CMST 24(4) 249–258 (2018) DOI:10.12921/cmst.2018.0000025

Using GPU Accelerators for Parallel Simulations in Material Physics

Mariusz Uchroński[1*], Paweł Potasz[2], Agnieszka Szymańska-Kwiecień[1], Mariusz Hruszowiec[3]

[1] *Wroclaw Centre of Networking and Supercomputing (WCSS)*
Wroclaw University of Science and Technology

[2] *Department of Theoretical Physics*
Wroclaw University of Science and Technology

[3] *Department of Telecommunications and Teleinformatics*
Wroclaw University of Science and Technology

**E-mail: mariusz.uchronski@pwr.edu.pl*

Received: 13 April 2018; revised: 24 November 2018; accepted: 26 November 2018; published online: 24 December 2018

Abstract: This work is focused on parallel simulation of electron-electron interactions in materials with non-trivial topological order (i.e. Chern insulators). The 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 – the configuration interaction method. The number of possible configurations exponentially increases with the number of electrons and energy levels; 12 electrons occupying 24 energy levels corresponds to the dimension of Hilbert space about 10^6. Solving such a problem requires effective computational methods and highly efficient optimization of the source code. The work is focused 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 further future they can be used to design novel nanomaterials. Heterogeneous architecture based on GPU accelerators and MPI nodes will be used for improving performance and scalability in parallel solving problem of electron-electron interaction systems.

Key words: GPU computing, MPI, OpenMP, material physics

I. INTRODUCTION

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 – the configuration interaction method [1]. The configuration interaction method is a universal method for quantum-mechanical many-body problems. It can be used in quantum chemistry [1], solid state physics [2–6], to quantum dots [7] and other nanostructures [8] in order to determine their electronic [6], magnetic [8] and optical properties [7]. In this particular work, we apply this method to find the many-body ground state of a system of a finite number of electrons occupying a topologically nontrivial energy band, which can lead to appearance of topological order (i.e. Fractional Chern insulators) [2–6].

To solve the problem of the electron-electron interaction system the Modified Lanczos Method for Two Particle Creation Annihilation Problem (MLM42PCAP) was implemented. The prepared GPU implementation improved performance and scalability of solving the considered problem. During technical work most promising routines of Fortran/OpenMP code were identified and ported to the GPU accelerators using CUDA. In many places OpenMP was also used to utilize processors and the final step was creating hybrid MPI+CUDA implementation.

II. PROBLEM OF ELECTRON-ELECTRON INTERACTION SYSTEMS

A fundamental principle of quantum mechanics is quantum superposition. Since the equation describing nonrelativistic quantum objects, the Schroedinger equation, is linear, any superposition of solutions is also its solution. Thus, if a quantum system consists of two subsystems, the total space has to be described by a tensor product of these subspaces, which states that all possible combinations of solutions of subspaces has to be taken into account. In the case of the many-body quantum mechanical problem, when each of n particles can be in m states, the total number of states is m^n (neglecting particle's statistics in this analysis), thus increases exponentially with a system size. If one considers a system of interacting electrons (fermionic statistic), the total number of states is given by binomial coefficient of n over m, which still exponentially increases with the number of particles n. Among many-body methods, the configuration interaction method (also called exact diagonalization) is the most exact method when basis expansion contains also possible particle distributions (called configurations). The problem is that the configuration-interaction method can be applied to systems with a small number of particles due to quick growth of computational cost. Any truncation of the configuration space makes it approximate. An important question is when one can restrict a number of configurations such that desired accuracy of the results is kept. In typical weakly interacting systems, mean-field approximation can be applied, corresponding to inclusion of only one dominant configuration. However, strongly interacting systems require inclusion of all possible configurations. A system of a finite number of electrons occupying topologically nontrivial energy band is an example of such problem.

