pointerchain: Supplementary Materials

Millad Ghane[a], Sunita Chandrasekaran[c], Margaret S. Cheung[a,b,d]

[a]*Department of Computer Science, University of Houston, TX*
[b]*Department of Physics, University of Houston, TX*
[c]*Department of Computer & Information Sciences (CIS), University of Delaware, DE*
[d]*Center for Theoretical Biological Physics (CTBP), Rice University, Houston, TX*

1. Supplementary Material A

Guidelines to applying optimizations: Below, we provide a detailed explanation of the ten optimization steps that we considered in our approach. Our proposed steps apply not only to CoMD but also serve as a guideline to accelerate other scientific applications. The following ten steps provide a roadmap for parallelization of scientific applications using the OpenACC programming model. *We used `pointerchain` in all of the following steps except the first one (**Step 1**) since it is based on UVM from NVIDIA.*

Step 1 - Kernel parallelization: After identifying the portions of the code as potential candidates for parallelization, we started to incrementally apply the OpenACC's `kernel` directive to parallelize the code. At this step, we are relying on the compiler's insight to parallelize the code and on UVM for on-demand data transfer between the host and the device. This is the rudimentary approach towards parallelizing fresh source code with OpenACC. UVM for NVIDIA GPUs is enabled by PGI compiler at the compile time with `-ta=tesla:managed` flag.

Annotating the computational kernels with `#pragma acc kernels` authorizes compilers to make decisions about parallelizing source code on behalf of the developers. In the case of force computation, however, the `kernels` construct was not able to parallelize the loop due to complex dependencies among variables inside the kernel. Thus, we used the `parallel` clause to express parallelism. This informs the compiler to parallelize only the outermost loop.

Step 2 - Efficient data transfer: Utilizing UVM leads to an arbitrary and untimely data transfer between the host and the device. Developers often prevent such on-demand transfers by explicitly specifying them in the code. That being said, we transferred the characteristics of atoms (position, velocity, force, momenta, and their potential energy) to the device only at the beginning of the program. The main copy of the data remains on the device and the data is only updated whenever the MPI framework transmits such data to and from other nodes.

From this step to the last one, we have applied the `pointerchain` directive to the code. If we disable UVM, we **have to** use our proposed directive; otherwise, the code does not work.

Step 3 - Manual parallelization: Relying on `#pragma acc kernels` does not necessarily guarantee a full utilization of computational resources on the device. Scientific developers, in most cases, are more knowledgeable about the problem, the code, and the data layout than the compilers. They are able to manually specify the parallelization opportunities by exploiting `#pragma acc loop` construct within the code. After decorating the kernel with the `#pragma acc parallel` clause, loops that benefit from parallelization are further decorated with the `loop` clause. In OpenACC, each loop is assigned to a different level of parallelism: `gang` and `vector`. For nested loops, the general approach is to parallelize the outermost loop with `gang` while the `vector` is applied to any loop under the `gang` level. This scheme is applicable to any *n*-level nested loop.

Unlike two-level nested loops in `AdvancePosition` and `AdvanceVelocity` kernels, the force computation in CoMD has four-level nested loops, which makes its parallelization a challenge. Due to dependency issues between the first and the second loop, instead of the second loop, the third loop was parallelized with `vector` parallelism in OpenACC. Exposing parallelism on the second loop caused a race condition leading to incorrect results. Despite having the opportunity to address this issue with atomic operations, we did not use them since experiments showed that atomic operations led to a two-fold performance loss.

