Using Compiler Directives for Performance Portability in Scientific Computing: Kernels from Molecular Simulation*

Ada Sedova[0000−0002−8233−3057], Andreas Tillack[0000−0002−1832−3030], and Arnold Tharrington[0000−0002−2877−8768]

Scientific Computing Group, National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge TN 37830, USA
{sedovaaa, tillackaf, arnoldt}@ornl.gov

Abstract. Achieving performance portability for high-performance computing (HPC) applications in scientific fields has become an increasingly important initiative due to large differences in emerging supercomputer architectures. Here we test some key kernels from molecular dynamics (MD) to determine whether the use of the OpenACC directive-based programming model when applied to these kernels can result in performance within an acceptable range for these types of programs in the HPC setting. We find that for easily parallelizable kernels, performance on the GPU remains within this range. On the CPU, OpenACC-parallelized pairwise distance kernels would not meet the performance standards required, when using AMD Opteron "Interlagos" processors, but with IBM Power 9 processors, performance remains within an acceptable range for small batch sizes. These kernels provide a test for achieving performance portability with compiler directives for problems with memory-intensive components as are often found in scientific applications.

Keywords: Performance portability · OpenACC · Compiler directives · Pairwise distance · Molecular Simulation

1 Introduction

Software development productivity is reduced when sections of high-performing programs must be frequently rewritten in low-level languages for new supercomputer architectures. This is not only a consequence of increased labor costs, but also because the code can become more error-prone due to shortened lifetimes, multiple authors, and the inherent difficulty of programming close to machine-level [?,?,?,?]. Because of such considerations, creating performance portable applications has become an important effort in scientific computing [?,?], and is recognized as a significant software design goal by both the U.S. Department of Energy (DOE) [?,?,?] and the National Science Foundation (NSF) [?].

Classical molecular dynamics (MD) simulation is a popular tool for a number of fields within the physical and chemical sciences [?,?] and has been successfully implemented in the high-performance computing (HPC) setting by several developers [?,?,?,?,?,?,?,?,?,?]. The associated reports pay testimony to the extensive effort involved in porting these programs to different HPC platforms in order to meet increasingly rising standards. A variety of non-portable components are employed in leadership MD programs that allow for cutting-edge performance to be obtained. Some of the most performance-enhancing elements for per-node speedup include the CUDA C language (and CUDA API) for GPU-based acceleration, and architecture-specific SIMD intrinsic functions along with threading for the CPU portions [?,?,?,?,?,?,?]. CUDA C and the CUDA API, for example, is currently usable only with NVIDIA GPUs, so sections of code written in CUDA will have to be rewritten or translated for use on a different GPU-vendor's product; AMD GPUs, for instance, have recently been shown to be competitive to NVIDIA GPUs [?,?]. For optimal performance on CPU-portions of heterogeneous architectures, architecture-specific SIMD instructions implemented with either intrinsic functions or vector instructions are often found to be essential in leadership MD programs [?]: without the use of SIMD, a majority of the processor's capacity may be unused by a program, and many compilers are not effective in auto-vectorizing code [?], but highly optimized SIMD instructions are architecture-specific and require a considerable effort. This amount of effort may not be optimal or even permissible for a domain scientist, as it will detract from time spent in scientific pursuits. Nevertheless, scientific computing needs can often be very niche-specific and thus commercial applications may not provide an adequate computational solution [?]. Modern science has advanced to a level that some amount of computing is required for both the theoretical and experimental branches: while computational science has become recognized as the "third pillar" of science by national agencies such as the NSF [?], current trends indicate that it is now essential to the functioning of the other two [?]. It is thus of great importance that scientific computing initiatives have accessible programming tools to produce efficient code that can be easily ported to a number of HPC architectures, and that the machine-level back ends are re-targeted

and optimized by system or API developers, while maintaining a consistent, unified front-end interface for the computational scientist to use.

High level, compiler-directive based programing models such as OpenACC and OpenMP have the potential to be used as a tool to create more performance portable code [**?**,**?**]. Results of such attempts have been mixed, however [**?**,**?**,**?**,**?**,**?**,**?**]. The creation of a dedicated portable program should provide the most optimal results [**?**,**?**]. Accordingly, here we test the possibility of creating a portable MD application starting with key kernels of the basic algorithm, and acceleration using OpenACC, to assess whether the resulting performance of these kernels is within an acceptable range to be used as part of HPC-based MD programs. This effort provides tests of the performance of OpenACC on kernels that involve non-negligible memory operations, and large memory transfers to the GPU, characteristic of many scientific applications. The kernels also represent calculations important to other types of computational work such as classification and data analysis.

2 Background

2.1 Performance Portability

To quantify portability, an index has been proposed, the degree of portability (DP):

$$DP = 1 - (C_P/C_R) \tag{1}$$

where C_P is the cost to port and C_R is the cost to rewrite the program [**?**]. Thus, a completely portable application has an index of one, and a positive index indicates that porting is more profitable. There are several types of portability; binary portability is the ability of the compiled code to run on a different machine, and source portability is the ability of the source code to be compiled on a different machine and then executed [**?**,**?**,**?**]. Here, costs can include development time and personnel compensations, as well as error production, reductions in efficiency or functionality, and even less tangible costs such as worker stress or loss of resources for other projects. For the HPC context, we can say that an application is performance portable if it is not only source-portable to a variety of HPC architectures using the Linux operating system and commonly provided compilers, but also that its performance remains in an acceptable range to be usable by domain scientists for competitive research. To avoid the ambiguity in the phrase "acceptable range," Pennycook proposed the following metric for PP [**?**,**?**]:

$$PP(a,p,H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } a \text{ is supported } \forall i \in H \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

where $|H|$ is the cardinality of the set H of all systems used to test the application a, p are the parameters used in a, and e_i is the efficiency of the application on each system $i \in H$. Efficiency, here, can be the ratio of performance of the given application to either the best-observed performance, or the peak theoretical hardware performance [**?**].

Use of a high-level programming interface with a re-targetable back end that is standardized and supported by a number of both commercial and open-source initiatives has been found to be a critical element of portable application design [**?**,**?**]. OpenACC [**?**] was first developed to provide a high-level programming model for GPU programming, and now has been extended to multi-core machines. Conversely, OpenMP [**?**], once specific to CPU-based threading, has now been extended to the GPU. Both of these APIs offer compiler-directive-based interfaces with which to wrap sections of code for parallelization; they both appear in a similar format to the syntax used by OpenMP, which has now become familiar to many programmers of all levels. These two APIs are supported by a number of commercial hardware and compiler developers, and in addition, by the GNU project [**?**].