III. MODIFIED LANCZOS METHOD FOR TWO PARTICLE CREATION ANNIHILATION PROBLEM

In the Modified Lanczos Method for Two Particle Creation Annihilation Problem (MLM42PCAP) application system with strongly interacting electrons on flat bands with non-trivial topology is solved. First, the configuration space of electrons distributed among possible single particle energy levels is generated. Such configuration space is divided into subspaces determined by conserved physical quantities, in this case into momentum subspaces. Next, for each configuration subspace a configuration interaction method is executed. The main element of this method is a `twooper` function describing two-body interaction between particles. 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 of two particles within a given configuration and creating two new particles on new positions which correspond to a new configuration. Nonzero matrix elements between different configurations are calculated and then the matrix is multiplied by an initial vector.

IV. IMPLEMENTATION DETAILS

IV. 1. Technology used

Fortran 90 is a widely used version of the world's oldest scientific programming language. It was accepted as an international standard in June, 1990 [9]. 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.

OpenMP is a shared-memory application programming interface (API) which enables programmers to use benefits of parallel programming in an easy way [10]. The first specification of OpenMP was introduced in October 1997 for Fortran, and the first version for C/C++ was introduced a year later. 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. The directives enable a programmer to transform a sequential code into a multithreaded one. The latest version of OpenMP is 4.5 and was announced in November 2015 [11].

```
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                             twooper_
  6.24   1829.39   120.66 1722550782    0.00     0.00  hash_get_
  3.14   1890.22    60.83                             frame_dummy
  0.89   1907.44    17.22                             MAIN__
  0.84   1923.69    16.25   271800       0.06     0.06  ortho_
  0.31   1929.72     6.02                             dlasr_
  0.10   1931.62     1.91 218939208     0.00     0.00  sort_two_
```

Fig. 1. Output of the gprof tool for a single run of the medium problem size (6 particles with 24 states and 5700 maximum number of configurations)

CUDA was introduced in November 2006 by NVIDIA, as a general purpose parallel computing platform and programming model based on using GPUs [12]. 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". Kernels are executed directly on GPUs. The idea behind using CUDA is to divide a given problem to a subset of problems that can be solved independently in parallel by threads organized in blocks. In this work CUDA 8.0 was used.

The Message-Passing Interface (MPI) is a message-passing library interface specification [13]. It is based on a message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through communication. MPI is just a specification, not implementation. In this work OpenMPI (2.0.1) [14] implementation was used. Programs that use MPI implementation must be compiled with a dedicated compiler, e.g. mpif90 and then run with a special command, i.e. mpirun.

IV. 2. Hardware used

The initial version for use with OpenMP and GPU was developed and tested at Nova system 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).

Final tests of the hybrid implementation were conducted at the Prometheus supercomputer at the AGH Cyfronet Supercomputing Centre. Nodes used for simulations were based on Intel Xeon E5-2680v3 processors with 24 cores each and clock at 2.5GHz working under the Linux CentOS 7 operating system. Each node has 128 GB system memory and two Nvidia Tesla K40 XL. The Tesla K40 XL is a GPGPU accelerator, which has 2880 CUDA cores, its maximum clock frequency is 928MHz and it has 12 GB DDR5 of available memory.

IV. 3. Implementation and testing

The implementation work started from basic code improvements, such as a porting code from Fortran77 to Fortran90, code reorganization and refactoring. Some effort was also put into improving implementation for generation of configurations of electrons distributed among possible single particle energy levels. The next step was a compilation of the MLM42PCAP application using gfortran from the GNU compiler suite (v4.9.2) and performance analysis in order to find bottlenecks. The compilation was completed successfully and the application was analysed with a 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 contained basic code improvements. The analysis has shown that the `twooper` function is using over 80% of the processor time (Fig. 1). It was an obvious choice to check if it can be improved, i.e. by implementation on the CUDA device [15, 16].

IV. 4. Single GPU implementation

In the initial implementation of the MLM42PCAP the `twooper` function was called within three, or even four, nested loops (Fig. 2). As it was noticed, each execution of the `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(...)
    end do
      ...
    k=1
    do
      k=k+1
      ...
      do i=1, iconf
        call twooper(...)
      end do
    end do
      ...
    do i=1, iconf
      call twooper(...)
    end do
      ...
  end do
end do
```

Fig. 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 Fig. 3. The most inner loop has been removed. Two additional functions were written in C and CUDA C to a provide proper interface for call from the Fortran code.