Step 4 - Loop collapsing: When working with the nested loops within computational kernels, which in some cases exceeds four loops or more, we have the opportunity to collapse the tightly nested loops into one and to generate a flat loop with a wide iteration space using the `collapse` clause in OpenACC. The compiler has to know the effective loop size beforehand as the OpenACC specifications demands the loop size to be "computable and invariant in all the loops [1]." As a result, we made the higher bound of the inner loop independent of the outermost one. As shown in lines 2-6 of Listing 1, we took care of out-of-bound cases with an `if` statement. Listing 1 demonstrates how we performed a rectangular transformation of our nested loop. The index for the outermost loop determines the upper bound of the innermost loop. Based on our domain knowledge, however, the `s->boxes->nAtoms[iBox]` term will not exceed `MAXATOMS`. Thus, we replaced the upper bound of inner loop with `MAXATOMS` and governed `ii` with an `if` statement, as pre-

Email addresses: `mghane2@uh.edu` (Millad Ghane), `schandra@udel.edu` (Sunita Chandrasekaran), `mscheung@central.uh.edu` (Margaret S. Cheung)

sented in lines 9-15 of Listing 1. The code demonstrated in listing 1 was borrowed from CoMD.

Listing 1: Required loop transformation to make the search space from triangular (top) to rectangular (bottom)

```
1  // Original two for-loops
2  for (int iBox=0; iBox<nBoxes; iBox++)
3    for (int ii=0; ii<s->boxes->nAtoms[iBox]; ii
         ++)
4    {
5      //<some computations>
6    }
7
8  //Making the search space rectangular
9  for (int iBox=0; iBox<nBoxes; iBox++)
10   for (int ii=0; ii<MAXATOMS; ii++)
11   {
12     if(ii >= s->boxes->nAtoms[iBox])
13       continue;
14     //<some computations>
15   }
```

Step 5 - Improving data locality: Since most data accesses start from the global memory, maximizing the bandwidth of the global memory is the key to improving overall performance of applications running on GPUs. We have achieved this by minimizing the total number of memory transactions from global memory in GPU-based applications. Generally, the two major layouts for data representation are the Array of Structure (AoS) and the Structure of Array (SoA). Typically, the SoA layout exploits a full memory bandwidth since it causes coalesced accesses to the global memory on the device; however, the AoS layout approach results mostly in an interleaving access to the memory, which unequivocally degrades performance due to cache thrashing and unnecessary memory access.

In many cases, it is preferable to utilize the SoA layout in contrast to AoS ([2, 3]); however, Giles et al. [4] and Mudalige et al. [5] provided the opposite viewpoint. They argued that, in their case, the AoS data layout provided better performance for their library.

Our CoMD implementation, up to this point, follows a data layout similar to the AoS layout. In the current design, the components of force, location, and momenta in the 3D space are placed in consecutive locations of the global memory. Figure 1a shows this layout. Such a configuration is not performance friendly, especially in terms of NVIDIA GPU's architecture as a SIMT architecture.

A naïve way to improve this design is to make the memory layout cache-friendly. In order to do so, we added a dummy field and increased the size of structure from three elements to four to treat their misalignment. This is similar to the idea of *padding* a structure. This approach leads to an increase in data size by 33%. This optimization is designated with an "a" in figures.

Additionally, we improved the data locality of our implementation by temporarily storing the array elements in current working set to local variables. Local variables to a thread are usually reduced to registers on the device, which helps us drop global memory accesses. This optimization is designated with a "b" in figures.

And finally, we modified the layout of our data structures to improve its data locality and to make it more cache-friendly. Figure 1b depicts the modified data structure and its effect on the layout of data in memory. Each of the atom's characteristics (force, position, and momenta) are represented with three different arrays for each dimension in 3D space (x, y, z). Please note that the data size of simulation does not change with this layout in comparison to the 33% increase from the first method. This optimization is designated with a "c" in figures.