2.2 Molecular Dynamics

In molecular dynamics, a system, represented by atomistic units, is propagated in time based on some calculated forces using a numerical integration of Newton's equations of motion. The simulation cannot proceed with the next step until the previous one is completed; furthermore, a very small time-step is required to keep the simulation from sustaining unacceptable drifts in energy, as compared to experimental timescales that the simulation may be modeling [**?**]. Therefore, minimization of time per step is highly important. Several open-source, highly parallel classical MD programs exist that can scale to over thousands of nodes of a supercomputer and are heavily used internationally for molecular research. These programs are able to perform a time step in less than two milliseconds for systems of hundreds of thousands of atoms, or in seconds for systems of hundreds of millions of atoms [**?**,**?**,**?**,**?**,**?**,**?**].

The classical molecular dynamics algorithm involves three main components: the integration step, the calculation of bonded forces, of pairwise short-range non-bonded (SNF) forces, and the calculation of long-range forces. The integration step is generally the quickest part of the calculation, and as it has some memory-intensive aspects, is often calculated using the CPU, in implementations using heterogeneous architectures. The long-range forces calculation, in most implementations, involves an Ewald-sum, and requires Fourier transforms. The SNFs consist of the Lennard-Jones interaction, and short-range electrostatic forces. The Lennard-Jones interaction is an empirical function created to approximate the dispersive, or van der Waals forces, which in reality are purely

quantum effects. The functional forms for these two additive forces are:

$$F_{LJ}(\mathbf{r_{ij}}) = \Big[12\big(\frac{\sigma_{ij}^{12}}{r_{ij}^{13}}\big) - 6\big(\frac{\sigma_{ij}^{6}}{r_{ij}^{7}}\big)\Big]\frac{\mathbf{r}_{ij}}{r_{ij}}, \tag{3}$$

$$F_{C}(\mathbf{r_{ij}}) = \frac{1}{4\pi\epsilon_0}\frac{q_i q_j}{r_{ij}^2}\frac{\mathbf{r}_{ij}}{r_{ij}}. \tag{4}$$

Here $F_{LJ}(\mathbf{r_{ij}})$ is the Lennard-Jones force on atom i due to atom j, with $\mathbf{r_{ij}}$ being the vector connecting atom i to atom j. σ is a parameter that depends on the atom type of both interacting atoms, and $F_C(\mathbf{r_{ij}})$ is the analogous Coulomb force, with q_n being the point-charge value assigned to atom n, and ϵ_0 the permittivity of free space; both are functions of the inter-atomic distance [**?**,**?**].

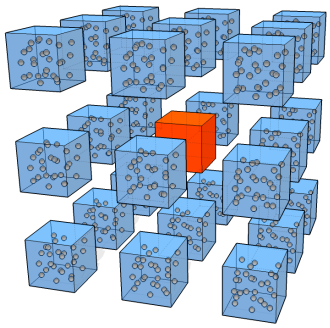

Fig. 1: Schematic of the interaction neighbors for cell-cell interactions involved in the spatial decomposition in the molecular dynamics algorithm. The central box (orange), interacts with itself, and with its 26 immediate neighbors, creating a total of 27 interactions for each cell in the grid, if in a periodic system, or a range of interactions from 8-27 if in a non-periodic system. Boxes are exploded outward for visualization purposes, but sides are touching in the actual grid.

The Lennard-Jones and short-range electrostatic forces rapidly decay to zero outside of a radius of about 10-14 angstroms. This creates an excellent mechanism for reducing the total calculation by imposing a distance-based radial cutoff on each atom, outside of which no interactions are considered. Algorithmically, the SNF calculation usually consists of a spatial decomposition, or domain decomposition, of the system, into a three-dimensional grid of cells, followed by a binning of the atoms into their associated cells with some sort of sorting procedure. After this the pairwise forces on each atom can be calculated and summed [**?**,**?**]. These forces, as can be seen from their equations, depend on the pairwise distances between an atom and all other atoms within the radial cut-off. If the spatial decomposition is performed so that the cells' dimensions are close to the LJ cut-off distance, then only the interacting cell-cell pairs need to be searched for interacting atoms, for each cell [**?**]. In the periodic regime, all cells have 26 neighbors, and distances of all atoms within the central cell must be calculated

as well, resulting in 27 cell-cell interactions that must be calculated for each cell in the grid of the domain decomposition. Figure 1 shows a central cell and its interacting cell neighbors. Figure 2 shows a sparsity plot of the distance matrix for all cell-cell interactions in the system, with those having distances greater than the cut-off set to zero and colored white, and interacting cells colored blue. As can be seen, the cut-off creates a banded structure to the matrix, and reduces the number of cell-cell calculations by about 90%.

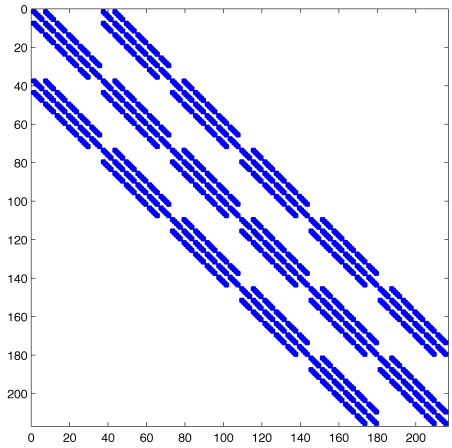

Fig. 2: Sparsity plot of distance matrix of all cell-cell distances, with all distances outside of radial cut-off of 10 angstroms set to zeros (and colored white), for a solvated system of 30,000 atoms (small protein), and all distances within the cut-off in blue. The number of cells in each direction is 6, resulting in a total of 46,656 cell-cell distances. 4096 of these are actually calculated in a non-periodic MD simulation due to the cut-off scheme.

3 Portability goals: timings and architectures

HPC MD developers have continuously pushed for increasingly shorter per-time-step execution rates. Currently, GROMACS [**?**] and NAMD [**?**] exhibit highly competitive timings per time-step. For 21 M atoms, NAMD attained about 5 ms per time-step and for a 224 M atom system, about 40 ms per time-step using using 4096 nodes, according to published benchmarks [**?**,**?**]. In 2015 GROMACS reported a sub-millisecond time-step for a 80,000 atom system using only 32 nodes, with Intel E5-2680v2 processors with 20 CPU cores and 2 K20X NVIDIA GPUs, and 1.7 ms per step for a 2 M atom system using 512 nodes of the same processor type but without GPU [**?**]; thus performance on a multi-core machine can actually exceed that of a GPU-enabled supercomputer for MD. Using GROMACS 5.1.3 on OLCF Titan, a Cray XK7 with AMD Interlagos CPUs and one NVIDIA K20X GPU per node we obtained a 1.2 ms time-step for a 1.1 M atom system using 1024 nodes. This level of performance has been

attained and is expected on many-core, multi-core, *and* GPU-containing HPC systems.

