Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Benchmarking and Analysis of DL_POLY 4 on GPU Clusters

Michael Lysaght[a], Mariusz Uchroński[b], Agnieszka Kwiecien[b], Marcin Gebarowski[b]
Peter Nash[a], Ivan Girotto[a] and Ilian T. Todorov[c]

[a]*Irish Centre for High End Computing, Tower Building, Trinity Technology and Enterprise Campus, Grand Canal Quay, Dublin 2, Ireland*
[b]*Wroclaw Centre for Network and Supercomputing, Wybrzeze Wyspianskiego 27, 50-370 Wroclaw, Poland*
[c]*STFC Daresbury Laboratory, Daresbury, Warrington WA4 4AD, United Kingdom*

Abstract

We describe recent development work carried out on the GPU-enabled classical molecular dynamics software package, DL_POLY. We describe how we have updated the original GPU port of DL_POLY 3 in order to align the 'CUDA+OpenMP'-based code with the recently released MPI-based DL_POLY 4 package. In the process of updating the code we have also fixed several bugs which allows us to benchmark the GPU-enabled code on many more GPU-nodes than was previously possible. We also describe how we have recently initiated the development of an OpenCL-based implementation of DL_POLY and present a performance analysis of the set of DL_POLY modules that have so far been ported to GPUs using the OpenCL framework.

DL_POLY

1. Introduction

The increasing scale of existing and future European tier-0 supercomputers encourages the use of hybrid programming models to address issues such as declining memory per core, multiple threads per core and the improvement of load-balancing. Currently, the most familiar hybrid model is that combining the use of MPI and OpenMP, where the latter API allows developers to take advantage of intra-node shared memory and where the former is typically employed for internode communication only. Recently, the need for hybrid programming techniques has been further enforced by the adoption of general-purpose Graphics Processing Units (GPUs) within Massively Parallel Processing (MPP) systems. GPUs are recognized as having the potential to considerably speedup or "accelerate" compute intensive algorithms over their equivalent single CPU core implementation, leading to an increase in the utilization of such devices on systems ranging from workstations to small clusters. More recently, GPU coprocessors have been incorporated into large-scale tier-0 architectures [1] and may form the basis for more tightly integrated hybrid tier-0 systems in the near future [2]. The increased heterogeneity introduced by GPUs means that harnessing the capability of large-scale hybrid architectures necessitates the implementation of programming models that go beyond the MPI+OpenMP paradigm, by also taking the available GPU cards on a compute node into account. In this paper we describe recent work that has been carried out on DL_POLY [3,4], a molecular dynamics software package that adopts the hybrid programming model to exploit not only the many-core

shared memory nature of CPU compute nodes, but also makes use of GPU programming frameworks to exploit the increasing availability of such 'accelerators' on state-of-the-art supercomputers. Two separate versions of the GPU-enabled DL_POLY code are considered here; the first (hereafter referred to as DL_POLY_CUDA) uses NVIDIA's CUDA programming framework to port compute intensive components of the DL_POLY package to NVIDIA GPU cards. The second (hereafter referred to as DL_POLY_OpenCL) uses OpenCL, a free open standard framework for writing programs that execute across heterogeneous platforms consisting of CPUs, GPUs, and other processors. Unlike CUDA, OpenCL has the advantage that it is not tied to any vendor and therefore, applications written using OpenCL can be ported to a variety of accelerator-based coprocessors. In this report we consider NVIDIA GPU cards for the DL_POLY_CUDA code and both NVIDIA and AMD GPU cards for the DL_POLY_OpenCL code. The paper is organized as follows: In section 2 we give a brief overview of the DL_POLY_CUDA code and follow this by providing benchmark results for calculations that we have recently performed on the 'Stoney' GPU cluster at ICHEC. In section 3, we provide a brief description of our recent development work on DL_POLY_OpenCL code and follow this up by providing a performance comparison between the OpenCL and CUDA implementations of DL_POLY running on two GPU-based systems at the Wroclaw Centre for Networking and Supercomputing (WCNS) in Poland. Finally, we conclude with our views on where improvements can be further made to optimize DL_POLY for existing and future hybrid MPP systems.

