, X, P, and W mean the vectors multiplied by the matrix A, that is, Ax, Ap, and Aw, respectively.

```
real(8), shared :: V_s(128,3)
!!
i=2*(blockIdx%x-1)*blockDim%x+threadIdx%x
ith=threadIdx%x
ibl=blockIdx%x
!!
V_s(ith,1)=x(i)*x(i)+x(i+128)*x(i+128)
V_s(ith,2)=x(i)*w(i)+x(i+128)*w(i+128)
V_s(ith,3)=w(i)*w(i)+w(i+128)*w(i+128)
call syncthreads()
!!
do k=1,3
 do j=6,0,-1
  n=2**j
  if (ith.le.n) V_s(ith,k)=V_s(ith,k)+V_s(ith+n,k)
  call syncthreads()
 enddo
enddo
v(ibl,1)=V_s(1,1);v(ibl,2)=V_s(1,2);v(ibl,3)=V_s(1,3);
```

Figure 4. Schematic CUDA Fortran code for calculating partial sums ($v(*,1)$, $v(*,2)$, and $v(*,3)$) of three inner products (x,x), (x,w), and (w,w) from two vectors x and w using the shared memory array `V_s`. Here, the code is executed using partitioned vectors whose length is 256, that is, the built-in variable `blockDim%x` is set as 128. After this calculation, the global sum is calculated using the partial ones. In actual execution of the LOBPCG method, we use the code extended with a similar strategy for calculating ten inner products using six vectors.

hand, when we consider the data dependencies, we can replace all loops of the operations with one loop as shown in Fig. 6. In addition, it is necessary to normalize vectors p, P, w and W using $p_{norm}(= ||p||)$ and $w_{norm}(= ||w||)$ before the inner product operations mentioned above, because the more the iteration converges, the smaller the norms of p and w become[3]. The normalization can be executed using a routine `cublasdscal`, but their loops can be also combined with the fused loop of the inner product operations

[3]It is also necessary to normalize the vectors x and X. However, the norm of the vector x is theoretically 1, therefore, these normalization are executed after calculating S_A and S_B. And we correct the corresponding elements of S_A and S_B in accordance with these normalization.

```
p ← γp (cublasdscal);      P ← γP (cublasdscal)
p ← p + αw (cublasdaxpy); P ← P + αW (cublasdaxpy)
x ← βx (cublasdscal);      X ← βX (cublasdscal)
x ← x + p (cublasdaxpy);   X ← X + P (cublasdaxpy)
w ← X (cublasdcopy);       w ← w − λx (cublasdaxpy)
p_norm ← ||p|| (cublasdnrm2); w_norm ← ||w|| (cublasdnrm2)
```

Figure 5. Operations to update vectors and calculate norm of vectors in LOBPCG method. All loops of the operations, which can be realized using cuBLAS routines, can be fused into one loop by considering data dependencies.

```
real(8), shared :: V_s(128,2)
!!
i=2*(blockIdx%x-1)*blockDim%x+threadIdx%x
ith=threadIdx%x
ibl=blockIdx%x
!!
p(i)=γ*p(i)+α*w(i); p(i+128)=γ*p(i+128)+α*w(i+128);
P(i)=γ*P(i)+α*W(i); P(i+128)=γ*P(i+128)+α*W(i+128);
x(i)=β*x(i)+p(i); x(i+128)=β*x(i+128)+p(i+128);
X(i)=β*X(i)+P(i); X(i+128)=β*X(i+128)+P(i+128);
w(i)=X(i)-λx(i); w(i+128)=X(i+128)-λx(i+128);
V_s(ith,1)=p(i)*p(i)+p(i+128)*p(i+128)
V_s(ith,2)=w(i)*w(i)+w(i+128)*w(i+128)
call syncthreads()
!!
do k=1,2
 do j=6,0,-1
  n=2**j
  if (ith.le.n) V_s(ith,k)=V_s(ith,k)+V_s(ith+n,k)
  call syncthreads()
 enddo
enddo
v(ibl,1)=V_s(1,1);v(ibl,2)=V_s(1,2);
```

Figure 6. Schematic CUDA Fortran code for fusing all operations shown in Fig. 5. We calculate the partial sums ($v(*,*)$ and $v(*,2)$) of two inner products (p,p), and (w,w) using the shared memory array `V_s`. Here, we set the built-in variable `blockDim%x` as 128. After this calculation, we calculate the two global sums using the partial ones, and then, the square root of them.

mentioned above. It is expected that these loop fusion operations improve the cache performance and realize speedup.