We test some key kernels from a MD calculation to see if parallelization with OpenACC can be performed while remaining under 7 ms/time-step for a system under 20 M atoms, or 55 ms/time-step for a system of about 220 M atoms, after domain decomposition. On a single node, the total times for the kernels must be well below these numbers while at the same time the job size on the node must be large enough so that the total domain decomposition would not use more that about 2000 nodes for a smaller system, and 4000 nodes for a larger system. Common domain decomposition for MD programs involves computing the SNFs acting on about 15 K atoms on a single node. For around 15 K atoms, there are about 3,000 cell-cell interactions, so what we aim for is a total kernel time under 6 ms for about 3,000 cell-cell interactions, or 50 ms for about 12,000 cell-cell interactions, which leaves time for communication and other less time-consuming portions of the calculation, and corresponds to an 80% efficiency score compared to NAMD, and if maintained for all architectures tested, would result in a minimum of 80% performance portability score in (2). We test whether this performance can be maintained using the same source code, on nodes with multi-core CPUs and on heterogeneous nodes containing a GPU.

4 Designing The Kernels

4.1 The programming model and its portable subset

C enjoys native support on a variety of machines and is familiar to most programmers, furthermore, C++ functionality has been added for some compilers [**?**], but can be problematic [**?**]. We try to use only the portable subset of C and OpenACC. For C, this means avoiding structures and classes, and programming elements that are difficult to parallelize with directives. While OpenMP provides SIMD constructs that enable machine-specific elements to be added to a parallel region, OpenACC does not contain syntax for such explicit targeting [**?**,**?**,**?**,**?**]. Additionally, it has been found that OpenACC threading on the CPU can be poor if the optimal organization of a particular kernel is not used, and that this re-organization for the CPU can decrease performance on the GPU [**?**]. We would like for parallel regions to not have to be rearranged with different constructs to obtain adequate performance on different architectures. We tried to use the simplest layouts as an initial test, with the hope that the design and modularity of the application could provide a large portion of the parallel performance gain. Although the format of directive-based parallelization with OpenMP and OpenACC initially seem similar, unfortunately the two APIs differ enough in how they must be used to obtain adequate parallelization, that they cannot be exchanged using simple macros. In many cases, different sections of nested loops

require different arrangement of parallel clauses, and in some cases, the code region must be re-arranged when switching between APIs [**?**,**?**,**?**]. There are initiatives that are aimed at performing an automated translation from one to the other; this is a positive development as currently only several compilers support each interface [**?**]. To facilitate such a translation it will also be advantageous to use the simplest syntax for each parallel region.

4.2 Modular format and kernels

To create a modular format that can facilitate portability, deconstruction the MD algorithm into its subtasks was performed. To simplify the algorithm we avoid the use of non-rectangular cells for the domain decomposition. Several highly optimized algorithms have been published that focused on the use of cells of varying complexity, [**?**,**?**]; these types of algorithms require more time to code, understand, and test, and thus are not a practical choice for a dedicated portability effort. We chose a rectangular grid, and once the grid is created, the location of each atom can very easily be calculated using a reduction over its three position coordinates. A one-digit address can be uniquely determined.

We focus on several computational modules that are repeated every time step. The creation of the cell-grid, based on the minimum and maximum values of the atomic positions along each dimension, and the cut-off radius given, is an example of a procedure that only needs to be calculated one time for a constant-volume simulation. This is also true for the creation of the interaction list for the cell-cell interactions. The first module that is repeated is "atom-binning." Involved in this task is the assignment of each atom to its corresponding cell, referred to here as the "cell-assign" procedure. Additional steps involve counting the number of atoms in each cell, and the filling of a data structure representing each cell with the appropriate atoms' coordinates, either with a gathering of all atoms belonging to, or with a halo-exchange type operation after an initial sorting. Cell-assignment is a completely parallel task that is trivial to distribute and requires no redesign from an analogous serial algorithm. For the rest of the parallel binning algorithm, however, it is impossible to simply parallelize the counting and the permutation-array steps with a directive added to a serial implementation: the concept of counting and all-prefix-sums are dependent on the sequential programmatic progression. This is an excellent example of how the use of OpenACC in a naïve way to speed-up a serial algorithm can fail completely. An algorithm's serial version may require a complete restructuring in a parallel programming model.

Another module we tested is the squared pairwise-distance calculation. This module comprises a large portion of the force computation in MD, which is the largest bottleneck [**?**,**?**]. The decay of the force functions, however, makes the cut-off approximation both accurate and very computationally important; the cell-based spatial decomposition makes excellent use of this [**?**]. The large, cell-

cell pairwise distances calculation (the pairwise distance of each atom in each cell with all other atoms in interacting cells) has a complexity of $O(N)$, where N is the total number of atoms being modeled, however the prefactor is very large. To obtain the pairwise distances, an element-wise square root must be applied.

The pairwise distance calculation is also important for numerous applications in statistics and data science. A pairwise-distance calculation over a large number of multi-dimensional observations is central to clustering algorithms such as k-means and some kernel methods [**?**,**?**,**?**,**?**,**?**]. Therefore an analysis of the potential for performance portability of a massively parallel distance matrix calculator is of interest in its own right [**?**,**?**].

Many MD programs also employ an atomic neighbor-list, which not updated every time step under the assumption that the atoms will not move considerably each step. This reduces the number of times the pairwise distances within the cell interactions are calculated, thus it reduces the prefactor in the $O(N)$ complexity. However, this procedure incurs some launch overhead, memory access, and communication costs: potential inefficiency of many small data-structure-accessing steps, increased bookkeeping requirements in the code, and the requirement to "batch" the calculations by hand on the GPU for efficiency, lead to increases in code complexity and thus potential error-generation, and decreases in portability. We did not address the neighbor-list calculation in this study.