2. The DL_POLY_CUDA code

The DL_POLY_CUDA code [5] continues to be developed at the Irish Centre for High End Computing (ICHEC). The version of the vanilla DL_POLY code that was first ported to GPUs was DL_POLY 3, which was also the first version of DL_POLY to use domain decomposition as a parallelization strategy (DL_POLY is written using Fortran 90 and MPI). On assessing the original DL_POLY code, the components listed in table 1 were identified as the most computationally intensive parts of the application. These components were then ported to GPUs using a single-client approach, in which each host MPI process binds to a GPU device and offloads a proportion of its computations to the GPU. To avoid GPU oversubscription, only one MPI process is run per attached GPU, thus resulting in idle CPU cores. To avoid this problem, most of the DL_POLY functions ported to CUDA have also been parallelized using OpenMP. The computational effort is then dynamically distributed between the OpenMP threads and the GPU. The dynamic load-balancing is performed for each accelerated component on a per-iteration basis and with the exception of the Contraints Shake component, all of the accelerated components listed in table 1 were also ported using OpenMP.

Table 1. CUDA-accelerated DL_POLY components

Component	Purpose
Constraints Shake (CS)	Apply bond constraints between atoms
Link-cell pairs (LCP)	Construct atom neighbor lists
Two-body forces (TBF)	Compute inter-atomic forces
Ewald SPME forces	Compute Coulombic and force terms in periodic systems

Between June 2011 and Feb 2012, several significant modifications were made to the DL POLY 3 code resulting in the release of DL_POLY 4 by the lead developers at STFC Daresbury (Dr. I Todorov). Where these modifications have affected the GPU-enabled components, mirror-like modifications have been implemented within the CUDA+OpenMP code. Updated CUDA+OpenMP-enabled modules have been committed to the source code's main 'CCPForge' trunk repository. On top of the aforementioned mirroring, an error-handling module has been developed for the CUDA+OpenMP code and several bugs affecting memory and structure alignment have been fixed. This has

allowed for the benchmarking of DL_POLY 4 over many more nodes of ICHEC's 'Stoney' GPU cluster than was previously feasible. The work described above was carried out as part of the PRACE 1IP-WP7.5 project.

Figure 1 shows the measured speedup values for two test problems run on ICHEC's 'Stoney' GPU cluster (Both tests were obtained from the DL_POLY ftp site). Each of Stoney's compute nodes has two 2.8 GHz Intel (Nehalem EP) Xeon X5560 quad-core processors and 48 GB of RAM. Twenty-four of the nodes have two NVIDIA Tesla M2090 cards installed, with each card providing 512 GPU cores and 6 GB of GDDR5 memory. Speedup values are shown for each accelerated component and the overall application for three different runtime configurations. The first configuration consists of two GPUs with each GPU attached to one of the two quad-core CPU sockets with 4 OpenMP threads each. In this case a single MPI process is affined to each GPU. The second configuration consists of 4 MPI processes being affined to the GPUs (representing an oversubscription of the GPUs) with 2 OpenMP threads per socket for the CPU-based computations. The third configuration consists purely of MPI processes being affined to the 8 CPU cores available within a node. It can be seen that with each of these configurations the CPU nodes of the cluster are fully populated. All speedup figures are reported for the double precision version of the application.