Step 6 - Pinned memory effect: Regular memory allocations on the host returns *pageable memory blocks* from the main memory. The OS has the permission to swap the pageable memory blocks to disk if it runs out the physical memory. This should not be the case for memory blocks that are used for *actual* data transfers between the host and the device. They have to remain in main memory throughout the data transfer operation. In such cases, the CUDA driver employs Direct Memory Access (DMA) to transfer data between host and devices. It starts by acquiring a temporary page-locked host memory (also referred to as the "staging" area) to initiate the copy operation. For a host-to-device copy operation, at first, the data is copied from the pageable area to the staging area. Then the copy operation is performed between the staging area on the host and the destination address on the device. Developers are able to utilize the staging area to their benefit and perform any allocation requests within the staging area. Such memories are called "pinned" memories in Nvidia terminology. By enabling a flag at compile time (`-ta=tesla:managed`), the PGI compiler replaces all regular memory allocation requests with the pinned memory ones. This way, we eliminate the internal copy operation to the staging area within the host (the copy operation from pageable to pinned memory). This step is dedicated to investigating the effect of enabling pinned memory on the performance our kernels.

Step 7 - Parameters of parallelism: After manually expressing parallelism opportunities for our code, we investigated how effective the compiler is in choosing the parallelization parameters for each kernel: `gang` and `vector`. These parameters accept a value, which developers are able to set at the runtime. They are directly mapped to their CUDA counter-parts, known as grid size and thread-block size, respectively. In most case when we do not manually specify values for these parameters, the runtime library promptly selects these values at the run time.

In some cases, the values chosen for these parameters by the compiler do not necessarily lead to optimal utilization of resources on the device. In such cases, computational resources are wasted since chosen values makes some of the resources inevitably idle. For instance, for a loop decorated with `vector` level, the total iteration count for the loop was 24 while the compiler specifies 128 threads for such a loop. This means that 81% of resources have to be idle! In addition, a reduction in the vector size leads to a higher cache hit ratio since the smaller number of threads compete for a limited resource; therefore, in such cases, more vectors lead to more cache thrashing.

Step 8 - Controlling resources at compilation time: Com-

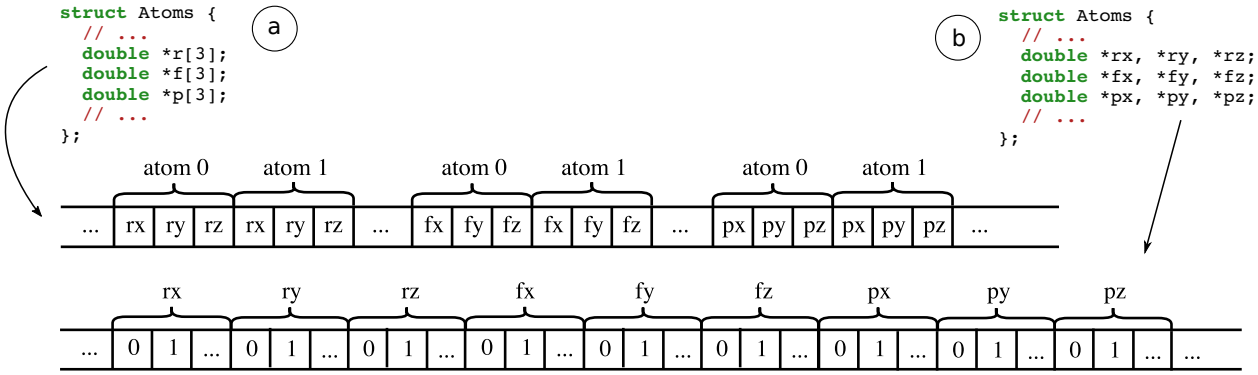

Figure 1: Data structures in CoMD and their corresponding data layout in memory. In the lower layout, numbers (0, 1, ...) refer to the atom ID (as shown in the upper layout).

pilers assign as many local variables as they can to GPU registers at compile time. Utilizing registers mitigates the latency of accessing global memory. The more our data resides in registers, the less we visit the global memory to fetch data. However, the total number of registers per `gang` is limited. The latest Pascal-based chip from NVIDIA has 65,536 registers per `gang`. We limited register usage per `gang` to simultaneously utilize most `gangs` on the device. In the PGI compiler, we limited the number of required registers for all kernels. Placing a cap on the number of registers per `gang` enables us to control the total number of `gangs` issued on the device simultaneously.