5 Binning module (Neighbor-list updates): bin-assign, bin-count, and bin sorting

5.1 Bin-assign, Bin-count

Listing 1.1 shows a serial version of the bin-assign and bin-counting procedures. The most efficient method, in serial, is to use a one-dimensional array of atoms' bin IDs, and an accompanying array that keeps track of how many atoms in each bin. In a serial implementation, this bin-count can be accomplished in the same for-loop as the bin-assign.

```
1
2 /* after determining the number of bins based on the total system size
      and the cutoff, allocate the array that keeps track of how many
      atoms are in each bin:          */
3   bin_count=(int *calloc(numbins*sizeof(int));
4  // binning procedure:
5   for (b = 0; b < num_atoms; b++) {
6 // read each atom's 3 coordinates and calculate the value of the 3-digit
7 // address:
8     temp[0] = floor((coords[b][0]/range[0]-kbinx)*num_divx);
9     temp[1] = floor((coords[b][1]/range[1]-kbiny)*num_divy);
10    temp[2] = floor((coords[b][2]/range[2]-kbinz)*num_divz);
11 // find the 1-digit address of the atom
12    n= num_divy*num_divz*(temp[0])+num_divz*(temp[1])+(temp[2]);
```

```
// enter the 1-digit address into that atom's index in the bin_ids array:
    bin_ids[b] = n;
// update the count in that bin's index in the bin_count array:
    bin_count[n]=bin_count[n]+1;
  }
/* The variable bin_ids is a one-dimensional array the length of the
    number the total number of atoms. Each element of bins contains the
    single-integer bin ID of the atom with corresponding array index,
    and the variable coords is a two-dimensional array allocated at the
    initialization of the program, containing the x, y, and z components
     of each atom's coordinates. The variable bin_count is a tally of
    the number of elements in each bin. */
```

Listing 1.1: Code snippet of a serial version of bin-assign/bin-count

For a parallel version, "counting" is ill-defined, and this seemingly trivial computation in serial, becomes a more difficult task in parallel. The serial version of the gathering step is also relatively trivial. Listing 1.2 shows a version of this type of nested solution in a serial implementation.

```
count = 0;
   for(b=0; b<numbins; b++) {
      for(c=0; c<num_atoms; c++) {
         if(bin_ids[c]==b){
         gather_array[count]=c;
         count++;
         }
      }
   }
```

Listing 1.2: Code snippet of a serial version of the gathering array generation

Since the bin-assign procedure is independent for each atom, it is easily parallelizable. The requirements simply involve using a single OpenACC directive to parallelize the serial version. Listing 1.3 shows the implementation of its parallelization using an OpenACC parallel loop pragma. It is further possible to potentially optimize this section using different OpenACC options, however, with just this simple addition, using the PGI compiler, OpenACC generates an implicit `copy_out` of `bin_ids`, an implicit `copy_in` of `coords` and `range`. For 500,000 atoms, this section of the binning algorithm required 115 microseconds (0.115 ms), and for 30,000 atoms, 11 microseconds (0.011 ms), using one node of Titan with the GPU. Although the use of gangs, workers, and other data distribution keywords provided by OpenACC are defined for particular divisions of tasks, the actual performance of these various methods to create compiler-written code for a particular HPC architecture is highly system dependent. We found for the above kernel, that these additional constructs did not improve performance. The most general pragma, the "kernels" directive, allows the API to determine what regions of the section can be parallelized, and to distribute these regions appropriately. A less general option is the parallel "loop" region, which specifically tells the compiler to parallelize the loop.

```
#pragma acc parallel loop private(temp)
for (b = 0; b < num_atoms; b++) {
// read each atom's 3 coordinates and calculate the value of the 3-digit
```

```
// address:
 temp[0] = floor((coords[b][0]/range[0]-kbinx)*num_divx);
 temp[1] = floor((coords[b][1]/range[1]-kbiny)*num_divy);
 temp[2] = floor((coords[b][2]/range[2]-kbinz)*num_divz);
// find the 1-digit address of the atom
 bin_ids[b]= num_divy*num_divz*(temp[0])+num_divz*(temp[1])+(temp[2]);
}
```

Listing 1.3: Code snippet of a simple OpenAcc parallelization of bin-assign

Manual task division for bin-assign together with OpenACC pragmas We also tested how some amount of manual splitting of the bin-assignment tasks would affect the speed-up. We separated the atoms into 5 evenly distributed blocks, and added OpenACC loops around both the blocks, and the inner bin-assign. Interestingly, this resulted in a 3-5 $\times$ speed-up, depending on the number of atoms. However, the speed-up may be system- and data-size- dependent, and it may be a difficult task to optimize this manual splitting by future users of the application.

5.2 Parallel algorithm design for bin count and gather

The bin count and gather operations are classical examples of more difficult problems in parallel computing. For this reason, for a portable application, these modules may be better handled with optimized routines from libraries rather than OpenACC. Furthermore, the optimal programmatic solution may vary greatly between architectures. Details are provided below.

Bin count Parallelization of the bin-count procedure can be approached in several ways. This is ultimately a histogramming task. One can use an atomic-add for the bin-count array variable, which can be kept in a shared location in memory. These types of operations are supported by OpenACC's more advanced directive options. Alternately, one can create a type of merge-count, so that the bin-ID array is split into subarrays, each is counted in serial by parallel gangs, and the results are merged. It is also possible that for various architectures, there will be a different optimal solution for this step. Thus this is an example of a region of code that may require encapsulation and increased documentation, as well as several kernels to be used for specific systems, or the potential for exchanging with an optimized library. A high performing histogram routine, for instance, could be employed in this section [**?**], as can a parallel prefix-sum routine [**?**].

Gather In order to gather all atomic coordinates belonging to a cell (bin) into a single data structure for passing to the pairwise distance calculation, the use of masks, or an efficient parallel scan algorithm can be used. This process involves a (fuzzy) sorting and somewhat complicated data movement patterns. The optimal solution can require a significant amount of effort and may vary greatly based

on the architecture targeted, and thus is the type of procedure that could also be replaced with a call to hardware-specific libraries, that would each provide an optimized solution for a specific architecture. One possibility is to exploit sorting procedures provided by HPC libraries such as Thrust [**?**]. This again is a region that must be encapsulated and well-documented, because it may involve machine-specific solutions [**?**].

```
for(b=0; b<num_batch; b++){
  /* rows=dim, columns=obs. */
  for (i = 0; i < num_coords_n; i++) {
    for (j = 0; j < num_coords_m; j++) {
      for (k = 0; k < 3; k++)
      {
        temp =  batchA[b][k + 3 * i] - batchB[b][k + 3 * j];
        y[k] = temp * temp;
      }
      temp = y[0];
      for (k = 0; k < 2; k++)
      {
        temp += y[k + 1];
      }
     batchC[b][i + num_coords_n * j] = temp;
    }
  }
  }
```

Listing 1.4: Code snippet of serial version of the squared pairwise distance calculation

After the initial gathering of atoms into their respective cell arrays, future gathering operations can be accomplished with a data exchange routine common in the halo-exchange algorithms used in stencil computations [**?**,**?**]. This takes advantage of the fact that atoms do not move large amounts over short time periods, and thus many atoms may not change cells for many time steps. Therefore the number of exchangers will be small. However, this approach involves communication expense and more complicated data movement patterns. Alternately, larger groups of atoms in neighboring cells can again be sorted by bin ID using a fast sorting algorithm that is most efficient on heavily presorted data [**?**,**?**,**?**]. For smaller systems where data is located on a single node, it may be faster to resort all atoms than to perform numerous communication and data exchange operations. Ultimately, the optimal choice of algorithm may also be hardware specific, and thus it is possible that the most performance portable solution for this module is a call to an HPC library.