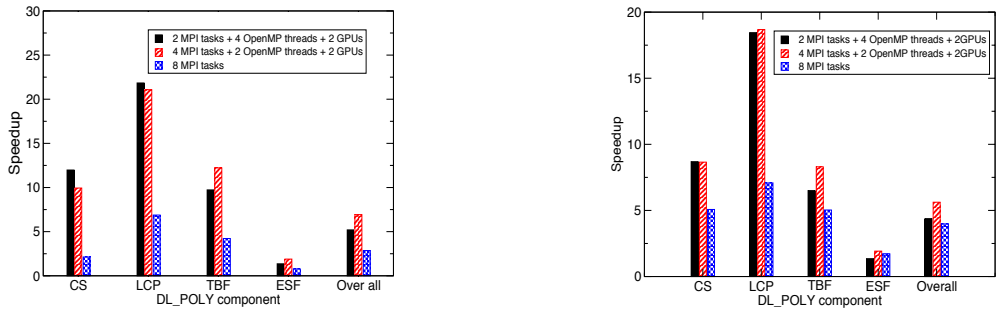

Figure 1 Performance of main GPU-enabled DL_POLY components vs their pure-MPI equivalents. Measurements were performed for TEST4 (a) and TEST8(b) from the DL_POLY benchmark suite.

The maximum component speedup is observed for TEST4 shown in figure 1(a) for which a speedup of ~22 is achieved for the Link-Cell Pairs (LCP) component running on two GPUs with eight OpenMP threads for the host computation. Whereas previous benchmarking results for DL_POLY_CUDA included a performance comparison with only the single-core implementation of the MPI-vanilla code, we also compare the performance of the GPU-enabled code with the MPI-vanilla code running on all available compute cores within the 8-way Stoney node (i.e., full node population) Figure 1(b) shows the results for TEST8 taken from the DL_POLY benchmark suite which represents an increase in the size of calculation, made possible by the recent development work as part of this project.

Figure 2 shows the strong scaling performance of both the vanilla MPI code and the DL_POLY_CUDA code run across 16 nodes of the Stoney GPU cluster. The benchmark calculation used to obtain these results was TEST2. The only modification made to TEST2 was to the 'CONTROL' input file where the nfold parameter was set to (2,2,2) in order to increase the size of the calculation for the purpose of scaling. For these measurements we investigated the same runtime configurations as were used in figure 1, although we do not show the results for the 4 MPI tasks+ 2 OpenMP threads+2 GPUs run as we did not see any significant difference with the results of the 2 MPI tasks + 4 OpenMP threads + 2GPUs run for TEST2. Figure 2 shows that, for the configurations investigated, harnessing the additional 2 GPU cards on each node results in a ~30% reduction in wall-clock time for all node counts.

As part of our recent work on DL_POLY_CUDA we have also investigated the performance of GPU-enabled third party FFT libraries on ICHEC's Stoney GPU cluster in order to evaluate the possible benefits of replacing DL_POLY's custom developed DAFT FFT library[6] which becomes a bottleneck when scaling to a high number of compute nodes. The Parallel Three-Dimensional Fast Fourier Transforms (P3DFFT) is a library [7] currently being developed at the San Diego Supercomputer Centre and has recently been ported to GPUs by a group at the Georgia Institute of Technology (where the ported library is dubbed DiGPUFFT [8]). While recent investigations point to a potential performance benefit of implementing third-party FFT libraries within DL_POLY 4, we have found from our investigations that the performance benefit of using a GPU-enabled FFT library over a pure MPI-based FFT library is, for the moment, even less clear. While we have found a performance advantage (~10% speedup) in using DiGPUFFT over P3DFFT, it should be noted that we have found this improvement only for large datasets (the dataset used was 8 -16 times larger than typical datasets used in DL_POLY). It should also be noted that our performance results were obtained for a single precision dataset, whereas DL_POLY uses double precision datasets. The DiGPUFFT library is currently only enabled for single precision calculations and it is expected that any performance advantage currently seen over P3DFFT will diminish further when using double precision.