```
call cu_twooper(max_iconf, iconf,
                     ...
                    sdbl_vec_size, 0)
```

Fig. 3. CUDA function call introduced in the Fortran code

```
extern "C" void cu_twooper_(
    const fint8 *max_iconf,
    ...

    const fint8 *tid)
{
  // memory allocation
  checkCuda( cudaMalloc((void**)&Mat_P,
    *iconf*(*N_st)*sizeof(fint8)) );
  ...
  checkCuda( cudaMalloc((void**)&sdbl_vec_P,
    (*sdbl_vec_size)*2*sizeof(fint8)) );

  // copy data to the device memory
  checkCuda( cudaMemcpy(Mat_P,
    Mat,
    *iconf*(*N_st)*sizeof(fint8),
    cudaMemcpyHostToDevice) );
  ...
  checkCuda( cudaMemcpy(sdbl_vec_P,
    sdbl_vec,
    (*sdbl_vec_size)*2*sizeof(fint8),
    cudaMemcpyHostToDevice) );

  unsigned threads = 256;
  // number of threads: max is 65535 in x,y,z
  unsigned grids = (65535 < *iconf/threads ? 65535 : *iconf/threads) + 1;

  cu_inner_loop<<<grids, threads>>>(*iconf,
    ...
    *sdbl_vec_size);
  // copy data from the pin memory
  checkCuda( cudaMemcpy(rr,
    rr_P,
    (*max_iconf)*sizeof(fcomplex8),
    cudaMemcpyDeviceToHost) );

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

Fig. 4. Interface to call function within the Fortran code

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

In the CUDA kernel function `cu_inner_loop` (Fig. 5) each thread calculates a nonzero matrix element and multiplies the matrix by an initial vector. The matrix is in a block diagonal form with a 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 sub-spaces. Every CUDA thread performs computation for one configuration of particles. In general each thread creates a new configuration by annihilation "ones" within the given configuration and creates

```
__global__ void cu_inner_loop(const fint8 iconf,
                              ...
                              const fint8 sdbl_vec_size) {
  fint i,j,k,l; // indexes for each thread

  // local variables
  fint8 p, empty[MAX_SIZE], cf[MAX_SIZE];
  ...
  __shared__ fcomplex8 rr_pp[256];

  p = blockIdx.x * blockDim.x + threadIdx.x;
  rr_pp[threadIdx.x] = rr[p];
  __syncthreads();
  if (p < iconf) { // check if indexes are valid
    for(ii=0; ii<N_part; ++ii)
      cf[ii] = Mat_P[p + ii*iconf];

    for(k=0; k < N_part; ++k) {
      for(l=k+1; l < N_part; ++l) {

        for(int ii=0; ii < N_st-N_part; ++ii)
          empty[ii] = emp[p + ii * iconf];
        ...
        for(j=0; j < N_st-N_part+1; ++j) {
          for(i=j+1; i < N_st-N_part+2; ++i) {
            empi = empty[i]-1;

            bin2 = bin | (1ULL << (N_st-(empj+1)));
            bin2 |= (1ULL << (N_st-(empi+1)));
            ...
            rr_pp[threadIdx.x].real = rr_pp[threadIdx.x].real +
            (mac.real * vec1[s_dbl-1].real - mac.imag * vec1[s_dbl-1].imag);
            rr_pp[threadIdx.x].imag = rr_pp[threadIdx.x].imag +
            (mac.real * vec1[s_dbl-1].imag + mac.imag * vec1[s_dbl-1].real);
          }
        }
      }
    }
  }
  rr[p] = rr_pp[threadIdx.x];
  __syncthreads();
}
```

Fig. 5. CUDA kernel function `cu_inner_loop`

new "ones". Index p for a new configuration is calculated using a hash table. A new configuration is checked if it is in the same subspace. Finally, a matrix is multiplied by a vector and results are stored in an output vector.

