Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Optimization of SHAKE and RATTLE Algorithms

Mariusz Uchronski[a], Marcin Gebarowski[a], Agnieszka Kwiecien[a,*]

[a]*Wroclaw Centre for Networking and Supercomputing (WCSS), Wyb. Wyspianskiego 27, 50-370 Wroclaw, Poland*

Abstract

SHAKE and RATTLE algorithms are widely used in molecular dynamics simulations and for this reason are relevant for a broad range of scientific applications. In this work, an existing CPU+GPU implementations of the SHAKE and RATTLE algorithms from the DL_POLY application are investigated. DL_POLY is a general purpose parallel molecular dynamics simulation package developed at Daresbury Laboratory by W. Smith and I.T. Todorov. OpenCL code of the SHAKE algorithm for DL_POLY application is analyzed for further optimization possibilities. Our work with RATTLE algorithm is focused on porting of the algorithm from Fortran to OpenCL and adjusting it to the GPGPU architecture.

1. The goals of the project

In this project the work is focused on the optimization of SHAKE and RATTLE algorithms, using the DL_POLY Molecular Simulation Package [1] as a base. Our plan is to evaluate these algorithms and further develop OpenCL versions of the main parts of them (Leapfrog Verlet and Velocity Verlet integration schemes). The main goal is increasing the potential of the algorithms to support asynchrony and check the possibility of improving the accuracy of the GPU code.

2. Work done in the project

SHAKE is a two stage algorithm based on the Leapfrog Verlet integration scheme. The RATTLE algorithm fits within the concept of the Velocity Verlet integration scheme [2]. These algorithms are widely used in molecular dynamics simulations and for this reason are relevant for a broad range of scientific applications. As a base for the study we used the DL_POLY application, which provides implementations of the SHAKE and RATTLE algorithms on CPU and some parts on GPU using CUDA, and an OpenCL partial implementation, developed by WCSS (within T7.5 / T7.2 of PRACE-1IP).

SHAKE

Implementation of the SHAKE algorithm for DL_POLY application has been continued and code has been analyzed to find possibilities for further optimizations.

Some performance bottlenecks of SHAKE OpenCL implementation have been identified. One of them is an initialization of the OpenCL environment – a possible solution for this issue is to divide OpenCL routines into separate contexts, for example create one OpenCL context for SHAKE algorithm and another OpenCL context for RATTLE algorithm.

The next bottleneck is GPU I/O operations – this is a well-known issue for GPU computations. This problem occurs while accessing GPU memory (by “accessing” we mean copying data between host and GPU, reading/writing from kernel code). Memory manipulations between host and GPU are very time consuming. In most situations it is better to copy more data than copy small amounts many times. I/O bottleneck

[*] Corresponding author email address: agnieszka.kwiecien@pwr.wroc.pl

in kernels must be tackled considering the utilized memory type (global, local, texture) on GPU. Every type has its advantages and disadvantages. Global memory (same as texture) has a considerably larger capacity, which makes it possible to store more data in it. However, access times to this memory type are quite high. Texture memory is cached and if accessed properly is significantly faster than the global memory, but one cannot store every data type in it. In addition, a texture object can be read or write only. The local (or shared) memory is, as described later, the fastest one but has very limited size. Due to these set of properties, the memory type to be utilized in GPU must be selected properly, considering the kind of operations to be performed on it.

The third bottleneck is due to synchronization between MPI processes in a multi-GPU environment. This kind of synchronization requires copying data from GPU memory to local memory, synchronizing the data and then copying synchronized data back to the GPU memory. There are already implemented kernels used for improving efficiency of data transfers between host (_hs) and GPGPU device (_dv): *gather_dv_scatter_hs* and *gather_hs_scatter_dv*. The data transfers are required for a synchronization of MPI processes and they are large in volume. The kernels are packing or unpacking data in parallel on a device, depending on the direction of communication. The only other solution for this issue in MPI+OpenCL code is to minimize the number of synchronization operations.

RATTLE

DL_POLY provides an implementation of the RATTLE algorithm written in Fortran. Our work focused on porting the algorithm to OpenCL and adjusting it to the GPGPU architecture.

The first step was an analysis of the algorithm and its Fortran implementation to identify possible points for improvements. The source code in Fortran has 175 lines and contains many loops, which makes it a perfect candidate for parallelization. The module was divided into two sections:

- calculate velocity constraint corrections,
- update velocities locally.