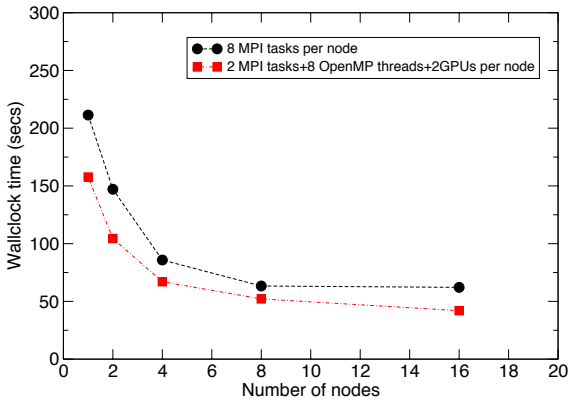

Figure 2. Strong scaling of DL_POLY on ICHEC's Stoney GPU cluster. Two runtime configurations were used: for the pure MPI-based DL_POLY we used 8 MPI tasks per node (full population) and for the DL_POLY_CUDA code we used 2 MPI tasks affined to GPU cards with an additional 8 OpenMP threads for the remainder of any work carried out on the CPU.

Finally, we have also implemented a call to NVIDIA's 3-D cuFFT routine inside DL_POLY. Currently, NVIDIA has no distributed version of its cuFFT library available so any call to the 3-D cuFFT routine must occur on a single CPU affined to a single GPU. With this configuration we have found a factor of ~8 speedup over DAFT running on a single CPU for a dataset of typical size used in DL_POLY. While such an implementation will obviously not scale, the significant speedup may point to the potential benefit of using a 'gather-scatter' approach to the FFT problem when running DL_POLY on small GPU clusters.

3. The DL_POLY_OpenCL code

Stimulated by the success of the CUDA+OpenMP port of DL_POLY to hybrid MPP architectures, we have recently initiated the development of a hybrid implementation of DL_POLY using the OpenCL framework. Here, we discuss our experience with developing with OpenCL so far and also compare the performance of the OpenCL port with the existing CUDA implementation where applicable.

So far, only the Constraints Shake (CS) DL_POLY component has been successfully ported using OpenCL. Several issues were encountered during the porting of this DL_POLY component from CUDA to OpenCL that we consider worth highlighting for the consideration of other developers when porting to GPUs. One of the major issues relates to the differences between the two programming languages in handling C data structures. In the CUDA version of

the CS component, structures are widely implemented both in the host code and GPU kernels (see figure 3 (Top)). According to the OpenCL specification structures in the OpenCL framework cannot contain OpenCL objects (e.g., buffers, images etc.). Our attempts to use structures with buffers (representing a direct port from the CUDA code) failed, as the data inside a buffer was not accessible from the kernel code. This problem forced a change in the way data structures were handled in the DL_POLY_OpenCL code by passing individual objects to kernels directly as arguments, one by one as seen in figure 3(Bottom).

CUDA Implementation:

```
template<typename T_, unsigned int BX_>
__global__ void constraints_shake_cuda_correct_positions_k1()
{
  for(int lI=blockIdx.y*BX_+threadIdx.x;
          lI<CONSTANT_DATA.mNATMS;
          lI+=gridDim.y*BX_)
  {
    int2 lB = CONSTANT_DATA.mIJSo[lI];
    if(lB.x>0)
    {
      T_ lDL = (((T_) 1) / ((real) CONSTANT_DATA.mLISTOT[lI])) * ((T_) lB.x);
      T_ lX=(T_)0, lY=(T_)0, lZ=(T_)0;
      for (int lQ=lB.y; lQ<(lB.y+lB.x); lQ++)
      {
        int lIndex = CONSTANT_DATA.mIJKPos[lQ];
        lX += CONSTANT_DATA.mXXT[lIndex];
        lY += CONSTANT_DATA.mYYT[lIndex];
        lZ += CONSTANT_DATA.mZZT[lIndex];
      }
      CONSTANT_DATA.mXXX[lI] = madd(CONSTANT_DATA.mXXX[lI], lX, lDL);
      CONSTANT_DATA.mYYY[lI] = madd(CONSTANT_DATA.mYYY[lI], lY, lDL);
      CONSTANT_DATA.mZZZ[lI] = madd(CONSTANT_DATA.mZZZ[lI], lZ, lDL);
    }
  }
}
```