IV. 5. Multi GPU implementation

After completing the implementation that utilized a single GPU, a new implementation was designed and programmed, which can divide all computations between GPUs available within a node. In order to achieve this goal, one needs to modify two of the most outer loops in the Fortran code (Fig. 2) and combine them into one loop (Fig. 6). In the presented listing OpenMP directives are also visible. The idea is to execute function `gpu_sub` for a number of GPUs by a given number of threads. In an ideal case, when the number of threads is equal to the number of available GPUs, each thread will execute `gpu_sub` exactly on one GPU. In the used configuration there were only 2 GPUs, so mod 2 thread executed the

given function on either the first or second GPU.

IV. 6. Hybrid MPI+CUDA implementation

In order to improve time of execution of the MLM42PCAP code we implemented a version designed to run in the hybrid environment based on MPI [14] and GPU computing. MPI is dedicated for splitting problems across several nodes/processes, and on every node GPU is used independently. Based on previous implementations for GPU and multiGPU code was adapted (with minor changes) for use at hybrid MPI+GPU nodes.

In this section details of the implementation are presented along with code and comments. Thanks to work done in implementing the multiGPU version of the MLM42PCAP only minor changes were needed.

According to the MPI philosophy there is one master node that distributes work and data between all other nodes. In the considered case the master node reads data from input files and then sends it to all other processes (listing 7). The data is read into four arrays: kxy, kxyid, el and eval, so only those data structures need to be distributed.

The main loop was divided across many nodes (Fig. 8) in a way that every node is computing its own subset of iterations. It was possible to adapt directly multiGPU implementation which enabled splitting calculations across many GPUs.

Each call of the subroutine lanczos_sub prepares all the necessary auxiliary data structures for execution at the GPU, which is needed to perform main computations. Configuration of electrons for each simulation is also generated in this subroutine.

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

Fig. 6. The source code executing computations on multi GPU

V. COMPUTATIONAL EXPERIMENTS

A set of computational experiments has been performed using three a different problem's sizes (small – eva34, medium – eva46 and large – eva56). Digits in problem names denotes Kx and Ky subspace length. 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. The upper limit for the thread number was set. On the other hand in a CUDA experiment 8, 16, 32, 64, 128 and 256 threads were used. Hybrid MPI+CUDA implementation were performed with a fixed CUDA number of threads: 64 and a changing number of MPI nodes: 1, 2, 4, 8, 16.

```
! MPI read local data at local node
if (mpi_my_id == 0) then
  open(1, file='input/'//arg1)
  open(2, file='input/'//arg2)
  ! reading data into variables el, eval
    ↪ , kxy, kxyid
  ...
  do i=1, mpi_num_procs-1
    call MPI_SEND(el, size(el),
      ↪ MPI_COMPLEX,
      i, 0, MPI_COMM_WORLD,
      ierr)
    ...
    call MPI_SEND(kxyid, size(kxyid),
      MPI_INTEGER, i, 0,
      MPI_COMM_WORLD, ierr)
  end do
else
  call MPI_RECV(el, size(el),
    ↪ MPI_COMPLEX,
    0, 0, MPI_COMM_WORLD,
    mpi_status, ierr)
  ...
  call MPI_RECV(kxyid, size(kxyid),
    MPI_INTEGER, 0, 0,
    MPI_COMM_WORLD,
    mpi_status, ierr)
endif
```

Fig. 7. Distributing data across nodes and receiving it

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) only 4 particles and 12 states with the maximum number of acceptable configuration 160 were considered. 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. All presented tests were performed at the Prometheus supercomputer at the AGH Cyfronet supercomputing centre. Nodes used for simulations were based on Intel Xeon E5-2680v3 processors with 24 cores each and clock at 2.5GHz. Each node has 128GB system memory and two Nvidia Tesla K40 XL.

```
do ix=mpi_my_id+1, Nx*Ny, mpi_num_procs
    call lanczos_sub(kxyid(ix, 1), kxyid
      ↪ (ix, 2),
        ...
        mod(mpi_my_id, cuda_gpu_num))
end do
```

Fig. 8. Algorithm main loop

Computation time obtained during test runs was used to calculate speedup values. The value of relative speedup – s can be found by the following expression $s = \frac{t_s}{t_p}$, where t_s constitutes the computational time of sequential algorithm (no GPUs and one OpenMP thread) and t_p – computational time of parallel algorithm. Every test run was repeated 10 times and average computation time was calculated. Tab. 1 contains performance results for OpenMP code executed on the Prometheus system.