4. Numerical test

In this section, we examine the performance of the LOBPCG method for the Hubbard model. We solve the ground state (the minimal eigenvalue and the corresponding eigenvector) of the eigenvalue problem derived from a 2-dimensional (4×4-site) model with 7 up-spins and 7 down-ones using the LOBPCG method on the GPU system, whose details are shown in Table 2. Table 3 shows the number of the iterations, the elapsed time, and the performance. The result indicates that the conventional method is slower than that using cuSPARSE routines. Moreover, it is confirmed that the performance improves by par-

Table 3. Elapsed time and performance for exact diagonalization on NVIDIA Tesla P100. The target Hamiltonian is the same as Table 1. Here, the Hamiltonian-vector multiplication is executed using the conventional code, cuSPARSE, and the tuned one. Moreover, other operations are execute using cuBLAS and the tuned code.

Multiplication	Conventional	cuSPARSE	Tuned	Tuned
Others	cuBLAS	cuBLAS	cuBLAS	Tuned
Number of iterations	164	164	164	164
Elapsed time (sec)	30.24	24.90	23.12	16.57
Performance (GFLOPS)	69.0	83.8	90.3	125.9

tially storing the matrix elements on the shared memory and its performance is superior to cuSPARSE's one. And then, when other operations are also tuned, the code achieves speedup of 1.5 times faster than the code using cuBLAS and cuSPARSE routines.

5. Conclusions

We have proposed the tuning strategy using the shared memory for Hamiltonian-vector multiplication on the exact diagonalization method for the Hubbard model for the CUDA GPU. Since the size of the shared memory is very small, we store only the matrix data required by the executing block in the shared memory. The numerical result shows that the matrix-vector multiplication using the strategy is about 1.3 times faster than that using the cuSPARSE routines. Moreover, we also tuned other linear operations of the LOBPCG method in order to reuse more cached data. Therefore, we fused some loops into one loop by considering data dependencies. At a result, it is confirmed that the LOBPCG method using the proposed tuning strategies is about 1.5 times faster than that using cuBLAS and cuSPARSE routines.

In future work, in order to examine the physical property of a large Hubbard model, we aim to realize the high performance exact diagonalization on multi-GPU systems. For this aim, we plan to investigate the tuning strategies in consideration of the effects of the data communication between GPUs and/or CPUs.

Acknowledgment

This research was partially supported by JSPS KAKENHI Grant Number 18K11345. Computations in this work were performed on GPU system in Japan Atomic Energy Agency.

References

[1] M. Rasetti, ed., The Hubbard Model: Recent Results, World Scientific, Singapore (1991)
[2] A. Montorsi, ed., The Hubbard Model, World Scientific, Singapore (1992)

[3] J.K. Cullum and R.A. Willoughby, Lanczos Algorithms for Large Symmetric Eigenvalue Computations, Vol.1: Theory, Philadelphia: SIAM, (2002)
[4] A. V. Knyazev, Preconditioned eigensolvers - An oxymoron?, Electronic Transactions on Numerical analysis 7, 104-123 (1998)
[5] A. V. Knyazev, Toward the optimal eigensolver: Locally optimal block preconditioned conjugate gradient method, SIAM J. Sci. Comput.,23, 517-541 (2001)
[6] cuBLAS. URL: https://developer.nvidia.com/cublas.
[7] T. Siro and A. Harju, Exact diagonalization of the Hubbard model on graphics processing units,Comp. Phy. Comm., 183, 1884-1889 (2012).
[8] S. Yamada, T. Imamura, and M. Machida, High Performance Eigenvalue Solver in Exact-diagonalization Method for Hubbard Model on CUDA GPU, Parallel Computing: On the road to Exascale (G. R. Joubert, et. al, Ed.), IOS Press, 361-369 (2016).
[9] cuSPARSE. URL: https://developer.nvidia.com/cusparse.
[10] S. Yamada, T. Imamura, and M. Machida, 16.447 TFlops and 159-Billion-dimensional Exact-diagonalization for Trapped Fermion-Hubbard Model on the Earth Simulator, *Proc. of SC05* (2005).
[11] M. Harris, Optimizing parallel reduction in CUDA, NVIDIA Developer Technology (2007).