6 The squared pairwise distance calculation: performance, portability, and effort

In this section we examine the performance portability of an OpenACC implementation of the calculation of the squared pairwise distance matrix for all atoms in sets of two interacting cells. This calculation does not suffer from the types of algorithmic complexities that the bin count and gather modules do; it is a more easily parallelizable routine much like the bin assign module. For this module,

in addition to the use of OpenACC for parallelization on the GPU and on the CPU, we also created two alternate implementations, one using a CUDA kernel, to compare performance of the directive-based implementation, and one completely using routines from newly emerging batched versions of accelerator-based Basic Linear Algebra Subprograms (BLAS)[**?**] libraries.

The BLAS version is a pedagogical example of a solution that is not only portable, but actually requires the least amount of parallel programming experience: it allows the user to perform the calculation without any knowledge of accelerator programming or even any experience with compiler directives. Thus the effort and skill required to port this version would be minimal. While batched versions of BLAS standard routines are not technically part of the standard, there is a growing need for these types of routines and they are available in many scientific libraries.

Listing 1.4 shows the serial version of such a calculation. The variables `batchA` and `batchB` are batched collections of atoms in interacting cells, `batchC` is an array of respective distance matrices for each cell pair from in `batchA` and `batchB`, and `num_cells` is the number of cells in each batch. There is a loop over the three dimensions in order to provide generality: for use in data analysis the dimension may be very large and parallelization of the loop may be necessary.

We tested single node, single GPU and CPU-only implementations, implementing parallelization with OpenACC, CUDA and cuBLAS using OLCF Titan, a Cray XK7 with 16-core AMD Opteron "Interlagos" CPUs and NVIDIA Kepler (K20X) GPUS, and OLCF Summit, a system containing 42 IBM POWER9 CPUs and 6 NVIDIA Volta (V100) GPUs per node, with 4 SMT hardware threads per CPU core [**?**].

```
#pragma acc data copyin(A[0:3*num_batch*num_coords_n]),copyin(B[0:3*
    num_batch*num_coords_m])

pairwise_batched(A, B, C, num_coords_n, num_coords_m, num_batch);

void pairwise_batched(double A[], double B[], double C[], int ldX, int
    ldY, int num_batch){
#pragma acc data present(A), present(B), present(C)
#pragma acc kernels
#pragma acc loop independent
  for (int b=0; b<num_batch; b++)
  {
    double y[3];
    double temp;
    /* rows=dim, columns=obs. */
#pragma acc loop independent
    for (int i = 0; i < ldX; i++) {
#pragma acc loop independent
      for (int j = 0; j < ldY; j++) {
#pragma acc loop seq
        for (int k = 0; k < 3; k++)
        {
          double temp =  A[b*3*ldX + k + 3 * i] - B[b*3*ldY + k + 3 * j
    ];
          y[k] = temp * temp;
        }
        temp = y[0];
```

```
#pragma acc loop seq
        for (int k = 0; k < 2; k++)
        {
          temp += y[k + 1];
        }
        C[b*ldX*ldY + i + ldX * j] = temp;
      }
    }
  }
}
```

Listing 1.5: Code snippet of a simple OpenACC parallelization of the squared pairwise distance calculation

6.1 Use of OpenACC for the squared distance calculation: GPU

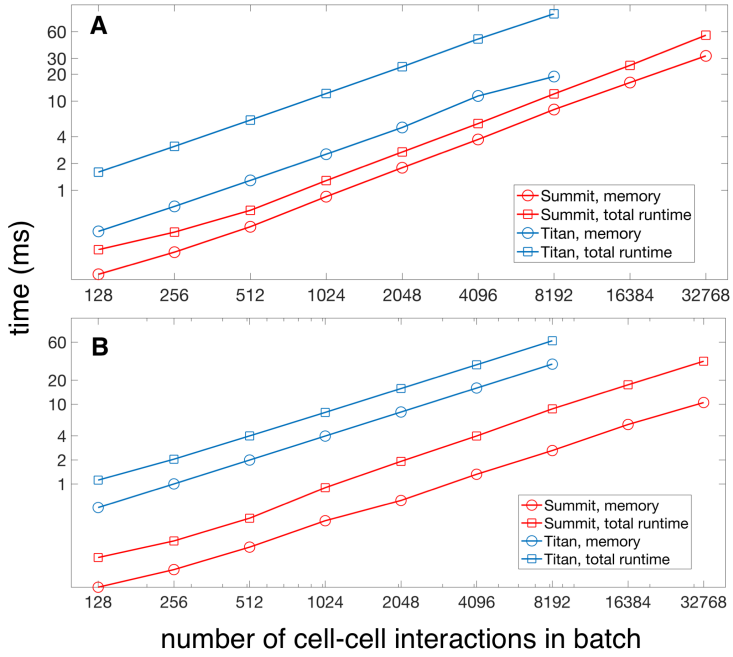

Fig. 3: Comparison of performance (time in ms), for CUDA and OpenACC versions of the GPU-based all-pairwise squared distances calculation on OLCF Titan (K20X) and Summit (V100), over increasing batch sizes. A: Using OpenACC distance kernel. B: Using CUDA distance kernel.

The GPU-based OpenACC version was created by adding a `data` region, an `acc kernels` region, and three `acc loop independent` regions around the serial pairwise distance function shown in Listing 1.4. The two dimensional array was also flattened. Listing 1.5 shows this implementation. While there may be further work to be done in determining the optimal OpenACC clauses to use for this calculation, for the scheme shown in Listing 1.5, results were surprisingly good. On both Titan and Summit, a reasonable number of batches could be processed in under 10 ms, and on Summit, all cell-cell interactions for a system the size of a small protein (about 6000 batches) could be processed on a single

GPU in under 10 ms. Figure 3-A shows timings for increasing batch sizes using one node and one GPU of each machine. These results are within the acceptable range we determined for an MD step, although for smaller systems the upper limit on the time-per-step greatly constrains the amount of batches that can be offloaded to a single node, resulting in the use of only a small percent of the peak FLOPs available on the GPU. With no such constraint, it would be possible to perform significantly more pairwise distance calculations per node in a relatively rapid amount of time based on how much the two tested GPUs' global memories can hold.

For larger systems, it may be advantageous to use more batches per node, and maximize the percentage of peak FLOPs used, as the amount of allowed time per time step for larger systems by current standard is higher. For a system of about 80,000 atoms, as in the GROMACS benchmark discussed in section 3, using about 1024 batches per node, the distance calculation can be completed for all interacting atoms in under 10 ms using less than 8 nodes. Using 32 nodes, as used in the benchmark, this calculation can be completed in under 0.5 ms on Summit. Of course there are some additional calculations to be performed, i.e. the square root and the application of the force functions to the distances, to complete the SNF routine, however, these involve fewer FLOPs and no further memory transfers. The possibility of using OpenACC on GPUs within a performance-portable HPC MD application is not excluded by these initial benchmarks.