Step 9 - Unrolling fixed size loops: Loop unrolling is a loop transformation technique that helps application avoid pollution of the instruction pipeline on processors. It has its own advantages and disadvantages in parallel computing. The obvious drawback of unrolling is an increase in compilation time and application size. Because of the trade-off between this step and *Step 8*, We ended up using more registers per kernel. We totally unrolled one of the inner loops in force computation step (possessing 27 iterations). This transformation led to a ten-fold increase in compilation time.

Step 10 - Rearranging computations: As the final step, we rewrote some of the computations with efficiency in mind. For instance, we reduced the multiplication of constant values to each other to one value. We also converted some division operations into multiplications. And we also omitted one of the redundant *reduction* operations.

The above-mentioned steps provide us a roadmap for parallelization of a scientific program using the OpenACC programming model. Table 1 summarizes all the steps that we discussed here. Each step is specified in a row with title of step, whether `pointerchain` was used or not, and a brief description of the steps. Each step was applied on top of another one. However, when some steps adversely affected performance (for instance Steps 5a and 5b), we stopped applying them on our code. Figure 4 shows how these steps build off one another and which ones were dead ends. Directed edges in the figure shows the direction of how one step is applied on top of the other one; e.g., Step 4 is applied to the code after Step 3.

2. Supplementary Material B

We identified the optimal values of the number of `gang` and `vector` parameters for parallelization by traversing through a parameter space. We also searched for an optimal number of registers at the compilation time. These values contribute directly to the performance of our code at the final step (*Step 10*). Therefore, we used them for comparing the performance of OpenACC to that of its CUDA version.

a) Optimal configuration for parallelization: To find the optimal values of parameters in *Step 7*, we ran the simulation for 100 time steps and measured the execution time of each kernel. When the execution time of a kernel reaches its minimum, we choose the corresponding configuration as optimal. Figure 2 shows a heat map of execution time per kernel with respect to different values for `gang` and `vector` parameters on small (32,000) and large (2,048,000) datasets[1] The top and the bottom rows show the results for a large and a small dataset, respectively. The ellipsis shows the default configuration chosen by the runtime of the PGI compiler. The dashed rectangles show the optimal value within our search space. Our search space per kernel for the small dataset has 132 data points (22 `gangs` and 6 `vectors`) while the large dataset has 96 data points (16 `gangs` and 6 `vectors`). Figure 2 shows how the default configuration does not necessarily lead to better performance in all cases. Memory-bound kernels benefit from large number of `vectors` per `gang` in order to amortize memory access latency. So, the default `vector` count is close to the optimal value in our search space for such kernels, as the heat map demonstrates. However, compute-bound kernels do not benefit from large `vector` counts as their memory operations are as low as possible. The heat map confirms our findings, as we see a big difference between the optimal configuration and the configuration chosen by the compiler (as a fixed value). The PGI compiler chose 128 `vectors`, by default, for any configuration.

b) Optimal number of registers for each kernel: We followed the same approach as above and identified the required number of registers for each kernel in order to improve their

[1] We clamped the results to 50% of peak value of execution time. Gray color means 50% of the peak execution time or higher.

