Using GPU Accelerators for improving Performance and Scalability in Material Physics Simulations

M. Hruszowiec\textsuperscript{a}, P. Potasz\textsuperscript{b}, A. Szymańska-Kwieciń\textsuperscript{a}, M. Uchorński\textsuperscript{a*}

\textsuperscript{a} Wroclaw Centre of Networking and Supercomputing (WCSS), Wroclaw University of Science and Technology
\textsuperscript{b} Department of Theoretical Physics, Wroclaw University of Science and Technology

Abstract

This work will be focused on parallel simulation of electron-electron interactions in materials with non-trivial topological order (i.e. Chern insulators). A problem of electron-electron interaction systems can be solved by diagonalizing a many-body Hamiltonian matrix in a basis of configurations of electrons distributed among possible single particle energy levels - a configuration interaction method. The number of possible configurations exponentially increases with a number of electrons and energy levels; 6 electrons occupying 24 energy levels corresponds to the dimension of Hilbert space about $10^8$; for 12 electrons it gives $10^9$ configurations. Solving such a problem requires effective computational methods and highly efficient optimization of the source code. The project will focus on many-body effects related to strongly interacting electrons on flat bands with non-trivial topology. Such systems are expected to be useful in study and understanding of new topological phases of matter, and in a further future can be used to design novel nanomaterials. GPU accelerators will be used for improving performance and scalability in parallel simulation of electron-electron interaction in materials with a non-trivial topological order.

Introduction

The main goal of this work, undertaken within PRACE-4IP project, was to prepare GPU implementation for improving performance and scalability in material physics simulations. During technical work most promising routines of Fortran/OpenMP code were identified and ported to the GPU accelerators using CUDA. Using GPU accelerators in simulations of electron-electron interactions can improve its performance and scalability. This work can support scientific communities focused on research in condensed matter physics. Project results may be useful/related to the CoEs focused on identify novel materials such as MaX or NoMaD.

In this work a problem of electron-electron interaction systems was solved by diagonalizing a many-body Hamiltonian matrix in a basis of configurations of electrons distributed among possible single particle energy levels - a configuration interaction method. In Modified Lanczos Method for Atom Annihilation Creation

* Corresponding author. E-mail address: mariusz.uchronski@pwr.edu.pl
(MLM4AAC) application configuration space of electrons distributed among possible single particle energy levels is generated. Such configuration space is divided into subspaces. Next for each configuration subspace a configuration interaction method is executed. The main element of this method is `twooper` function in which nonzero matrix elements are calculated and then matrix is multiplied by an initial vector. The matrix is in a block diagonal form with row index corresponding to distributions of particles on states (configuration). In general a new configuration is generated by annihilation “ones” within given configuration and creates new “ones”. Index `p` for a new configuration is calculated using a hash table. New configuration is checked if it is in the same subspace. Finally, a matrix is multiplied by a vector and results are stored in output vector.

**Technology used**

Fortran 90 is widely used version of the world's oldest scientific programming language. It was accepted as an international standard in June, 1990 [1]. Fortran dates from 1954, when the first FORmula TRANslation system was developed at IBM by a team led by John Backus. It is used in many computationally intensive areas such as finite element analysis, computational fluid dynamics and computational physics. It is a popular language for high-performance computing and is used in programs that benchmark and rank the world's fastest supercomputers.

OpenMP is a shared-memory application programming interface (API) which enables programmers to use benefits of parallel programming in an easy way [2]. The first specification of OpenMP was introduced in October 1997 by Architecture Review Board for Fortran, a year later the first version for C/C++ was introduced. It is not a new programming language, but a notation that can be added to a sequential program written in Fortran, C or C++ in order to work in the shared-memory model. OpenMP introduces a set of special directives i.e. `#pragma` in C/C++ and `$OMP` in Fortran. The directives enable a programmer to transform sequential code into multithreaded one in a quite easy way. It is strongly focused on parallelizing loops, when each iteration is independent from all others, then every iteration can be done by separate thread. The latest version of OpenMP is 4.5 and was announced in November 2015 [3].