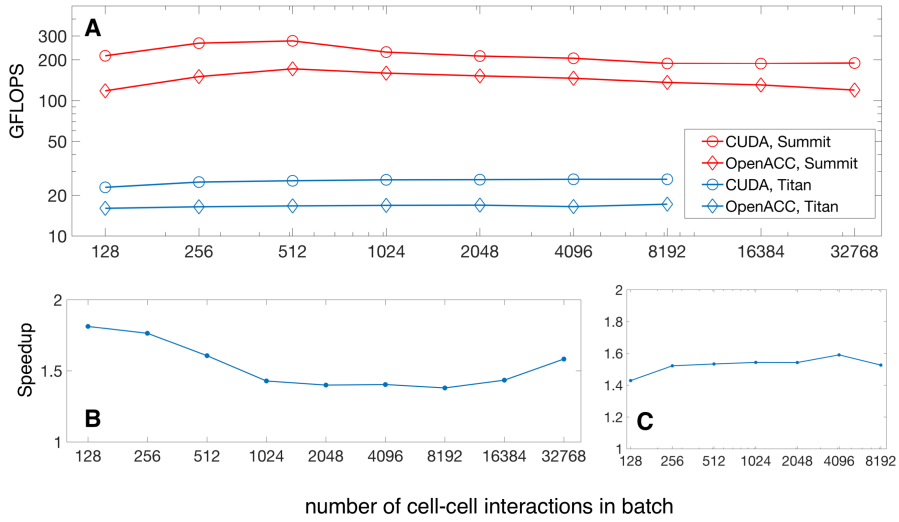

Fig. 4: A. Comparison of performance by GFLOPS for CUDA and OpenACC versions of the GPU-based all-pairwise squared distances calculation on OLCF Titan (K20X) and Summit (V100), over increasing batch size. B: Speedup (×) of CUDA kernel over OpenACC kernel distance kernel on Summit, for total runtime and memory transfer time. C: Speedup of CUDA kernel over OpenACC kernel distance kernel on Titan.

6.2 Comparison to CUDA Kernel

Figure 3-B shows timings for the CUDA implementation of this calculation on Titan and Summit, and Figure 5 shows memory transfer times and speedup over OpenACC version. OpenACC is slower, but less than 2 ×. Performance is still within the acceptable range, and excellent on Summit, using OpenACC.

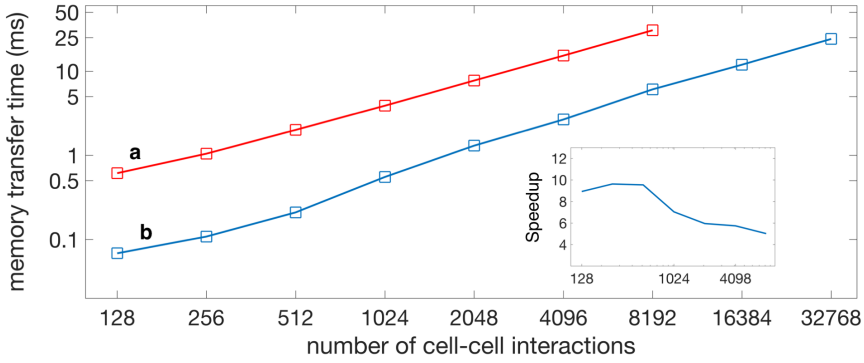

Fig. 5: Comparison of memory transfer time for CUDA versions (and BLAS version) of the GPU-based all-pairwise squared distances calculation on a) OLCF Titan (K20X) and b) Summit (V100), for different batch sizes. Inset: Speedup (×) for Summit versus Titan

6.3 OpenACC on the CPU

We tested the parallelization of the kernel using OpenACC for CPU-based threading with the `ta=multicore` compiler flag, using the identical code. There are some algorithms that can be performed faster on the CPU, with OpenMP threading, than on the GPU with OpenACC or CUDA [**?**,**?**,**?**]. To get maximum performance on the CPU you must use threading, alignment, and vectorization [**?**], OpenACC has no functionality for intrinsic-function level specification. It also has no option for treating thread affinity. OpenACC seems to be less useful for creating truly performant CPU-based kernels than GPU version, for kernels like the distance calculation. Memory transfer time was sub-microsecond, and is not reported. Figure 6 shows kernel runtimes for varying batch sizes and scaling data on Summit and Titan. For smaller batch sizes, times can be within an acceptable range for Summit, but not Titan. Furthermore, the small batch size limit reduces the number of cell-cell interactions to those in an equivalent system of about 20,000 atoms. Therefore, we see that performance of OpenACC, even on new supercomputer cores, is barely within the lowest limit for performance portability.

6.4 Comparison to a purely BLAS-based algorithm: Lowest programming knowledge required

A well-known algorithm for the pairwise distance calculation can be implemented completely with subroutines from BLAS libraries. Surprisingly, although this algorithm involves slightly more total flops and a significant amount of memory operations than the direct method, it has been considered in the past as a fast way to implement the distance calculation on the CPU, if using threaded scientific libraries like Intel's MKL [**?**]. Matrix operations such as matrix-matrix multiplication are included in most high-performing scientific libraries provided by system manufacturers, and have also become benchmarks for measuring the performance of these systems. Thus they are competitively implemented in a highly optimized manner. The algorithm for the BLAS-based distance matrix calculation is shown in Algorithm 1.

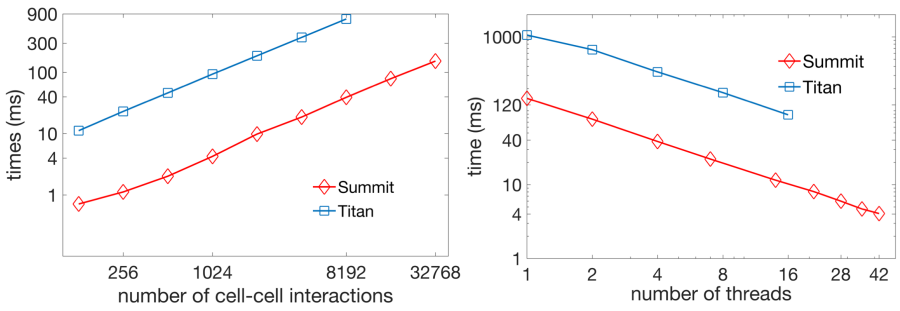

Fig. 6: Performance and scaling (time in ms) of OpenACC threading on the CPU, Summit and Titan, for all-pairwise squared distances calculation. Left: Performance. Right: Scaling plot for 1024 cell-cell interactions (batch number 1024).