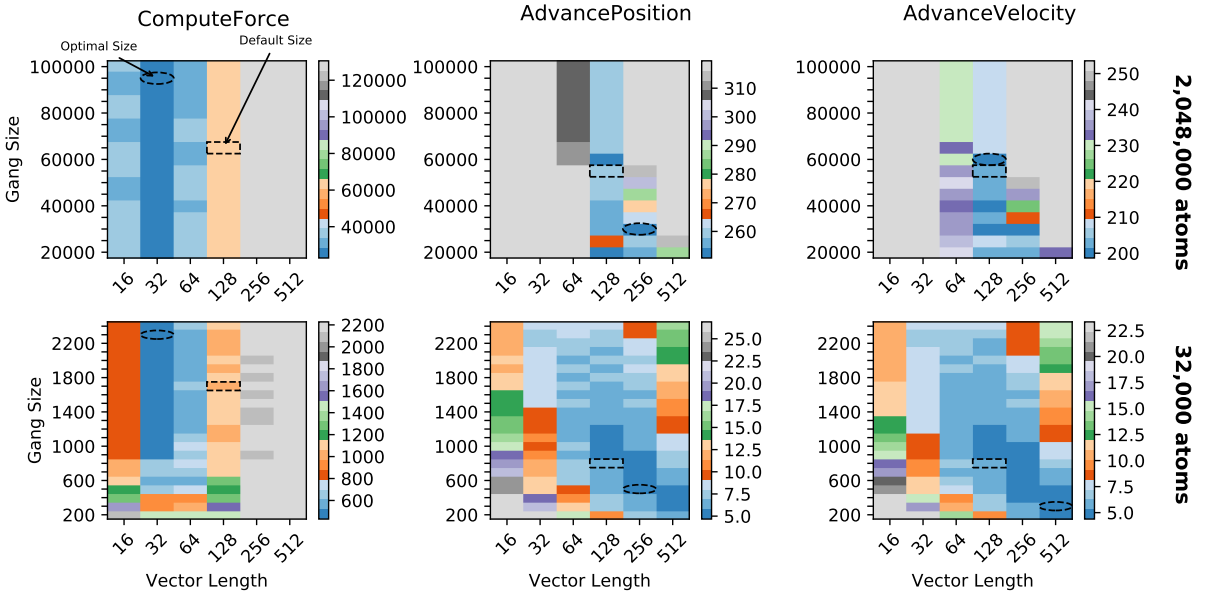

Figure 2: Heatmap of the `gang` (Y-Axis) and `vector` (X-Axis) parameters. Colors show execution time (μs) of the kernels with specific configuration. The top row presents results for 2,048,000 atoms and the bottom row for 32,000 atoms. Default size (*rectangle*) is the configuration that compiler has chosen and Optimal size (*ellipsis*) is the configuration that performs the best in our *search space* (dark blue represents the best execution time while the gray color represents the worst execution time). They differ in all three kernels; the difference is significant in the `ComputeForce` case.