Tab. 1. Performance results (speedup) for OpenMP implementation

OpenMP threads	Prometheus(Intel Xeon E5-2680v3)		
	eva34	eva46	eva56
2	1.20	1.80	1.87
4	1.42	2.74	3.26
8	1.71	4.06	4.04
16	1.36	4.01	4.34
32	0.94	4.00	4.78
64	0.64	4.03	4.85

For eva34 test case the speedup value is increasing from value 1.2 to 1.36 along with increasing the number of OpenMP threads from 2 to 8. Then the increasing number of OpenMP threads results in decreasing the speedup value – for a small problem size there is no significant benefit from parallel computing. Very similar behaviour can be observed on the Prometheus system for eva46 test case, but in this case the speedup is stabilized around factor 4. For eva56 test case speedup value is increasing along with increasing the number of OpenMP threads. This behavior of speedup values can be explained by Amdahl's law [17] (for fixed problem size speedup is limited) and Gustafson's law [18] (the speedup value increases with the problem size).

Tab. 2 contains performance results for the CUDA code executed on Tesla K40 XL located within the Prometheus cluster for one and two GPUs. For eva34 test case speedup values are very small and do not change with the increasing number of CUDA threads per block in one and two GPU cases. For eva46 test case speedup values stabilized around factor 27 for the single GPU and around factor 52 for two GPUs. Results are independent of the number of threads per block. On the other hand, speedup achieved for the largest dataset (eva56) increased along with the number of threads per block.

The maximum speedup for this case was a factor 21.76 for a single GPU with the 64 threads per block and factor 41.77 for two GPUs with the 256 threads per block. The best value of speedup (eva46) for one GPU is equal 27.34 and for two GPUs is equal 53.31 are, respectively, 5x and 10x better in comparison with the best OpenMP speedup equals 4.85.

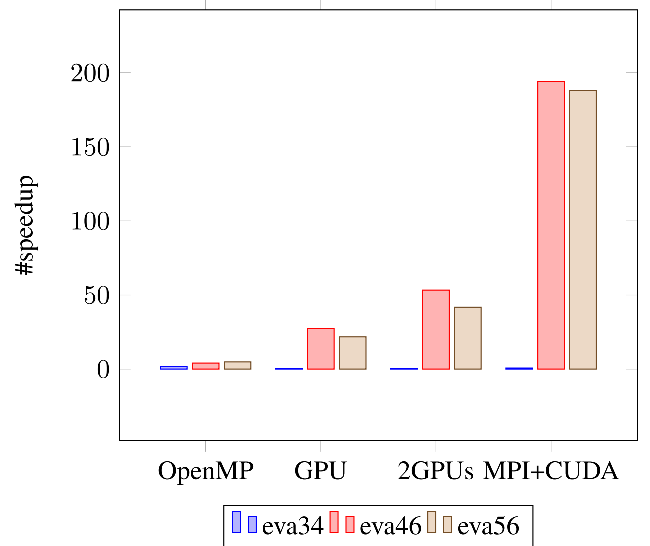

Fig. 9. Best speedup values

Tab. 3 contains performance results for hybrid MPI+CUDA code executed on nodes based on 2 x Intel Xeon E5-2680v3 and 2 x Tesla K40 XL. This configuration allowed reducing the computation time. In the future it will give a possibility to simulate larger systems. As for previous tests with GPU implementations this implementation for eva34 resulted in small speedup. Nevertheless, speedup increases with the number of MPI nodes. For eva46 and eva56 the obtained speedups are larger than those achieved during 2 GPUs tests. Four MPI nodes ensured better performance than the best ones achieved in earlier tests. The best speedups were achieved for 16 MPI nodes, which are 194.03 and 188.02 for eva46 and eva56, respectively. It is 40x and 38x better in comparison with the best OpenMP speedup equals 4.85.