The BLAS-based algorithm would probably be used on a single large matrix, in data analysis. Here we explored the potential of using such an algorithm on the GPU with a more recently developed massively-parallel extension of matrix-matrix multiplication (MM), the batched MM routines, to compare performance to our other two versions. While as-yet not a standard BLAS routine, a batched version of MM for many small matrices exits in both the NVIDIA-provided cuBLAS library [**?**], and Intel MKL for multicore architectures and the KNL [**?**]. Furthermore, an version of a batched MM routine is provided by Magma, [**?**] an open-source effort that creates accelerated BLAS routines for a number of architectures. There is a growing possibility that that batched BLAS routines will enter into the standard, as they arise naturally when large problems are decomposed on parallel platforms [**?**].

The BLAS method creates a floor for the amount of programming skill and effort required for an accelerated squared pairwise distance calculation. With

this version programmer would not need to have any experience in any programming languages other than C/C++ or FORTRAN, not even the use of compiler directives. However, this algorithm works best for large matrices that are closer-to-square in shape, unlike our coordinate arrays.

Algorithm 1 Pairwise squared distance calculation using matrix operations, adapted form Li et al., 2011 [**?**]

1: load matrices $\mathbf{A}$ and $\mathbf{B}$ and allocate memory for matrix $\mathbf{C}$
2: $\mathbf{A}$ has dimension N by 3, $\mathbf{B}$ has dimension M by 3, and $\mathbf{C}$ has dimension N by M
3: note: $(\cdot)$ denotes elementwise multiplication
4: $\mathbf{v}_1 = (\mathbf{A} \cdot \mathbf{A})[1, 1, 1]^{\mathrm{T}}$
5: $\mathbf{v}_2 = (\mathbf{B} \cdot \mathbf{B})[1, 1, 1]^{\mathrm{T}}$
6: $\mathbf{P}_1 = [\mathbf{v}_1, \mathbf{v}_1, ..., \mathbf{v}_1]$ (dimension N by M)
7: $\mathbf{P}_2 = [\mathbf{v}_2, \mathbf{v}_2, ..., \mathbf{v}_2]^{\mathrm{T}}$ (dimension N by M)
8: $\mathbf{P}_3 = \mathbf{A}\mathbf{B}^T$(dimension N by M)
9: $\mathbf{D}^2 = (\mathbf{P}_1 + \mathbf{P}_2 - 2\mathbf{P}_3)$, where $\mathbf{D}^2$ is the matrix of squared distances
10: pairwise distance matrix can be recovered from $\mathbf{D}^2$ by element-wise square-root

We implemented this algorithm using cuBLAS, the CUDA-based BLAS library provided by NVIDIA. Using matrices of size 200 by 3, we tested this implementation on a single GPU of Titan and Summit. Matrices $\mathbf{A}$ and $\mathbf{B}$ in our situation are the 3-D coordinates for atoms in two interacting cells. In order to perform lines 6 and 7 with BLAS routines, one can use the `dger` routine (outer product) of $\mathbf{v_1}$ or $\mathbf{v_2}$ with a vector of ones of length N. However, cuBLAS does not provide a batched version of `dger`, and thus we used batched MM again with input "matrices" $\mathbf{v_1}$ or $\mathbf{v_2}$ and a vector of ones. The element-wise multiplication is available in MKL as a Hadamard product, but not in cuBLAS, thus lines 4 and 5 were performed on the CPU and not included in timings. Because of this, we found that on the GPU, this algorithm cannot be performed completely with cuBLAS functions. Even without these first two components of the calculation, the performance of this method on the GPU compared to that of OpenACC or CUDA-C is much lower. Figures 7 and 8 show timings and comparison to the CUDA kernel. This (partial) version's performance is quite poor, but better than the OpenACC-threaded CPU version. Despite optimized BLAS routines on the GPU provided by NVIDIA, the memory operations swamp the performance in comparison to the CUDA-C and the GPU-based OpenACC versions.

7 Programming effort

It is difficult to measure worker effort, especially when skill levels of workers may differ. Some papers report that CUDA requires more effort than OpenACC, even for workers familiar with both APIs [**?**,**?**,**?**]. However, different compilers each

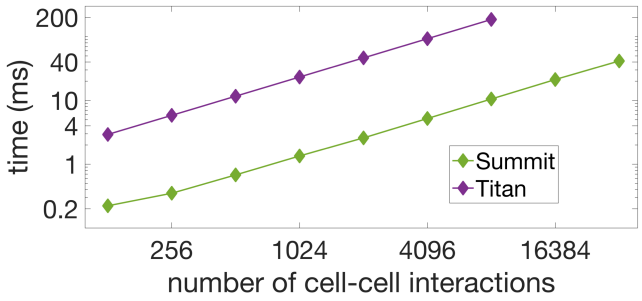

Fig. 7: Comparison of performance of BLAS version, all-pairwise squared distances calculation, on the GPU, using OLCF Titan (K20X) versus Summit (V100).

may implement a particular directive instruction differently, and variable performance may require alternate constructs to be used to parallelize a particular section of code, leading to some level of trial and error in each port. This lack of a defined outcome increases potentials for performance portability, as there are more possibilities that optimal performance will be obtained by using different constructs and different compilers, but can be frustrating for the user, and increased experience may not increase the ease of this process. Therefore, we cannot say that the use of OpenACC requires significantly less total worker effort than use of CUDA-C for small kernels. On the other hand, the amount effort required for OpenACC parallelization is not large, and the result is far more portable than CUDA-C after the first implementation has been created. It is also possible that the effort required for OpenACC is less than for some alternative portable solutions, such as OpenCL [**?**]. The use of the cuBLAS-batched indeed required the minimum amount of programming skill, however, creating the kernel involved more programming steps than the addition of a directive to a serial kernel, and more testing to make sure the result was correct. On the other hand, after the initial implementation is created, it should be able to be used without any changes except for linking to a different library and any small changes to the call syntax.

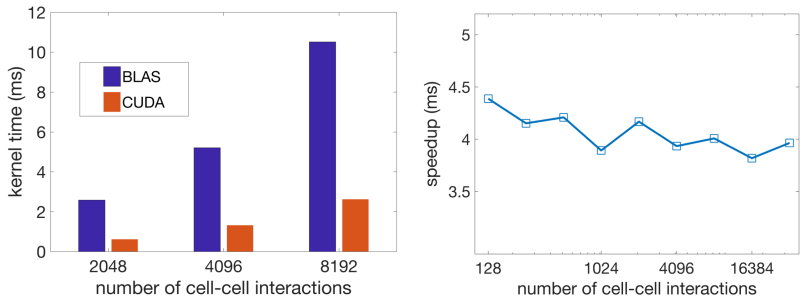

Fig. 8: Left: Comparison of performance (time, ms) of BLAS versus CUDA version of all-pairwise squared distances calculation, using one GPU on Summit. Right: Speedup (×) of CUDA version on Summit vs. cuBLAS-batched version on Summit.