These sections have been implemented as OpenCL kernels to run on GPGPU devices (Figure 1). The first implementation of the kernels was a simple port of the Fortran code. Further study of the input data led to a discovery of several major problems of the algorithm, when parallelized on GPGPU. Multiple threads running on GPGPU were assigned to the same indexes causing memory collisions on *read*/*write* operations.

The *read* collisions decrease the program performance due to higher memory access latency. These collisions can be eliminated in a number of ways. One solution is to use a local (shared) GPGPU memory. The access time for such memory is significantly smaller than for the global memory and read collisions do not cause such big delays. However, this solution has its own costs. First of all, GPU devices usually do not have a lot of local memory (for GTX 480 it is 48KB per Streaming Multiprocessor (SM)). In GPGPU, threads are organized in groups, which are then assigned to SM. This means that every group of threads has its own shared memory with data in it and other groups cannot access it.

Taking into account these constraints, we decided to use the local memory in the OpenCL kernels as a buffer for the data required in a current iteration of computations. We determine which data will be needed using indexes assigned to threads. Buffers in the local memory have size equal to groups sizes, which can lead to situations when some threads cannot compute in current iteration because there is no data for them. For example we have 64 threads in group. Thread with lowest id needs data from index 1 and 2, and last thread needs data from 64 and 65. Rest of the threads need data between indexes 1 and 64. As mentioned earlier local array can hold only 64 elements (because that is the size of group). This means that last thread cannot compute because it lacks data from index 65. In such situations the respective thread waits and its task will be given to the first thread in the next group or in the same group but in next iteration.

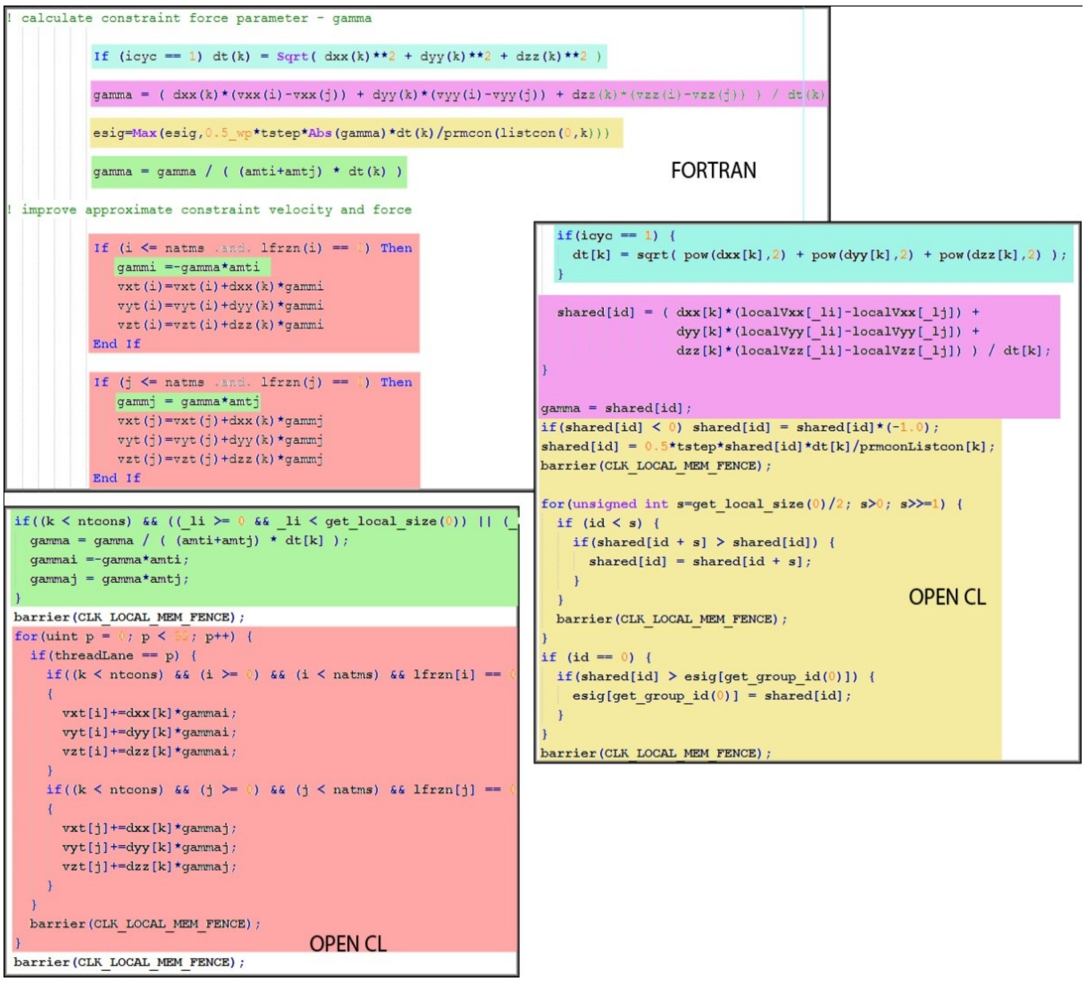
Second type of memory access collisions identified in the implementation is those which occur during *write* operations. They are worse in terms of possible consequences and more difficult to eliminate than the *read* conflicts. Even a simple code like: *x = x +1;* is liable to *write* collision, when executed by two threads.