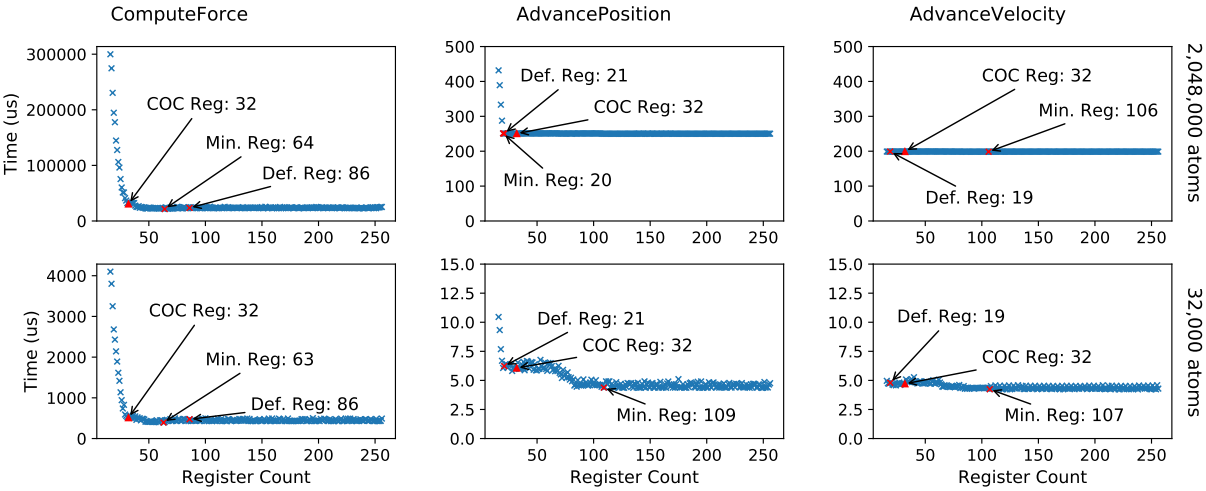

Figure 3: Effect of choosing an optimal register count. Embarrassingly parallel kernels (`AdvancePosition` and `AdvanceVelocity`) seem to not be affected. *Def. Reg.* and *COC Reg.* represent default register count selected by PGI compiler and proposed register count by COC, respectively. *Min. Reg.* represents the minimum register count and also the optimal value of register count, which determines the minimum execution time.

performance at *Step 8*. We set the `gang` and `vector` parameters to their best values from our last step and changed the register count. The maximum number of registers allocated to a kernel is determined only at the compilation time by the PGI compiler. Consequently, we changed the register count and then recompiled the program to collect the measurements for each kernel. The register count analysis was done per kernel per data size.

Figure 3 demonstrates the effect of register utilization on kernel performance. We changed the number of registers (X axis) from 16 to 256 and measured the execution time (Y axis) per kernel per data size. The optimal configuration is the one that achieves the minimum execution time with the smallest register count (*Min. Reg*). This implies that any data points close to the origin are optimal configuration points. The compiler has done a good job in choosing the correct number of registers except for only one case (*ComputeForce* for 32,000 atoms). Figure 3 also shows the default register count chosen by the PGI compiler (*Def. Reg*) and the CUDA Occupancy Calculator[2](*COC Reg*).

In the case of memory-intensive kernels, the PGI compiler was able to do a better job at allocating the right amount of register for each kernel; however, for the compute-intensive kernel from our application (`ComputeForce`), the compiler's choice lead to a waste of hardware resources on the device despite achieving equivalent performance. With the lesser number of registers per kernel, the scheduler will be able to dispatch more gangs to the system in hope of a better device utilization.

Occupancy in CUDA devices is defined as the number of active warps[3] on a Streaming Multiprocessor (SM) to the maximum number of active warps by an SM. That is, the COC reveals the achievement of maximum occupancy with our current configuration. For the compute-intensive kernel, `ComputeForce`, the register count selected by COC performs similarly to that of the optimal register count choice. For the two memory-intensive kernels, performance loss is negligible. Nevertheless, the register count selected by the compiler leads to a 30% performance loss in some cases (`AdvancePosition` for 32,000 atoms). With this step, we intend to show values chosen by the compilers affect the final performance of an application, and compiler developers must be discriminating in the choice of values for the parameters.

References

[1] OpenACC Language Committee, OpenACC Application Programming Interface, Version 2.6, `https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.6.final.pdf` (November 2017).

[2] R. Farber, CUDA Application Design and Development, Applications of GPU computing series CUDA application design and development, Elsevier Science, 2011.

[3] G. Mei, H. Tian, Impact of data layouts on the efficiency of gpu-accelerated idw interpolation, SpringerPlus 5 (1) (2016) 104.

[4] M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, I. Reguly, Designing op2 for gpu architectures, Journal of Parallel and Distributed Computing 73 (11) (2013) 1451–1460.

[5] G. R. Mudalige, M. B. Giles, J. Thiyagalingam, I. Z. Reguly, C. Bertolli, P. H. Kelly, A. E. Trefethen, Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems, Parallel Computing 39 (11) (2013) 669–692.

[2]The CUDA Occupancy Calculator (COC) helps developers to find the occupancy of multiprocessors of a GPU for a given CUDA kernel. The occupancy of a multiprocessor is the ratio of active warps to the maximum number of supported warps for that multiprocessor.

[3]A group of 32 adjacent threads within a gang