Best values of speedup for OpenMP, CUDA and hybrid MPI+CUDA are shown in Fig. 9. It can be seen that speedup values for the code executed on GPU are significantly larger than for the code executed on CPU, but using hybrid implementation is most efficient. For large test case data size

Tab. 2. Performance results (speedup) for CUDA implementation

CUDA threads per block	speedup on one GPU			speedup for two GPUs		
	eva34	eva46	eva56	eva34	eva46	eva56
8	0.32	27.34	17.39	0.42	53.06	33.22
16	0.32	27.27	17.36	0.42	52.15	33.81
32	0.31	27.15	21.47	0.43	52.61	40.67
64	0.32	27.27	21.76	0.42	53.01	41.17
128	0.31	27.01	21.23	0.43	53.31	41.62
256	0.31	27.18	21.21	0.42	52.84	41.77

copied from the host to device memory is about 15MB and about 75kB for copying the results back to the host memory. A similar problem with copying data between nodes occurs within the hybrid code. As for CUDA it is also visible only for the smallest example – eva34.

Tab. 3. Performance results (speedup) for hybrid MPI+CUDA implementation. At every node two GPUs were used with 64 threads per block

MPI Nodes	Prometheus(2 x Tesla K40 XL)		
	eva34	eva46	eva56
1	0.18	19.23	14.48
2	0.27	34.39	26.52
4	0.47	67.58	49.44
8	0.56	132.37	97.84
16	0.70	194.03	188.02

VI. CONCLUSION

In conclusion, our work demonstrated the potential of using GPU accelerators for improving performance and scalability in material physics simulations. Hybrid architecture based on nodes with GPUs was also tested and verified for use 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 a `gprof` tool was used and a configuration interaction method was identified as most promising to port on GPU. The prepared GPU implementation provides a significant increase of performance (x5 speedup) in comparison with parallel OpenMP base implementation. The nature of the 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 the next significant increase of performance – x10 speedup for execution on two GPUs. The next step was to split the calculations between independent nodes with GPUs. Data was distributed within set of MPI nodes, which has 2 GPUs each. Hybrid MPI+CUDA approach results in next significant increase of performance – x40 speedup for execution with 16 MPI nodes. The proposed approach results in shortening of the computation time and gives a possibility to simulate larger electron-electron interaction systems in future exascale HPC systems with GPUs.

Acknowledgments

This research was supported in part by PLGrid Infrastructure and Wroclaw Centre for Networking and Supercomputing HPC infrastructure (grant No. 368).

This work was financially supported in part by the PRACE-4IP project funded 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 (decision 3563/H2020/2016/2 and DIR/WK/2016/18).

This work was financially supported in part by the National Science Center (NCN), Poland, Grant Sonata No. 2013/11/D/ST3/02703.

References

[1] A. Szabo, N.S. Ostlund, *Modern quantum chemistry: introduction to advanced electronic structure theory*, New York 1982.

[2] T. Neupert, L. Santos, C. Chamon, C. Mudry, *Fractional Quantum Hall States at Zero Magnetic Field*, Physical Review Letters **106**, 236-804 (2011).

[3] Y. Wang, Z. Gu, C. Gong, D.N. Sheng, *Fractional Quantum Hall Effect of Hard-Core Bosons in Topological Flat Bands*, Physical Review Letters **107**, 146-803 (2011).

[4] D.N. Sheng, Z. Gu, K. Sun, L. Sheng, *Fractional quantum Hall effect in the absence of Landau levels*, Nature Communications **2**, (2011).