CUDA was introduced in November 2006 by NVIDIA, as a general purpose parallel computing platform and programming model based on using GPUs [4]. CUDA comes with a dedicated software environment that enables developers to use C/C++ high-level programming language. CUDA has introduced extensions that allow creating GPU dedicated functions called "kernels" along with type specifiers such as `__device__`, `__global__`, `__local__`, `__shared__`. These specifiers are used to indicate if a given variable or function can be used by CPU, GPU or both. Kernels are executed directly on GPUs. This at its core has three abstractions: a hierarchy of thread groups, shared memories and barrier synchronization. The idea behind using CUDA is to divide a given problem to a subset of problems that can be solved independently in parallel by blocks of threads. Each sub-problem can be also divided into finer pieces that can be solved in cooperation in parallel by all threads in the block. The latest version of CUDA is 8.0 and was announced on September 28, 2016.

**Hardware used**

For development and testing we have used the Nova system with 3 fat nodes with four sixteen core AMD Opteron 6274 processors with 256 GB of memory and two NVIDIA Tesla M2075 (448 cores, 6 GB of memory) each. We have also used the BEM cluster with 720 computing nodes with 24-core Intel Xeon E5-2670 v3 2.3 GHz each, Haswell and 192 nodes with 28-cores Intel Xeon E5-2697 v3 2.6 GHz, Haswell each. Both systems are located at Wroclaw Centre for Networking and Supercomputing (WCSS).

**Implementation and testing**

The work undertaken within this task started from basic code improvements such as: porting code from Fortran77 to Fortran90, code reorganization and refactoring. Also some effort was spend on improving implementation for generation of configurations of electrons distributed among possible single particle energy levels. The next step was compilation of the MLM4AAC application on Nova using `gfortran` from the GNU compiler suite (v4.9.2). The compilation was done successfully and the application was analysed with `gprof(v2.20)` tool. The medium size problem was taken as an example for performance tests and it was conducted on the Nova. The source code was after basic code improvements. The analysis has shown that function `twooper` is using over 80% of the processor time (Figure 1). It was obvious choice to check if it can be improved, i.e. by implementation on CUDA device.
Figure 1 Output of gprof tool for a single run of the medium problem size (6 particles with 24 states and 5700 maximum number of configurations)

```
Each sample counts as 0.01 seconds.
   % cumulative self  self  total
time  seconds  seconds  calls  ms/call  ms/call  name
 88.33  1708.73  1708.73          
  6.24   1829.39  120.66  1722550782  0.00   0.00  hash_get_
  3.14   1890.22   60.83          
  0.89   1907.44   17.22          
  0.84  1923.69   16.25  271800    0.06   0.06  ortho_
  0.31  1929.72   6.02          
  0.10  1931.62   1.91  218939208  0.00   0.00  sort_two_
  0.05  1932.57   0.95          
  0.04  1933.37   0.80          
  0.02  1933.74   0.37          
  0.02  1934.09   0.35          
  0.01  1934.35   0.26  3624    0.07   0.07  norm_
```

Single GPU implementation

In the initial implementation of the MLM4AAC the \texttt{twooper} function was called within three, or even four, nested loops (Figure 2). As it was noticed each execution of \texttt{twooper} function was independent, so in a natural way it could be parallelized.

```
do ix=1,Nx
   do iy=1,Ny
      ... 
      do i=1, iconf
         call twooper(max_config, N_st, N_part, M, emp, kxy, 
                       Nx, Ny, bin, bit, Nbit, N_GS, N_EX, 
                       EX_GS, el, eval, r, v1, i, dbl_vec)
      end do
   end do
   k=k+1 
   ... 
   do i=1, iconf
      call twooper(max_config, N_st, N_part, M, emp, kxy, 
                   Nx, Ny, bin, bit, Nbit, N_GS, N_EX, 
                   EX_GS, el, eval, r, v2, i, dbl_vec)
   end do
   ... 
end do
```

Figure 2 Skeleton of twooper function calls
The first step was to prepare an implementation of the `twooper` function for CUDA, so we can call it as it is shown in a Figure 3. The most inner loop has been removed. Two additional functions were written in C and CUDA C to provide proper interface for call from the Fortran code.

```c

call cu_twooper(max_iconf, iconf, N_st, N_part, M, emp, kxy, Nx, Ny, bin,&
                Nbit, N_GS, N_EX, GS, el, eval, r, v1, sdbl_vec, sdbl_vec_size, 0)
```