Possible scenarios are:

- Thread#1 reads *x* value and adds *1* to it writing result to memory. Thread#2 reads the value which is now *x + 1* and final result will be *x + 1 + 1*.
- Thread#1 reads *x* value and adds *1* to it but before it will write result Thread#2 reads the memory. Thread#1 writes its result *x + 1* which is later overwrite by Thread#2 with *x + 1*.

The results obtained in the first and second scenario are different. We can also imagine a scenario where final result will be "not a number'' (NAN) because of some sort of collision or overwrite of some bits. This example shows how important it is to avoid *write* hazards, which can lead to non-deterministic behavior of the algorithm.

Write collisions inside a thread group were fixed by using sequential writing. Only one thread with the *id* equal to some iterator can write and the rest must wait. Utilizing warps leaves only 32 threads we must take care of because only these threads can execute commands in the same time (we know that thread with *id=T+32* will not be writing in the same time as thread with *id=T*). Warp is a definition of group of 32 threads which execute one common instruction at the time. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes on each branch path, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path [4]. Group can have more than one warp, but only one warp will be working at the time. This property was used in collision-less code.



```
! calculate constraint force parameter - gamma

        If (icyc == 1) dt(k) = Sqrt( dxx(k)**2 + dyy(k)**2 + dzz(k)**2 )

        gamma = ( dxx(k)*(vxx(i)-vxx(j)) + dyy(k)*(vyy(i)-vyy(j)) + dzz(k)*(vzz(i)-vzz(j)) ) / dt(k)

        esig=Max(esig,0.5_wp*tstep*Abs(gamma)*dt(k)/prmcon(listcon(0,k)))

        gamma = gamma / ( (amti+amtj) * dt(k) )

! improve approximate constraint velocity and force

        If (i <= natms .and. lfrzn(i) == 0) Then
           gammi =-gamma*amti
           vxt(i)=vxt(i)+dxx(k)*gammi
           vyt(i)=vyt(i)+dyy(k)*gammi
           vzt(i)=vzt(i)+dzz(k)*gammi
        End If

        If (j <= natms .and. lfrzn(j) == 0) Then
           gammj = gamma*amtj
           vxt(j)=vxt(j)+dxx(k)*gammj
           vyt(j)=vyt(j)+dyy(k)*gammj
           vzt(j)=vzt(j)+dzz(k)*gammj
        End If
```

```
  if(icyc == 1) {
    dt[k] = sqrt( pow(dxx[k],2) + pow(dyy[k],2) + pow(dzz[k],2) );
  }

  shared[id] = ( dxx[k]*(localVxx[_li]-localVxx[_lj]) +
                 dyy[k]*(localVyy[_li]-localVyy[_lj]) +
                 dzz[k]*(localVzz[_li]-localVzz[_lj]) ) / dt[k];
}

gamma = shared[id];
if(shared[id] < 0) shared[id] = shared[id]*(-1.0);
shared[id] = 0.5*tstep*shared[id]*dt[k]/prmconListcon[k];
barrier(CLK_LOCAL_MEM_FENCE);

for(unsigned int s=get_local_size(0)/2; s>0; s>>=1) {
  if (id < s) {
    if(shared[id + s] > shared[id]) {
      shared[id] = shared[id + s];
    }
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
if (id == 0) {
  if(shared[id] > esig[get_group_id(0)]) {
    esig[get_group_id(0)] = shared[id];
  }
}
barrier(CLK_LOCAL_MEM_FENCE);
```