[5] N. Regnault, B.A. Bernevig, Fractional Chern Insulator, Physical Review X 1, 21-14 (2011).
[6] B. Jaworowski, A. Manolescu, P. Potasz, *Fractional Chern insulator phase at the transition between checkerboard and Lieb lattices*, Physical Review B 92, 245-119 (2015).
[7] A.D. Güçlü, P. Potasz, O. Voznyy, M. Korkusinski, P. Hawrylak, *Magnetism and Correlations in Fractionally Filled Degenerate Shells of Graphene Quantum Dots*, Physical Review Letters 103, 246-805 (2009).
[8] A.D. Güçlü, P. Potasz, P. Hawrylak, Electronic Shells of Dirac Fermions in Graphene Quantum Rings in a Magnetic Field, Acta Physica Polonica A **116**, 832-834 (2009).
[9] T.M. Lahey, T.M. Ellis, FORTRAN 90 Programming, Boston 1994.
[10] B. Chapman, G. Jost, R. van der Pas, Using OpenMP: Portable Shared Memory Parallel Programming, Cambridge 2007.
[11] OpenMP specification, http://www.openmp.org/specifications/, [Online; accessed 14-November-2018].
[12] NVIDIA CUDA C Programming Guide, https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.
[13] MPI-forum, http://mpi-forum.org/, [Online; accessed 14-November-2018].
[14] OpenMPI homepage, https://www.open-mpi.org/, [Online; accessed 14-November-2018].
[15] M. Hruszowiec, P. Potasz, A. Szymańska-Kwiecień. M. Uchroński, *Using GPU Accelerators for improving Performance and Scalability in Material Physics Simulations*, `www.prace-ri.eu/IMG/pdf/WP235.pdf`, 2017, [Online; accessed 14-November-2018].
[16] B.B. Gursoy, M. Browne, M. Lysaght, *Evaluation of Tools and Techniques for Future Exascale Systems*, www.prace-ri.eu/IMG/pdf/D7.4_4ip.pdf", 2017, [Online; accessed 14-November-2018].
[17] G.M. Amdahl, *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*, Proceedings of the Spring Joint Computer Conference, 483-485 (1967).
[18] J.L. Gustafson, Reevaluating Amdahl's Law, *Communications of the ACM* **31**, 532-533 (1988).

Mariusz Uchroński is a lead programmer at Wroclaw Centre for Networking and Supercomputing (WCSS) and researcher at Wroclaw University of Science and Technology, Department of Control Systems and Mechatronics. He obtained his PhD from Wroclaw University of Science and Technology, Institute of Computer Engineering, Control and Robotics in 2014 in the field of control and robotics. His areas of research interest include parallel algorithms, software design and development, HPC computing, scheduling and discrete optimization. His scientific achievements: co-author of several scientific papers published in peer-reviewed journals and conference proceedings in the field of parallel processing, software design and development, scheduling and optimization. He was involved in several R&D projects, such as PRACE 2IP/3IP/4IP/5IP, PLGrid, SPIN-LAB, AZON.

Paweł Potasz received his PhD from the Institute of Physics, Wroclaw University of Science and Technology, in 2012, for his work on electronic and optical properties of graphene nanostructures. His research interests are concentrated around electronic properties of low-dimensional systems, correlated phases and topological effects. Currently, he is assistant professor at the Department of Theoretical Physics, Faculty of Fundamental Problems of Technology, at Wroclaw University of Science and Technology.

Agnieszka Szymańska-Kwiecień received her master's degree in Computer Science in 2001 from Wroclaw University of Science and Technology, Faculty of Computer Science and Management. Currently she is head of the Project Development Team at Wrocław Centre for Networking and Supercomputing and works as an application and programming technologies expert. Her main interests concern parallel and distributed computing, resource management in distributed environments and use of novel architectures in HPC. She has been involved in national and EU projects in the field of HPC and Grid computing, distributed storage and network security, as coworker, key personnel and task or project manager.

Mariusz Hruszowiec graduated in Applied Computer Science from Wrocław University of Technology in 2012. At present he is a PhD student at the Faculty of Electronics at Wrocław University of Technology. The main topics of his interest are gyrotron theory, electromagnetic field theory and numerical methods. Simultaneously he is working at Nokia at the Smart Radio department. His work is focused around digital signal predistortion.

 DOI:10.12921/cmst.2018.0000025