8 Conclusions

We have found that portable kernels that remain within an acceptable performance range can be created for calculations representing bottleneck regions in MD. Using OpenACC, we found that while performance on the GPU was closer to the performance of CUDA kernels, on the CPU, performance of threaded kernels was much lower, and on older CPUs such as the AMD Bulldozers, would not provide acceptable performance. However, on the Power 9 processors, CPU performance remained within the low range of acceptability for smaller job sizes. Future work can compare the performance of these kernels when using OpenMP both on the CPU and the GPU. It is possible that the need for some amount SIMD-level instructions could be required for better performance on the CPU, and can also be tested in future work with OpenMP SIMD constructs.

Testing key kernels in scientific applications in this way creates examples of directive-based parallelization that include memory-limited calculations and difficult-to-parallelize algorithms, and expose routines that may perform in a less-than-efficient way. These examples, in turn, give the API developers test problems that may be outside of their usual testing routines, and thus help to maintain the cycle of collaboration between computational scientists and API developers that is seen as a requirement for the creation of portable, high-level interfaces for applications.

Challenges presented by the designing of HPC-portable applications using compiler directives include difficulties in the creation of parallel versions from serial routines, and can reveal the need for the use of high-performance libraries created for each particular architecture by specialists for certain encapsulated sections, instead of using the directives in those regions. It is possible that through the use of carefully designed modules and functions, together with directive-based programming models such as OpenACC, acceptable performance for some tasks can be achieved relatively easily. This can allow for a unified, performance portable interface for applications.

A Artifact Description Appendix: Using Compiler Directives for Performance Portability in Scientific Computing: Kernels from Molecular Simulation

A.1 Abstract

This appendix details the run environments, compilers used, and compile line arguments for the four tested methods details in the text. Note that hardware access is limited to OLCF users.

A.2 Description

Check-list (artifact meta information)

- **Algorithm: Select kernels used in molecular dynamics**
- **Compilation: See compliers and commands below**
- **Binary: C++/CUDA or C++/OpenACC**
- **Run-time environment: Modules displayed below**
- **Hardware: OLCF Titan and Summit as described in main text**
- **Run-time state: Summit used SMT=1 for CPU threading. Run commands below**
- **Execution: Run commands below, BLAS routines were called using standard calls to the cuBLAS library**
- **Publicly available?: All kernels are provided in the text and appendix**

All kernels used are listed in the main text, except the CUDA kernel. This is provided below:

```
template<typename T, int BS>
__global__ void
distance_kernel (T** A_data, T** B_data, T** C_data, int lda)
{
  int row_stride = lda;
  int row = blockIdx.x*BS+threadIdx.x;
  int col = blockIdx.y*BS+threadIdx.y;
  __shared__ T *A, *B, *C;
  if(threadIdx.x+threadIdx.y==0)
  {
    A = A_data[blockIdx.z];
    B = B_data[blockIdx.z];
    C = C_data[blockIdx.z];
  }
  __syncthreads();
  if ((row < lda) && (col < lda))
  {
    T elementSum = (T)0.0;
#pragma unroll
    for(int i=0; i<3; i++)
    {
      T diff = A[i*row_stride+row] - B[i*row_stride+col];
      elementSum += diff*diff;
    }
    C[col * row_stride + row] = elementSum;
  }
}
void
cuda_distance(double** A_data, double** B_data, double** C_data,
              int lda, int numBatches)
{
  const int BS = 16;
  int NB = (lda+BS-1)/BS;
  dim3 dimBlock(BS,BS);
  dim3 dimGrid(NB,NB,numBatches);
  distance_kernel<double,BS><<<dimGrid,dimBlock>>>
  ((double**)A_data, (double**)B_data, (double**)C_data, lda);
}
```

Listing 1.6: CUDA version of batched pairwise distance calculation

Software dependencies Below are the modules, compilers, and run commands used.

```
CUDA/BLAS ON SUMMIT, MODULES:
  gcc/5.4.0  cuda/9.1.85
CUDA/BLAS TITAN MODULES:
 gcc/6.3.0 cudatoolkit/9.1.85_3.10-1.0502.df1cc54.3.1

Compiler calls
CPP = g++
CPPFLAGS = -Wall -O3
NVCC = nvcc
NVCCFLAGS = -arch=sm_35 -Drestrict=__restrict__ -DNO_CUDA_MAIN -O3

OPENACC, GPU VERSION:
COMPILER = pgc++
COMP_FLAGS = -acc -ta=nvidia:cc35 -Minfo=accel -mp
#replace with Summit version nvidia:cc70

SUMMIT EXECUTION:
jsrun --rs_per_host ${NPPNODE} --nrs ${NP} -ELD_LIBRARY_PATH -c7 -g1 ./
    matrix_mul_batched > benchmark.SUMMIT.job
jsrun --rs_per_host ${NPPNODE} --nrs ${NP} -ELD_LIBRARY_PATH -c7 -g1 ./
    directDist3 > ACC_benchmark.SUMMIT.job
}

Compiler output, Titan:
make pairwise_batched.o
make[1]: Entering directory
pgc++ -acc -ta=nvidia:cc35 -Minfo=accel -mp -c pairwise_batched.cpp -o
    pairwise_batched.o
pairwise_batched(double *, double *, double *, int, int, int):
      4, Generating present(D[:],Y[:],X[:])
      9, Loop is parallelizable
         CUDA shared memory used for y
     15, Loop is parallelizable
     17, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
          9, #pragma acc loop gang /* blockIdx.z */
         15, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.
    x */
         17, #pragma acc loop gang /* blockIdx.y */
         19, #pragma acc loop seq
         26, #pragma acc loop seq
     19, Complex loop carried dependence of X->,y,Y-> prevents
    parallelization
make[1]: Leaving directory '/autofs/nccs-svm1_home1/andreas/sources/
    D2Calc/AT/ACC'
pgc++ -acc -ta=nvidia:cc35 -Minfo=accel -mp -o directDist3 directDist3.
    cpp pairwise_batched.o
directDist3.cpp:
main:
     43, Generating copy(D[:batch_count*40000])
     61, Generating copyin(T[:batch_count*600],U[:batch_count*600])
     73, Generating copyin(U[:batch_count*600],T[:batch_count*600])

Compiler output, Summit: Identical

OPENACC, CPU VERSION:
COMPILER = pgc++
COMP_FLAGS = -acc -ta=multicore -Minfo=accel -mp
```

Listing 1.7: Compile and run information