Figure 3 CUDA function call introduced in the Fortran code

As we used `gfortran` compiler the only way to connect C code with Fortran was to link it with each other at linking stage. Two functions were created; the first one is the interface to call function within the Fortran code (Figure 4). The C function `cu_twooper` implements data transfer from host to device memory and execution of a proper kernel function on GPU device and copying the results back to the host memory.

```c
extern "C" void cu_twooper(
    const ftn8 *max_iconf,
    ...
    const ftn8 *tid)
{
    // memory allocation
    checkCuda( cudaAlloc((void**)&Mat_P,
                          +iconf+(N_st)+sizeof(ftn8)) );
    ...
    checkCuda( cudaMalloc((void**)&sdbl_vec_P,
                          +sdbl_vec_size+2*sizeof(ftn8)) );

    // copy data to the device memory
    checkCuda( cudaMemcpy(Mat_P, Mat,
                          +iconf+(N_st)+sizeof(ftn8),
                          cudaMemcpyHostToDevice) );
    ...
    checkCuda( cudaMemcpy(sdbl_vec_P, sdbl_vec,
                          +sdbl_vec_size+2*sizeof(ftn8),
                          cudaMemcpyHostToDevice) );

    unsigned threads = 266;
    // number of threads: max is 65535 in g++
    unsigned gridx = (65535 < +iconf/threads) ? 65535 : +iconf/threads + 1;
    cu_inner_loop<<<gridx, threads>>>(iconf, ...
                                   , sdbl_vec_size);
    cudaThreadSynchronize();

    checkCuda( cudaMemcpy(rr, rr_P, +max_iconf+sizeof(fcomplex8),
                          cudaMemcpyDeviceToHost) );

    // release memory
    checkCuda( cudaFree(Mat_P) );
    ...
    checkCuda( cudaFree(sdbl_vec_P) );
}
```

Figure 4 Interface to call function within the Fortran code

In the CUDA kernel function `cu_inner_loop` (Figure 5) each thread calculates nonzero matrix element and multiply the matrix by an initial vector. The matrix is in a block diagonal form with row index corresponding to distributions of particles on states which is called a configuration. Each configuration is represented by a binary number. Such configuration space is divided into subspaces. Every CUDA thread performs computation for one configuration of particles. In general each thread creates a new configuration by annihilation “ones” within given configuration and creates new “ones”. Index \( p \) for a new configuration is calculated using a hash table. New configuration is checked if it is in the same subspace. Finally, a matrix is multiplied by a vector and results are stored in output vector.
After completing implementation that utilized single GPU, a new implementation was design and programmed, which can divide all computations between GPUs available within a host. In order to achieve this goal, one needs to modify two of the most outer loops in the Fortran code (Figure 2) and combine them into one loop (Figure 6). In the presented listing OpenMP directives are also visible. The idea is to execute function `gpu_sub` for a number of GPUs by given number of threads. In an ideal case when number of threads will be equal to the number of available GPUs, each thread will execute `gpu_sub` exactly on one GPU. In used configuration there was only 2 GPUs, so mod 2 thread executed given function on either first or second GPU.

```c
#ifdef CUDA
!$OMP PARALLEL DO PRIVATE(myid, ix)
#endif
do ix=1,Nx*Ny
    myid = omp_get_thread_num()
    call gpu_sub(kxyid(ix, 1), kxyid(ix, 2),
               N_st, N_part, Nx, Ny, kxy, el,
               N_size_T, mod(myid, cuda_gpu_num))
end do
#ifdef CUDA
!$OMP END PARALLEL DO
#endif
```

Figure 6: The source code executing computations on multi GPU
Computational experiments

A set of computational experiments has been performed on two different architectures using three different problem's sizes (small, medium and large). Both OpenMP and CUDA performance were measured for different number of threads. In an OpenMP experiment the following number of threads was considered: 1, 2, 4, 8, 16, 32 and 64. Upper limit for thread number was set. On the other hand in a CUDA experiment 8, 16, 32, 64, 128 and 256 threads were used. For a reference in every measurement, the time measurement result of 1 OpenMP thread was considered. As it was mentioned above three different size examples were taken into account as test samples. In the first example (small) there were considered only 4 particles and 12 states with maximum number of acceptable configuration 160. In this example a Kx and Ky subspace length were 3 and 4 respectively. In the second example (medium) there were 6 particles with 24 states and 5700 maximum number of configurations with Kx, Ky subspaces length 4 and 6 respectively. In the last test case (large) there were 5 particles with 30 states and 150000 maximum configurations with Kx, Ky subspaces length 5 and 6 respectively. Tests were performed on two different computational nodes located in two different clusters. The first one was NOVA cluster based on AMD Opteron processors, the second was our latest cluster BEM, which utilize Intel Xeon processors. The first experiment setup was based on a node equipped with 64 AMD Opteron 6274 processors with 256GB RAM. It also has two NVIDIA Tesla M2075, each with 6GB of memory. It was located within NOVA cluster. The second experiment setup was based on a node equipped with 48 Intel Xeon E5-2670 v3 processors with 64GB RAM. It was located within BEM cluster.