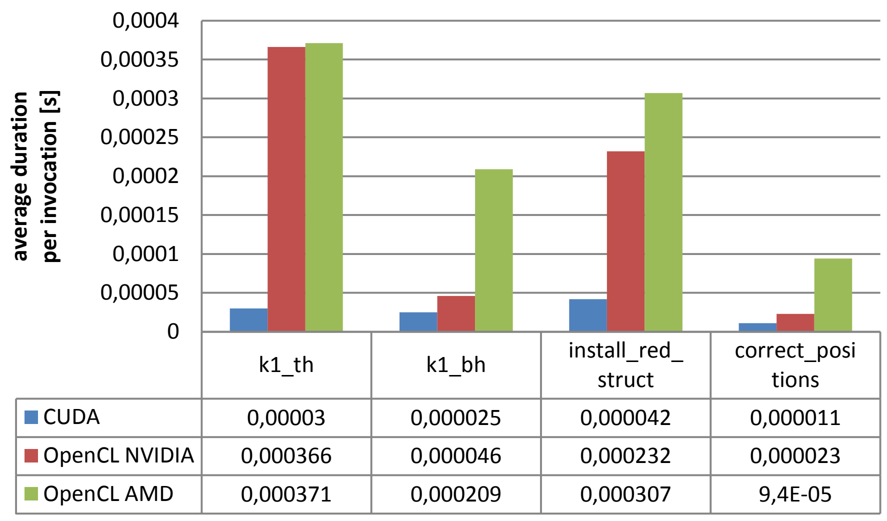
```
if((k < ntcons) && ((_li >= 0 && _li < get_local_size(0)) || (_
  gamma = gamma / ( (amti+amtj) * dt[k] );
  gammai =-gamma*amti;
  gammaj = gamma*amtj;
}
barrier(CLK_LOCAL_MEM_FENCE);
for(uint p = 0; p < 32; p++) {
  if(threadLane == p) {
    if((k < ntcons) && (i >= 0) && (i < natms) && lfrzn[i] == 0
    {
      vxt[i]+=dxx[k]*gammai;
      vyt[i]+=dyy[k]*gammai;
      vzt[i]+=dzz[k]*gammai;
    }
    if((k < ntcons) && (j >= 0) && (j < natms) && lfrzn[j] == 0
    {
      vxt[j]+=dxx[k]*gammaj;
      vyt[j]+=dyy[k]*gammaj;
      vzt[j]+=dzz[k]*gammaj;
    }
  }
  barrier(CLK_LOCAL_MEM_FENCE);
}
barrier(CLK_LOCAL_MEM_FENCE);
```

Figure 1. Fortran and OpenCL code comparison

3. Results obtained

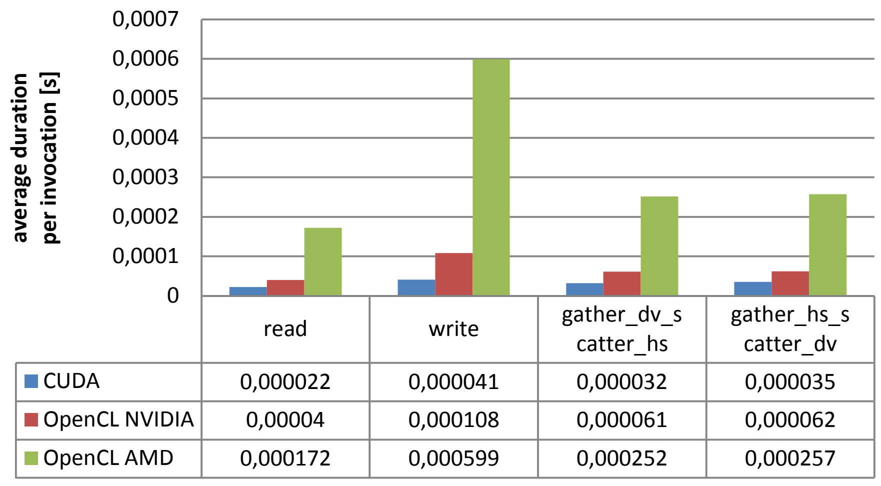
SHAKE

The implementation of the SHAKE algorithm for DL_POLY application has been evaluated and tested on local WCSS GPU machines (2x GTX480, 2x AMD Radeon HD 6900 Series). Performance results for the H2O benchmark show that the OpenCL implementation works slower than the CUDA version of the same algorithms (Figure 2). The biggest difference between an average duration time per invocation for NVIDIA GPUs is for kernels: *k1_th* (OpenCL code is 10x slower than CUDA code), and *install_red_struct* (OpenCL code is 5.5x slower than CUDA code). For other kernels OpenCL calls are 2x slower than particular CUDA calls.