OpenCL Implementation:

```
__kernel void constraints_shake_opencl_correct_positions_k1(
              const unsigned int BX_, const int mNATMS,
              __global int2* mIJSo, __global int* mLISTOT,
              __global int* mIJKPos,
              __global double* mXXT, __global double* mYYT,
              __global double* mZZT, __global double* mXXX,
              __global double* mYYY, __global double* mZZZ)
{
  for(int lI=get_group_id(1)*BX_ + get_local_id(0);
          lI<mNATMS;
          lI+=get_num_groups(1)*BX_)
  {
    int2 lB = mIJSo[lI];
    if (lB.x>0)
    {
      double lDL = (((double) 1) / ((double) mLISTOT[lI])) * ((double) lB.x);
      double lX=(double)0, lY=(double)0, lZ=(double)0;
      for(int lQ=lB.y; lQ<(lB.y+lB.x); lQ++)
      {
        int lIndex = mIJKPos[lQ];
        lX += mXXT[lIndex];
        lY += mYYT[lIndex];
        lZ += mZZT[lIndex];
      }
      mXXX[lI] = madd(mXXX[lI], lX, lDL);
      mYYY[lI] = madd(mYYY[lI], lY, lDL);
      mZZZ[lI] = madd(mZZZ[lI], lZ, lDL);
    }
  }
}
```

Figure 3 (Top). CUDA implementation of a kernel associated with the Contraints Shake component of DL_POLY. (Bottom) The equivalent OpenCL implementation.

Once the CS component was fully GPU-enabled test runs were performed on GPU-enabled systems at the WCNS. The first GPU system consisted of an Intel Core i7 CPU with a link to two AMD Radeon HD 6950 GPUs. The second system also consisted of an Intel Core i7 CPU but with a link to a NVIDIA Tesla S2050 GPU card. For runtime configurations we used 2 MPI processes and 2 OpenMP threads for both DL_POLY_CUDA and DL_POLY_OpenCL and used TEST4 for performance measurements (double floating point precision). Figure 4 shows the average duration time per invocation for CS components including initialization, particular kernel calls, and read/write CPU-GPU operations. Performance results show that the OpenCL implementation is slower than the CUDA version for almost all CS component algorithms. The largest difference between average duration time per invocation on the Tesla S2050 GPU was found to be for the 'read' operations (OpenCL code is 21x slower than CUDA code) and the kernel 'gather_dv_scatter_hs' (OpenCL code is 2x slower than CUDA code).

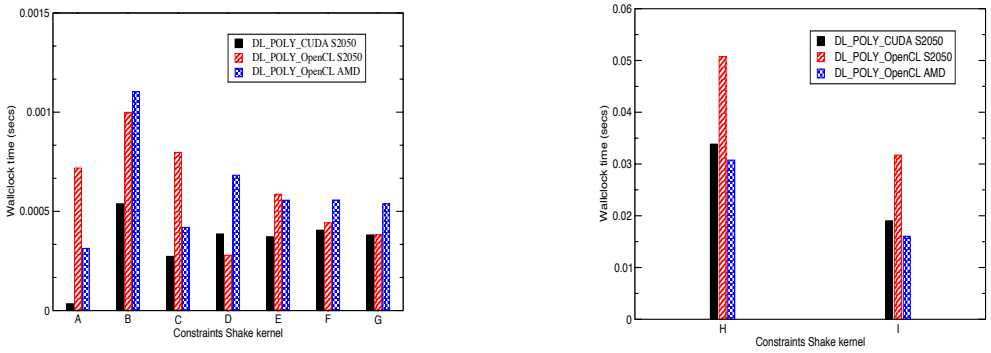

Figure 4 (a) and (b): Performance comparison between DL_POLY_CUDA and DL_POLY_OpenCL running on different systems. Key for Constraints Shake kernels: A=read, B=write, C=gather_dv_scatter_hs, D=gather_hs_scatter_dv, E=k1_th, F=k1_bh, G=correct_positions, H=initialize, I=install_red_struct.

For other kernels OpenCL calls are less than 2x slower than their equivalent CUDA calls. Execution times on AMD GPUs are shorter than for TeslaS2050 GPUs for 'initialization' and kernel 'install_red_struct'. For other calls execution times on AMD GPUs are longer than for the CUDA equivalent run on the Tesla S2050. The latter difference is explained by the fact that the OpenCL code was developed and optimized for NVIDIA GPUs and does not take the specificities of the AMD GPU architecture into account.

4. Conclusion

We have reported on the benchmarking of the 'CUDA+OpenMP' port of the molecular dynamics software package, DL_POLY, where our recent software development focus has been on synchronizing the CUDA+OpenMP version of DL_POLY with the latest changes that have occurred and which have resulted in the release of version 4 of the DL_POLY vanilla MPI code between Dec 2011 and Feb 2012. As a result of updating the CUDA+OpenMP section of the code, we have been able to benchmark the GPU-enabled version of DL_POLY4 on ICHEC's Stoney GPU cluster. For a small problem size we have seen a marked performance advantage in using GPU-enablement over the vanilla MPI code in cases where the pure MPI code has shown good scaling. We have also investigated the possibility of replacing the DAFT FFT library within DL_POLY 4 with the GPU-enabled library, DiGPUFFT, and have found that for the grid sizes of interest to DL_POLY there is, for the moment, no benefit of using the GPU capability of this library over its pure MPI-based equivalent. While we have found an impressive speedup using the single-CPU-core-single-GPU CUFFT library within DL_POLY, we only see this implementation being of benefit

within an MPI 'gather-scatter' strategy on small-scale GPU clusters and are currently in the process of investigating this approach further.

Inspired by the success of porting DL_POLY to hybrid architectures using the CUDA framework we have also described how we have recently initiated the porting of DL_POLY to more general accelerator-based architectures using the OpenCL framework At this early stage of development we have found that employing the OpenCL framework requires more effort than developing with the CUDA framework. However, it is worth emphasizing that OpenCL is not tied to a particular vendor or even to a particular accelerator-based architecture. We have demonstrated this flexibility by benchmarking the DL_POLY_OpenCL code on different accelerator-based architectures including AMD's Radeon cards along with more familiar multicore CPUs. Although the performance results indicate that the DL_POLY_OpenCL code is slower than DL_POLY_CUDA, it should be noted that there is room for further optimization of the DL_POLY_OpenCL code to achieve better performance for particular accelerators. We believe that the experience gained during the development of the OpenCL code for the Constraints Shake component of DL_POLY will greatly benefit the further translating of other CUDA+OpenMP components to OpenCL. This porting will be continued by the WCNS team in PRACE-2IP WP12.2 as the task "Optimization of SHAKE and RATTLE algorithms". The WCNS team also intends to evaluate DL_POLY_OpenCL on PRACE prototype AMD APUs installed at the Poznan Supercomputing and Networking Centre (PSNC) in the near future.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-211528 and FP7-261557. The work was achieved using the PRACE Research Infrastructure resources STONEY (ICHEC, Ireland) and local GPU clusters at WCNS.

References

1. TGCC CURIE system: http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm
2. .CSCS Todi system: http://user.cscs.ch/hardware/todi_cray_xk6/index.html
3. I. Todorov and W. Smith, 2004, Phil. Trans. R. Soc. Lond. A, 362, 1835. 2, 165
4. http://www.stfc.ac.uk/CSE/randd/ccg/software/DL_POLY/25526.aspx
5. C. Kartsaklis, I. T. Todorov, W. Smith, Symposium on Chemical Computations on GPGPUS, 240th ACS National Meeting and Exposition, Boston, 2010
6. I.J. Bush and I.T. Todorov, Comp. Phys. Commun. 175 (5) 323-329 (2006)
7. P3DFFT Home Page, http://code.google.com/p/p3dfft/
8. DiGPUFFT Home Page, http://code.google.com/p/digpufft/