Performance results

Computation time obtained during test runs was used to calculate speedup values. Table 1 contains performance results for OpenMP code executed on NOVA and BEM systems. For eva34 test case speedup values are relative small and don not change significantly with increasing number of OpenMP threads on both systems. The same behaviour can be observed on NOVA system for eva46 and eva56 test case. On BEM system for eva46 and eva56 test cases speedup values increases significantly with increasing number of OpenMP threads.

<table>
<thead>
<tr>
<th>OpenMP threads</th>
<th>speedup on NOVA (AMD Opteron 6274)</th>
<th>speedup on BEM (Intel Xeon E5-2670 v3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eva34</td>
<td>eva46</td>
</tr>
<tr>
<td>2</td>
<td>1.02</td>
<td>1.56</td>
</tr>
<tr>
<td>4</td>
<td>1.45</td>
<td>2.68</td>
</tr>
<tr>
<td>8</td>
<td>1.66</td>
<td>3.83</td>
</tr>
<tr>
<td>16</td>
<td>1.46</td>
<td>3.44</td>
</tr>
<tr>
<td>32</td>
<td>1.33</td>
<td>3.09</td>
</tr>
<tr>
<td>64</td>
<td>1.03</td>
<td>3.18</td>
</tr>
</tbody>
</table>

Table 1 Performance results for OpenMP implementation

Table 2 contains performance results for CUDA code executed on Tesla M2075 located within NOVA cluster for one and two GPUs. For eva34 test case speedup values are very small and don not change with increasing number of CUDA threads per block in one and two GPU cases. For eva46 and eva56 test cases best values of speedup was obtained for 64 CUDA threads per block. Best value of speedup (eva46) for one GPU is equal 20.76 and for two GPUs is equal 57.87 are respectively 2x and 5x better in comparison with best OpenMP speedup equals 10.13.

<table>
<thead>
<tr>
<th>CUDA threads per block</th>
<th>speedup for one GPU</th>
<th>speedup for two GPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>eva34</td>
<td>eva46</td>
</tr>
<tr>
<td>8</td>
<td>0.16</td>
<td>11.49</td>
</tr>
<tr>
<td>16</td>
<td>0.13</td>
<td>13.92</td>
</tr>
<tr>
<td>32</td>
<td>0.17</td>
<td>16.27</td>
</tr>
<tr>
<td>64</td>
<td>0.17</td>
<td>20.76</td>
</tr>
<tr>
<td>128</td>
<td>0.17</td>
<td>19.50</td>
</tr>
<tr>
<td>256</td>
<td>0.17</td>
<td>20.45</td>
</tr>
</tbody>
</table>

Table 2 Performance results for CUDA implementation
Best values of speedup for OpenMP and CUDA are shown on Figure 7. It can be seen that speedup values for code executed on GPU are significantly larger than for code executed on CPU. Using two GPUs for computations results in speedup increase. For large test case data size copied from host to device memory is about 15MB and about 75kB for copying the results back to the host memory.

Future work

Possible way of future evaluation of GPU acceleration in optimized code could be implementation hybrid MPI+CUDA code which allows execute simulations on GPU clusters. This approach can result in shortage of computation time and give possibility to simulate larger systems. Mentioned future work requires some changes in existing code but level of its complexity is average and code can be exploited on future exascale HPC systems.

Conclusion

In conclusion our work demonstrated the potential of using GPU accelerators for improving performance and scalability in material physics simulations. The main factor in obtaining high performance GPU computing is to identify promising areas of application that allow for massive parallelism. For this purpose gprof tool was used and a configuration interaction method was identified as most promising to port on GPU. Prepared GPU implementation provides significant increase of performance (~ x2 speedup) in comparison with parallel OpenMP base implementation. Nature of problem allows enabling GPU computational potential by enabling code execution on multiple GPUs. This goal has been achieved by using the possibility of independent computations for each subspace of configurations space. Multi GPU approach results in next significant increase of performance - ~ x5 speedup for execution on two GPUs. Proposed approach results in shortage of computation time and gives possibility to simulate larger electron-electron interaction systems in future exascale HPC systems with GPUs.

References


Acknowledgements

This work was financially supported by PRACE project funded in part by the EU’s Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838, and by the Polish Ministry of Science and Higher Education under grant agreement 3563/H2020/2016/2.