	k1_th	k1_bh	install_red_struct	correct_positions
🟦 CUDA	0,00003	0,000025	0,000042	0,000011
🟥 OpenCL NVIDIA	0,000366	0,000046	0,000232	0,000023
🟩 OpenCL AMD	0,000371	0,000209	0,000307	9,4E-05

Figure 2. CUDA vs OpenCL kernels for DL_POLY constraints shake component



	read	write	gather_dv_scatter_hs	gather_hs_scatter_dv
🟦 CUDA	0,000022	0,000041	0,000032	0,000035
🟥 OpenCL NVIDIA	0,00004	0,000108	0,000061	0,000062
🟩 OpenCL AMD	0,000172	0,000599	0,000252	0,000257

Figure 3. CUDA vs OpenCL I/O and communication for DL_POLY constraints shake component

RATTLE

The RATTLE algorithm has been analyzed and parts of it identified as good candidates for optimization by moving computations to GPU. These parts have been successfully ported to OpenCL and integrated with the DL_POLY application. Results for a subset of test cases [3] are presented in Table 1. Tests have been performed at WCSS (2x GTX480). The results are promising, but as for the SHAKE algorithm, the OpenCL environment initialization and I/O operations constitute performance bottlenecks.

Performance results in general show that selected parts of the Fortran code can be executed faster on GPU using OpenCL (overall GPU computation time) but the bottlenecks have great influence on the overall computation time (test cases: H2O, TEST14). Further optimization work should be focused on decreasing this influence.

Benchmark	**Fortran**	**OpenCL**			**Speedup**
		Initialization, I/O	**Kernels**	**Overall**	
H2O	0.001101	0,001792	0,000294	0,002086	0,5278
TEST3	0,152225	0,011471	0,008079	0,01955	7,7864
TEST4	1,247404	0,048104	0,062032	0,110136	11,3260
TEST7	0,134984	0,022904	0,017279	0,040183	3,3592
TEST8	1,09763	0,075339	0,136363	0,211702	5,1847
TEST13	0,162882	0,02114	0,018883	0,040023	4,0697
TEST14	1,280869	3,031746	0,088209	3,119955	0,4105

Table 1. Results for RATTLE algorithm integrated with DL_POLY code (double floating point precision)

4. Conclusions

In this report we presented results of our work on optimizing the SHAKE and RATTLE algorithms. Some performance bottlenecks of SHAKE OpenCL implementation has been identified - initialization of the OpenCL environment, GPU I/O operations and synchronization between MPI processes in a multi-GPU environment. OpenCL version of RATTLE algorithm for DL_POLY application has been implemented and tested. The RATTLE OpenCL algorithm shows up to 11x speedup for the test benchmark with 413896 atoms. Results also show that GPU I/O operations constitute performance bottlenecks.

Using OpenCL for the RATTLE algorithm implementation has one major disadvantage. GPUs are strongly parallel and many collisions between threads may occour. Thus, before starting the implementation the input data and their representation in the language data structures must be analyzed and the structures must be adjusted to the new paradigm. Otherwise the code may not run effectivly. Copying data between GPU and host system is a very time-consuming operation.

From the results obtained it may be concluded that dividing the code into more kernels may decrease application performance, since the program spends too much time on transfering data. To avoid this bottleneck, the kernels should be designed as compute-heavy with a minimal data exchange needs. Some operations of both SHAKE and RATTLE algorithms need to be executed on a CPU, and these are generally some kind of synchronization points, when data from different work groups or whole GPUs are aggregated and then processed by one thread.

Acknowledgement

This work was financially supported by the PRACE-2IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-283493.

References

1. DL_POLY Molecular Simulation Package home page:
http://www.stfc.ac.uk/CSE/randd/ccg/software/DL_POLY/25526.aspx
2. The DL_POLY_4 User Manual:
ftp://ftp.dl.ac.uk/ccp5/DL_POLY/DL_POLY_4.0/DOCUMENTS/USRMAN4.01.pdf
3. The DL_POLY test data:
ftp://ftp.dl.ac.uk/ccp5/DL_POLY/DL_POLY_4.0/DATA/
4. NVIDIA CUDA C Programming Guide:
http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf