# Evaluating Performance Portability of OpenMP for SNAP on NVIDIA, Intel, and AMD GPUs using the Roofline Methodology

Neil A. Mehta[1], Rahulkumar Gayatri[1], Yasaman Ghadar[2], Christopher Knight[2], and Jack Deslippe[1]

[1] NERSC, Lawrence Berkeley National Laboratory
[2] Argonne National Laboratory

**Abstract.** In this paper, we show that OpenMP 4.5 based implementation of TestSNAP, a proxy-app for the Spectral Neighbor Analysis Potential (SNAP) in LAMMPS, can be ported across the NVIDIA, Intel, and AMD GPUs. Roofline analysis is employed to assess the performance of TestSNAP on each of the architectures. The main contributions of this paper are two-fold: 1) Provide OpenMP as a viable option for application portability across multiple GPU architectures, and 2) provide a methodology based on the roofline analysis to determine the performance portability of OpenMP implementations on the target architectures. The GPUs used for this work are Intel Gen9, AMD Radeon Instinct MI60, and NVIDIA Volta V100.

**Keywords:** Roofline analysis · Performance portability · SNAP.

## 1 Introduction

Six out of the top ten supercomputers in the list of Top500 supercomputers rely on GPUs for their compute performance. The next generation of supercomputers, namely, Perlmutter, Aurora, and Frontier, rely primarily upon NVIDIA, Intel, and AMD GPUs, respectively, to achieve their intended peak compute bandwidths, the latter two of which will be the first exascale machines. The CPU, also referred to as the host and the GPU or device architectures that will be available on these machines are shown in Tab. 1.

**Table 1.** CPUs and GPUs on upcoming supercomputers.

| System | Perlmutter | Aurora | Frontier |
|---|---|---|---|
| **Host** | AMD Milan | Intel Xeon Sapphire Rapids | AMD EPYC Custom |
| **Device** | NVIDIA A100 | Intel Xe Ponte Vecchio | AMD Radeon Instinct Custom |

The diversity in the GPU architectures by multiple vendors has increased the importance of application portability. A wide range of programming frameworks such as, Kokkos, [1] SYCL, [2] and HIP [3] have risen to address this challenge. These languages provide a single front-end for application developers to express parallelism in their codes while the frameworks provide an optimized backend implementation on the chosen architecture. However, these

programming models require an extensive rewrite of the application codes in C++, including the GPU kernels. Compiler directive-based programming models, such as OpenMP and OpenACC, present an attractive alternative for their ease of use and non-intrusive approach to parallelizing applications. OpenMP has been a popular compiler directive-based programming framework for CPUs and, OpenMP 4.0 onward has included directives that allow application developers to offload blocks of code onto GPUs for execution. OpenMP 4.5 and OpenMP 5.0 have increased the number of directives that will enable effective utilization of the available GPU resources. Compilers such as LLVM/Clang, XL (IBM), Cray, and GCC have already provided backend implementations to offload OpenMP directives on NVIDIA GPUs. Intel and AMD compilers are committed to supporting OpenMP 5.0 directives on their respective GPUs. Meanwhile, NVIDIA has a contract with NERSC to support a subset of OpenMP 5.0 directives for its compiler on the upcoming Perlmutter supercomputer, demonstrating long term investment in supporting OpenMP.

In this paper, we present an OpenMP 4.5 based implementation for the Spectral Neighborhood Analysis Potential (SNAP) module in LAMMPS. [4] TestSNAP is a stand-alone proxy app for SNAP that can be run independently of LAMMPS and is written in C++. While we have developed Kokkos, CUDA, and HIP versions of TestSNAP that we could have used for this profiling study, the wider use of OpenMP and its support by the GPU vendors makes it the perfect candidate for this study. The goal of this work was to create and test a single source-code implementation that can be compiled and scheduled on the NVIDIA, Intel, and AMD GPUs.

Application "Portability" implies the ability to compile and execute a single source code on multiple architectures. "Performance Portability" includes the ability to efficiently utilize available resources on the said architectures. A more formal definition states that a code can be considered "performance portable" is it consistently achieves consistent ratio of time-to-solution with the best time-to-solution on each platform with minimal platform specific changes to the code. In our study, the use of OpenMP 4.5 ensures that no platform specific changes are required. However, because GPUs from various vendors have different compute architectures, the "time-to-solution" is an inefficient metric for comparison. To assess the efficiency of an application on the target hardware, we have used the roofline analysis to test our OpenMP implementation of TestSNAP by comparing its arithmetic intensity (AI) with the peak achievable AI of the hardware.

We have compiled and executed TestSNAP on testbeds for each of the supercomputers mentioned above, i.e., Perlmutter, Aurora, and Frontier. Testbeds contain intermediary hardware that will fall somewhere between the Summit and exascale systems in terms of power and capabilities. The GPU racks on Cori at NERSC, the Iris node on Joint Laboratory for System Evaluation (JLSE) at Argonne National Lab, and the Hewlett Packard Enterprise built Cray Tulip machine serve as testbeds for Perlmutter, Aurora, and Frontier machines, respectively. GPUs available on each testbed are shown in Tab. 2. Even though, Intel's Gen9 GPU will not be used on the upcoming HPC machines, it does serve

**Table 2.** CPUs and GPUs available on test beds.

| Test bed | Cori-GPU | JLSE | Tulip |
|---|---|---|---|
| **Host** | Intel Skylake | Intel Xeon | AMD EPYC |
| **Device** | NVIDIA V100 | Intel Gen9 | AMD MI60 |

as a good platform to test performance portability of the code on the upcoming next generation discrete GPUs from Intel.

## 2 OpenMP offload implementation of TestSNAP

SNAP is an interatomic potential provided as a component of the LAMMPS MD toolkit. [4] When using the SNAP model, the potential energy of each atom is evaluated as a sum of weighted bispectrum components. The bispectrum, also known as the descriptor, describes the positions of neighboring atoms and the local energy of each atom based on its location for a given structural configuration. This bispectrum is represented by its components, which are used to reproduce the local energy [5]. The neighboring atom positions are first mapped over a three-dimensional sphere using the central atom as the origin to generate the bispectrum components. The mapping ensures that the bispectrum components are dependent on the position of the central atom and three neighboring atoms. Next, we calculate the sum over a product of elements of Wigner D-matrix, a smoothing function, and the element dependent weights. Because this product is not invariant under rotation, we modify it by multiplying with coupling coefficients, analogous to Clebsch-Gordan coefficients for rotations on the 2-sphere, to generate the bispectrum components. The band limit for bispectrum components is set by **J**, which determines how many and which bispectrum components are used for the simulation. We do not provide a detailed discussion on the SNAP algorithm since it is not in the scope of this paper. Instead, we provide the reader with the implementation details of TestSNAP, the proxy-app for SNAP, since they are necessary to understand it's OpenMP 4.5 implementation. The SNAP algorithm is explained in [6] by the original authors Thompson, *et al.*

### 2.1 Refactoring routines for GPUs

The pseudo-code for TestSNAP is shown in listing 1.1. Each of the compute routines shown in listing 1.1 iterate over the bispectrum components and store their individual contributions in a 1D array.

**Listing 1.1.** TestSNAP code

```
for(int natom = 0; natom < num_atoms; ++natom)
{
    // build neighbor-list for all atoms
    build_neighborlist();

    // compute atom specific coefficients
    compute_U(); //Ulist[idx_max] and Ulisttot[idx_max]
```

```
8       compute_Y(); //Ylist[idx_max]
9
10      // for each (atom,neighbor) pair
11      for(int nbor = 0; nbor < num_nbor; ++nbor)
12      {
13          compute_dU(); //dUlist[idx_max][3]
14          compute_dE(); //dElist[3]
15          update_forces()
16      }
17  }
```

`idx_max` represents the maximum number of bispectrum components and is determined by the value of **J**. TestSNAP problem sizes 2J14, 2J8, and 2J2 represent an `idx_max` size of 15, 9, and 3, respectively. For all three problem size, we use 2,000 atoms with 26 neighbors for each atom. The three problem sizes denote the number of descriptors used to describe the energy of the atom with respect to its surrounding. Therefore, even though the number of atoms for all three problem sizes remain the same, the number of descriptors used to describe the energy of these atoms ranges as 15, 9, and 3.

The `for-loop` in line 1 of listing 1.1 loops over all atoms in the simulation to compute forces in a given time-step. First, a list of neighboring atoms within a certain $R_{\text{cut}}$ distance, is generated for each atom inside the routine `build_neighborlist`. The `compute_U` routine calculates expansion coefficients for each (atom, neighbor) pair and stores this information in `Ulist`. The expansion coefficients for each atom are summed over all its neighbors and stored in `Ulisttot`. Next, the Clebsch-Gordon products for each atom are calculated in the routine `compute_Y` and stored in `Ylist`. As a precursor to force calculations, derivatives of expansion coefficients, stored in `Ulist`, are computed by `compute_dU` in all 3 dimensions using spherical co-ordinates and stored in `dUlist`. Using `dUlist` and `Ylist`, the force vector for each (atom,neighbor) pair is computed by `compute_dE` and stored in `dElist`. Finally, the force on each atom is computed from `dElist` in `update_forces`. A correctness check is built-in, which compares the proxy code output against a reference solution.

Based on the strategy used by newer SNAP implementation [7], the basic TestSNAP algorithm discussed above was refactored to prioritize the completion of each stage/routine for all atoms over the completion of all stages/routines for a single atom. In the algorithm shown above, which is based on the older GPU implementation of SNAP, [8], the work of each atom is mapped onto a GPU thread block, and hierarchical parallelism is used to exploit additional parallelism over the neighbor loop and the bispectrum components. However, converting the four major routines, namely, `compute_[U,Y,dU,dE]` as GPU kernels allow better utilization of GPU resources. In the refactored code, the atom loop is placed inside `compute_[U,Y]` and similarly, the atom and neighbor loops are placed inside `compute_[dU,dE]`. As an example, the refactored `compute_U`, shown in listing 1.2 is further refactored into two nested for loops, one to calculate `Ulist` and the other for `Ulisttot`.

**Listing 1.2.** compute_U

```
1   void compute_U()
2   {
```

```
    compute_uarray();
    add_uarraytot();
}
void compute_uarray()
{
    for(int natom = 0; natom < num_atoms; ++natom)
        for(int nbor = 0; nbor < num_nbor; ++nbor)
            for(int j = 0; j < idx_max; ++j)
                Ulist(natom,nbor,j) = ...
}
void add_uarraytot()
{
    for(int natom = 0; natom < num_atoms; ++natom)
        for(int nbor = 0; nbor < num_nbor; ++nbor)
            for(int j = 0; j < idxu_max; ++j)
                Ulisttot(natom,j) += Ulist(natom,nbor,j);
}
```

### 2.2 Use of multidimensional (MD) data structures

One of the disadvantages of refactoring is that it makes it necessary to store the atom and/or neighbor information as individual data structures across all routines. After refactoring, we need to store atom specific information in `Ulisttot` and `Ylist`, and (atom, neighbor) specific information in `Ulist`, `dUlist` and `dElist` arrays. To store this additional information, we create classes that mimic the behavior of multi-dimensional (MD) arrays, such that all elements are stored in a contiguous block of memory to improve memory locality. To achieve this behavior, we have created C++ classes that include a pointer to the contiguous block of memory and the information about the dimensions to calculate indexes of individual elements based on the access pattern.

**Listing 1.3.** Array2D

```
template <class T>
struct Array2D
{
    int n1, n2, size;
    T *dptr;

    Array2D(int in1, int in2)
        :n1(in1), n2(in2)
    {
        size = n1*n2;
        dptr = new T[size];
    }

    inline T& operator() (int in1, int in2)
    {
        return dptr[in1*n2 + in2];
    }

    Array2D(const Array2D& p)
    {
        n1 = p.n1; n2 = p.n2; size = 0;
        dptr = p.dptr;
    }

    ~Array2D()
    {
        if(size && dptr)
            delete[] dptr;
```

```
29      }
30  };
```

A bare bone structure of a 2D class is shown in listing 1.3. The first and second dimensions of the 2D array are stored as `n1` and `n2`, while `size` represents the total number of elements, i.e., `n1` × `n2`. `dptr` points to a contiguous block of memory for `size` number of elements. The operator overload of `()` allows us to implement a FORTRAN style indexing for the exact element that is requested. Hence on line 18 of listing 1.2, the element accessed by `Ulisttot` will evaluate to `Ulisttot.dptr[natom*idx_max + j]`. The copy constructor assigns `size` to zero, which allows us protection against multiple deletions of the same memory block, as shown in the destructor of the class on lines 22-26 of listing 1.3. Similar to `Array2D`, `Array3D` and `Array4D` classes are created to represent 3D and 4D arrays respectively. Array[2,3,4]D, i.e., ArrayMD classes, are templated over the data type of their elements for generalization. ArrayMD objects of complex-double type are created using a simple structure of two doubles to represent a complex number as shown in line 1 of List. 1.4. We are aware that there are standard multi-dimensional array classes available through C++ libraries. However, we wanted the ability to control on data storage and array access patterns specific to their usage on CPUs versus on GPUs. Therefore, these classes were created for the purposes of representing MD arrays in TestSNAP and only contain features that are needed by the application.

**Listing 1.4.** ArrayMD definitions of TestSNAP data structures.

```
1  struct SNAcomplex {double re,im;};
2
3  Array2D<SNAcomplex> Ulisttot(num_atoms,idx_max);
4  Array2D<SNAcomplex> Ylist(num_atoms,idx_max);
5  Array3D<SNAcomplex> Ulist(num_atoms,num_nbor,idx_max);
6  Array4D<SNAcomplex> dUlist(num_atoms,num_nbor,idx_max,3);
```

Data has to be moved from CPU to GPU memory space before distributing the work across GPU threads. We use the `map` clause in OpenMP to move data between CPU and GPU. `Ulist, Ylist, dUlist` are only needed on the GPU to store intermediary results between the compute routines. Hence, we use the `alloc` mapper-type with the `map` clause to avoid unnecessary memory allocation on CPU. ArrayMD classes are provided with a member function that creates an object without memory allocation, specifically for this purpose. In contrast, `dElist` is required for computing forces on the CPU after it is updated on the GPU. Therefore, we create a block of memory on the CPU and use the `to` and `from` mapper-type for data movement. Listing 1.5 shows how we achieve these two distinct mappings. On line 1 of List: 1.5, we map `Ulist` and `dElist` data structures on to the device using the `to` mapper-type, which performs a shallow copy of data structures on the device. On line 2, the `alloc` mapper type is used to allocate a block of memory on the device for `size` number of elements associated with `Ulist`, whereas, in line 3, a deep copy of the memory block pointed by the `dptr` of object `dElist` is performed. We use line 5 to copy the updated `dElist` array back to the CPU.

**Listing 1.5.** Use of OpenMP directives to `map` data

```
#pragma omp target enter data map(to: Ulist, dElist)
#pragma omp target enter data map(alloc: Ulist.dptr[0:Ulist.size])
#pragma omp target enter data map(to: dElist.dptr[0:dElist.size])

#pragma omp target exit data map(from: dElist.dptr[0:dElist.size])
```

### 2.3 Optimizing routines for OpenMP offload

In addition to refactoring TestSNAP routines, it is also necessary to understand data access patterns and OpenMP directives for further optimization of TestSNAP performance. We implemented three optimization strategies, each building on the previous one to make the final TestSNAP version highly performant on all three GPUs.

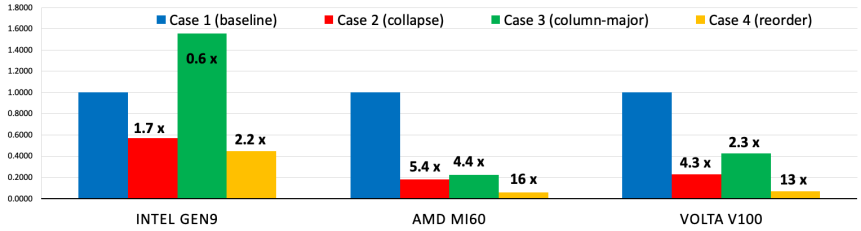


**Fig. 1.** Code speed-up improvement relative to naive OpenMP 4.5 implementation after array structure and loop access modifications at problem size 2J14.

Code performance is measured using grind-time, which is the average time taken per atom per time-step to complete the force calculation and is calculated in microseconds. The effectiveness of each optimization is measured in terms of,

$$Speed - up = \frac{new\ \ grind\ \ time}{naive\ \ grind\ \ time}. \tag{1}$$

The *naive grind time* is obtained by running the most trivial GPU parallelization on each architecture. The speed-up measured in this way ensures a fair way of comparing optimization gains specific to each architecture. If the bar is lower than one, it represents speed-up compared to baseline, and a greater than one measurement implies performance degradation. Our results for each of these optimizations are shown in Figs. 1 and 2. The performance plot is divided into three categories, one for each GPU under consideration. Improvement gains due to each optimization step are measured with respect to the naive OpenMP implementation, referred to as Case 1. All optimizations are discussed with the help of `add_uarraytot` kernel shown in List: 1.2.

**Case 1:** We need a baseline to compare the efficiency of our optimizations. Case 1 refers to the naive OpenMP implementation where the atom-loop is distributed across the GPU threads for `compute_[U,Y,dU,dE]` routines.

**Listing 1.6.** Atom loop parallelization in add_uarraytot

```
void add_uarraytot()
{
#pragma omp target teams distribute parallel for
    for(int natom = 0; natom < num_atoms; ++natom)
        for(int nbor = 0; nbor < num_nbor; ++nbor)
            for(int j = 0; j < idxu_max; ++j)
                ulisttot(natom,j) += ulist(natom,nbor,j);
}
```

An example of our naive OpenMP implementation on `add_uarraytot` is shown in List. 1.6.

**Case 2:** Except in `compute_Y`, each atom loops over its neighbors in all other routines. A logical progression to parallelizing the atom loop is to include the neighbor loop in the parallelization effort wherever possible. We can achieve this by using the `collapse` clause in OpenMP. An unavoidable consequence of the collapse clause makes it necessary to use `atomic` operations when updating `Ulisttot`, as shown in List. 1.7.

**Listing 1.7.** Atom and neighbor loop parallelization in add_uarraytot

```
void add_uarraytot()
{
#pragma omp target teams distribute parallel for collapse(2)
    for(int natom = 0; natom < num_atoms; ++natom)
        for(int nbor = 0; nbor < num_nbor; ++nbor)
            for(int j = 0; j < idxu_max; ++j)
            {
                #pragma omp atomic
                ulisttot(natom,j) += ulist(natom,nbor,j);
            }
}
```

While `atomic` calls are expensive, in this case, the benefits of increase in parallelism achieved by looping over the neighbor dimension outweighs the overhead incurred due to atomic operations. Distributing work over the atom and neighbor dimension by the use of `collapse` clause gave us a 1.7× performance boost on Intel Gen 9, while on AMD and Volta GPUs it gave us a 5.4× and 4.3× performance improvement respectively.

**Case 3:** One of the most common optimizations on GPUs is the use of column major data access pattern to improve memory coalescing. However, as shown on line 16 of List. 1.3, we use the row-major style of indexing into the elements of ArrayMD structures which helps to avoid cache thrashing and false sharing on CPUs. Because of the modular design of ArrayMD structure, we can easily modify the operator overload to support column major data access, as shown in List. 1.8.

**Listing 1.8.** Column major indexing in Array2D

```
inline void operator()(int in1, int in2) {return dptr[in2*n1 + in1];}
```

But this modification does not lead to the intended speed-up. In fact, it leads to performance degradation on all GPUs compared to case 2, as shown in Fig. 1.

The reason for this performance degradation is explained in Case 4.

**Case 4:** The advantage of column major data access on GPUs is the alignment of memory accesses to reduce memory latency on SIMD architectures. In our case this leads to atom dimension being accessed first by consecutive threads. Collapsing the loops makes the index of the innermost loop as the fastest moving index, which implies that the neighbor index becomes the fastest moving index. In order to gain benefit from the column major access pattern, we swap the loop order of atoms and neighbor in each of the routines, as shown on lines 4 and 5 in List. 1.9.

**Listing 1.9.** Atom and neighbor loop swap

```
1  void add_uarraytot()
2  {
3  #pragma omp target teams distribute parallel for collapse(2)
4      for(int nbor = 0; nbor < num_nbor; ++nbor)
5          for(int natom = 0; nbor < num_atom; ++natom)
6              for(int j = 0; j < idxu_max; ++j)
7              {
8                  #pragma omp atomic
9                  ulisttot(natom,j) += ulist(natom,nbor,j);
10             }
11 }
```

This allows us to take the advantage of coalesced memory access and gives us the best performance across all 3 GPUs. Speed-ups obtained for problem size 2J8 are similar to those for 2J14, as shown in Fig. 2. Applying case 3 optimization to the 2J8 problem did not degrade the performance to the extent observed in 2J8, which may be because 2J8 problem size relies on smaller ArrayMD structs. The performance gains after case 4 optimizations, although not as high as those for 2J14, are still significant highlighting the efficacy of the applied optimizations.

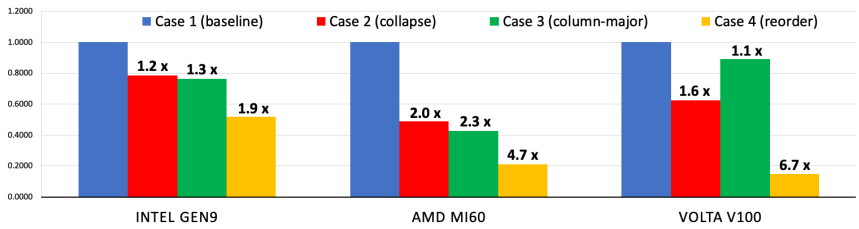


**Fig. 2.** Code speed-up improvement relative to naive OpenMP 4.5 implementation after array structure and loop access modifications at problem size 2J8.

Compiler maturity plays a significant role in our ability to efficiently map OpenMP directives on GPUs. Because the support for OpenMP directives on GPUs is still in its early stages, each new version of the compiler can give a significant advantage in terms of new features and increased efficiency of the existing directives. Our OpenMP version of TestSNAP can be successfully compiled and executed on an NVIDIA V100 GPU with the LLVM/10.0 compiler.

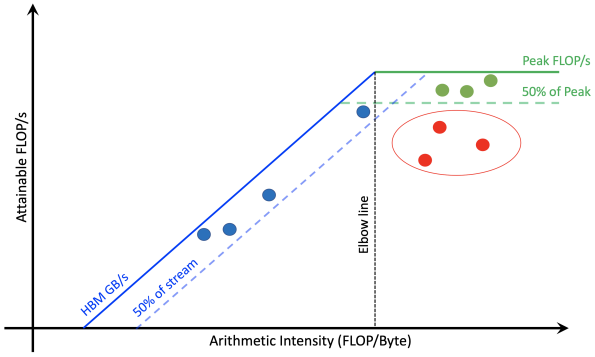


**Fig. 3.** Schematic of roofline plot, showcasing typical kernel placements.

However, with LLVM/11.0 [9] as well as Intel's ® DPC++/C++ (ICX), [**?**] the code triggers a bug, which results in the compiler being unable to map our `SNAcomplex` structure, shown in List. 1.4, on to the device memory. To bypass this bug, we have modified our ArrayMD structures of complex-doubles to structures of doubles with twice the size, such that even and odd indices point to real and imaginary values, respectively. On AMD M160 we have used AOMP version 11.5.1, which is based on LLVM/11.0. The bug reported to LLVM 11.0 has since been fixed in version 12.0 and has been under review by Intel compiler developers. It is important to note that we allowed the compiler to optimize the number of teams and threads when running TestSNAP on all three GPUs. We observed that the compiler optimized teams and threads input always provided better run times compared to runs with manual input.

## 3 Methodology of roofline analysis

Modern computing architectures are varied and are considered guarded proprietary information of the vendor. Therefore, for a fair comparison, the roofline model utilizes a simplified memory model, which assumes that all caches are perfect. Under this assumption, the data flows between DRAM to cache with sufficient bandwidth to not affect performance. Other assumptions include, the communication and computation perfectly overlap, and cores can attain peak floating point operations per second (FLOPs) on local data. These assumptions allow one to measure kernel performance in terms of FLOPs capped by either the peak attainable machine FLOPs or the amount of data that can be moved based on the peak bandwidth throughput.

The measure of how well a kernel can benefit from the data reuse and device bandwidth is quantified by the AI, which is calculated as the number of FLOPs executed per byte of memory transferred to the memory level and is calculated for each level of the memory hierarchy. A roofline plot is formed by plotting attainable FLOPs as a function of AI for a given kernel on a log-log plot. The x-

and y-axis represent AI and performance, i.e., FLOPs, respectively. A schematic of a typical roofline plot is shown in Fig. 3. The solid blue line represents the peak attainable bandwidth for a particular memory hierarchy, in this case, the HBM or DRAM of a given machine. No kernels can lie to the left of this line as the attainable FLOP rate will always be bottle-necked by the throughput capacity of the device. At any performance (FLOPs), for a kernel to lie to the left of the bandwidth line, the denominator, i.e., the data transfer rate, will have to be greater than the peak bandwidth of the memory hierarchy. Similarly, the solid green line represents the peak FLOP rate of the machine, which is determined by the machine cycle and is dependent on the compute architecture. The point at which these two bounds meet is known as the "elbow", and the line joining the elbow to the x-axis is called an elbow line. All kernels to the left of this elbow line are termed as "memory-bound" because their performance is strongly affected by the memory bandwidth of the machine. Kernels to the right of the elbow line are classified as "compute-bound" because they are bound by the compute capability of the machine.

A roofline helps determine kernels where optimization efforts are most beneficial. A couple of kernels are shown in blue, green, and red in Fig. 3. Kernels shown in blue lie to the left of the 50% peak bandwidth line. While these kernels have low AI, any additional improvement which increases the FLOPs will lead to a relatively small gain in code performance as these kernels are "memory-bound". Kernels represented by green dots lie above the 50% peak FLOPs rate line and are therefore making good utilization of the machine. In contrast, the kernels shown in red have higher AI than many of the kernels shown in blue, but they have not yet reached 50% of either compute or memory capacity. Optimizing these kernels will provide maximum gains in performance compared to other kernels, which are already capped by either the bandwidth or peak FLOP rate of the machine. The roofline plot also allows one to estimate the kernel performance on future machine architectures. Assuming that an application has a majority of kernels that are "memory-bound", running this application on machines with higher compute capability but the same memory bandwidth will provide only a small improvement in the run-time and vice versa. Ideally, for modern GPUs, where we have more compute power than the memory bandwidth, developers should aspire to make their kernels compute-bound.

## 4 Results and discussion

### 4.1 Profiling code performance

To understand the performance difference between LLVM/11.0 and Intel ® DPC++/C++ compilers, we have profiled TestSNAP on the Skylake 8180 processor. As shown in Tab. 3, the step and grind times are similar for LLVM/11.0 and Intel ® DPC++/C++ for the serial TestSNAP code on the Skylake processor. The LLVM/11.0 compiler is marginally better, which we suspect may be due to the maturity of the LLVM/11.0 compilers in terms of performance refinement compared to the newly

**Table 3.** Comparison of OpenMP offload profiles on GPU measured for the 2J14 problem size for 100 time steps.

| Version | Serial (Skylake) | | OpenMP offload GPU | | |
|---|---|---|---|---|---|
| | LLVM/11 | ICX | Gen9 | MI60 | V100 |
| **Step time (s/step)** | 9.7671 | 9.8669 | 1.8215 | 0.1394 | 0.0565 |
| **Grind time (ms/atm-stp)** | 4.8835 | 4.9334 | 0.9107 | 0.0697 | 0.0282 |
| `compute_U` (s) | 0.6211 | 0.6221 | 0.1975 | 0.0153 | 0.0099 |
| `compute_Y` (s) | 7.6839 | 7.6789 | 1.2005 | 0.0748 | 0.0271 |
| `compute_dU` (s) | 1.2008 | 1.3363 | 0.3484 | 0.0389 | 0.0155 |
| `compute_dE` (s) | 0.2604 | 0.2288 | 0.0741 | 0.0086 | 0.0028 |

introduced Intel ® DPC++/C++. However, for the intent of our comparison, the performance of both compilers on the Skylake processor is considered equal.

**Table 4.** Comparison of OpenMP offload kernel time loads of top 5 kernels, measured for problem size 2J14, 2000 atoms, and 100 time steps.

| Version | Intel Gen9 | | AMD MI60 | | NVIDIA V100 | |
|---|---|---|---|---|---|---|
| Rank | Time (%) | Kernel | Time (%) | Kernel | Time (%) | Kernel |
| 1 | 65.65 | `compute_Y` | 57.01 | `compute_Y` | 45.32 | `compute_Y` |
| 2 | 19.15 | `compute_dU` | 31.53 | `compute_dU` | 25.80 | `compute_dU` |
| 3 | 10.58 | `compute_U` | 8.61 | `compute_U` | 15.75 | `compute_U` |
| 4 | 4.02 | `compute_dE` | 2.44 | `compute_dE` | 8.60 | `memcpy HtoD` |
| 5 | 0.41 | `WriteBuffer` | 0.29 | `zero_uarraytot` | 3.96 | `compute_dE` |

We use the IProf, ROCProf, and NVProf to profile our OpenMP implementation of TestSNAP on Intel, AMD, and NVIDIA GPUs, respectively. The relative time required by the individual kernels is shown in Tab. 4. While all three GPUs spend the highest amount of time in `compute_Y` followed by `compute_dU` and `compute_U`, the individual percentages vary. `compute_Y` is computationally the most expensive kernel followed by `compute_dU`, and hence they are proportionally the most expensive on each architecture. The initial data movement from device to host on V100, shown in Tab. 4 as `memcpy HtoD` contributes 8.6% to the total runtime. Currently, we are unable to obtain the time spent in the data movement on the MI60 GPU using ROCProf, and therefore they are absent in the table. The data movement cost on the Gen9 GPU, represented as `WriteBuffer`, is much lower because of the non-discrete nature of the Gen9 GPU design. `zero_uarraytot` only initializes `Ulisttot` to zero and therefore has very low cost.

It should be noted that the percentage times for the kernels shown in Tab. 4 are obtained for problem size 2J14 and 100 timesteps. Reducing the number of time steps leads to an increase in the fraction of time spent on data movement. Similarly, reducing the problem size to 2J8 changes the order of kernel time contribution, such that `compute_dU` is most expensive followed by `compute_Y`,

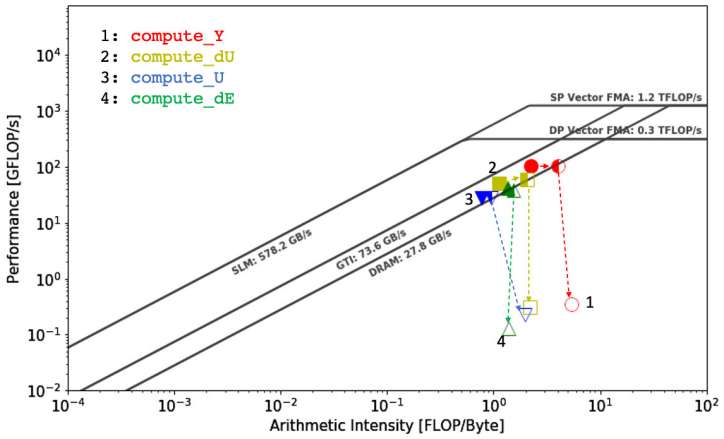


**Fig. 4.** DRAM roofline plot on **Intel Gen9**. Arrows point from 2J14->2J8->2J2. Problem sizes 2J14, 2J8, and 2J2 are represented by full, half, and open symbols, respectively.GTI and SLM are abbreviations of "Graphics Technology Interface" and "Shared Local Memory", respectively. The SLM is analogous to L3 cache.

`compute_U`, and `compute_dE`. As noted previously, problem size 2J2 is too small to obtain meaningful data for real-world applications. However, we will discuss the roofline results for 2J2 because it highlights some interesting differences between the three GPUs.

### 4.2 Roofline analysis of TestSNAP code

**Performance on Intel Gen9 GPU** A high-level building block of Intel Gen 9 GPUs is the slice, and for an Intel Xeon Processor E3-1585 v5 with Iris Pro Graphics P580 (GT4e), which was used in this work, contains 3 GPU slices. Each GPU slice consists of 3 sub-slices, an L3 data cache bank, and shared local memory. A sub-slice has 8 execution units (EUs), each containing 7 threads. Intel processors include fast high bandwidth embedded DRAM (EDRAM) of 128 MB into which the GPU may allocate memory.

We have used Intel® Advisor to collect the relevant metrics necessary to generate the roofline plot and obtain the memory and compute peaks of the Gen9 GPU. Metrics were obtained using the command shown in List. 1.10.

**Listing 1.10.** Command to collect the metrics for Intel Gen9

```
1 advixe-cl --collect=roofline --enable-gpu-profiling --project-dir=$PRJ --
      search-dir src:r=$SRC -- ./test_snap.exe -ns 100
```

here, `$PRJ` and `$SRC` denote the locations of user defined project directory into which roofline results are stored and the code source directory, respectively. The roofline plot of TestSNAP running on the Gen9 GPU is shown in Fig. 4 for problem sizes 2J14, 2J8, and 2J2. The numbers in Fig. 4, next to the symbol,

correspond to the kernel names shown in the legend. Of note, from the Gen9 GPU roofline plot, while indicating the performance of kernels, it is also possible to obtain details of data flow specific to the Gen9 compute architecture. As a rule, kernel roofline symbols should never cross the memory hierarchy peak bandwidths for which they are measured. For the roofline shown in Fig. 4, the roofline data is generated at the DRAM level for all three problem sizes. However, for 2J14 problem size, the roofline symbols are placed left of the DRAM peak bandwidth line. This indicates that the data movement is not measured across DRAM but across the faster embedded-DRAM or eDRAM, a special feature of the Gen9 GPU.

All the kernels for the 2J14 and 2J8 problem sizes are bound by the peak bandwidth of DRAM, as indicated by the kernel symbols located close to the DRAM bandwidth line. `compute_Y` is located close to the elbow created between DRAM bandwidth and DP vector FMA peaks, which represents the region separating "compute" and "memory-bound" regions. This indicates the simultaneous usage of the available compute and memory resources. The other three kernels are not as close to the elbow and are "memory-bound". Finally, based on the location of the 2J14 and 2J8 kernels, it is observed that all kernels are "memory-bound". When running a smaller problem size of 2J8, less data movement is required than 2J14, leading to higher AI for similar performance numbers, leading to a rightward shift of the kernel roofline positions. When running the smallest problem size 2J2, the required number of FLOPs is much less than the data moved, which leads to a large downward shift of the kernel roofline positions, markedly demonstrating poor use of compute resources.

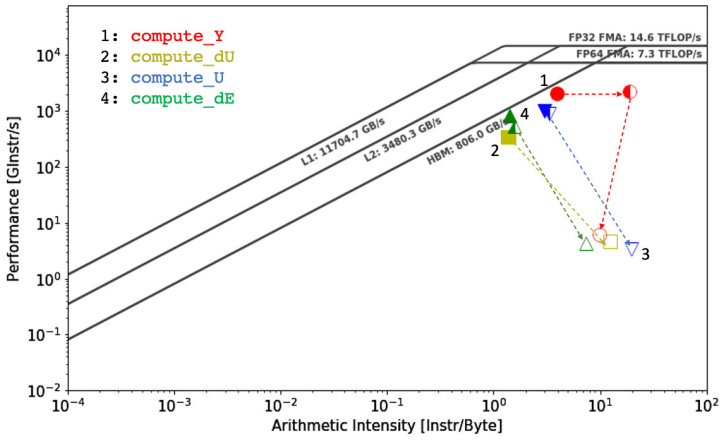


**Fig. 5.** DRAM roofline plot on **AMD Instinct MI60**. Arrows point from 2J14->2J8->2J2. Problem sizes 2J14, 2J8, and 2J2 are represented by full, half, and open symbols, respectively.

**Performance on AMD Radeon Instinct MI60 GPU** We have used the ROC profiling tool to obtain the metrics required for the roofline plot on AMD MI60. ROCProf used in this work is a part of the AMD ROCm version 3.6, which is an open-source code development platform. Unlike Intel Advisor or NVIDIA NSight Compute, we could not obtain the FLOP count of each kernel. Instead, we have used the instruction based roofline model [10] to evaluate the roofline performance of TestSNAP kernels on MI60. To calculate the number of instructions executed, we have used metrics `SQ_INSTS_VALU` and `SQ_INSTS_SALU` to obtain the number of vector and scalar instructions issued, respectively. We have used metrics `FETCH_SIZE` and `WRITE_SIZE`, to gather read and write data movements, respectively. These metrics are listed in the `input.xml` and provided to ROCProf using the command shown in List. 1.11. Compute and memory bandwidth peaks were obtained from Richards, *et al.* [11]

**Listing 1.11.** Command to collect metrics for MI60

```
1 ROCProf -i input.xml -o roofline.csv ./test_snap.exe -ns 100
```

The instructions based roofline data generated using the metric collected at the DRAM level is shown in Fig. 5. `compute_dU`, `compute_U`, and `compute_dE` are all "memory-bound". Similar to the roofline plots discussed previously, kernel `compute_Y` is the most compute-intensive and, therefore, has the highest AI. It is also close to the DRAM peak bandwidth line as well as the compute-bound region, which indicates that `compute_Y` is well-optimized and makes good use of MI60 resources.

Comparing the roofline data of problem size 2J8 with 2J14, except `compute_Y`, all kernels retain their position on the roofline plot. This is because all the necessary data required to run these kernels for problem sizes 2J14 and 2J8 are bound by the DRAM bandwidth, meaning that almost all the required data is fetched from DRAM for both cases. This leads to similar AI and performance numbers for both 2J14 and 2J8. However, this is not the case for `compute_Y`, where the amount of data moved required for instructions executed is lower when running the smaller problem size, 2J8. `compute_Y` relies on the beta coefficients stored in the database files, and at a smaller problem size of 2J8, a lesser number of coefficients are used, and therefore, less data has to be moved. This is definitely the case for all kernels at the smallest problem size 2J2, and consequently, all kernels shift to the "compute-bound" region. The location of the kernels for 2J2 roofline indicates poor use of AMD MI60 GPU resources but shows better utilization at problem size 2J14 and 2J8.

**Performance on NVIDIA Volta V100 GPU** The V100 belongs to the Volta family of NVIDIA GPUs and, compared to Intel's Gen9 and AMD's MI60, has been more widely adopted. Metrics necessary to generate the roofline plot were collected using NVIDIA NSight Compute, an interactive kernel profiler. NSight Compute functionality is supported for applications running on NVIDIA GPUs and is provided with CUDA toolkit version 11.0. A total of 11 metrics are collected to obtain the average elapsed time, the number of single and double

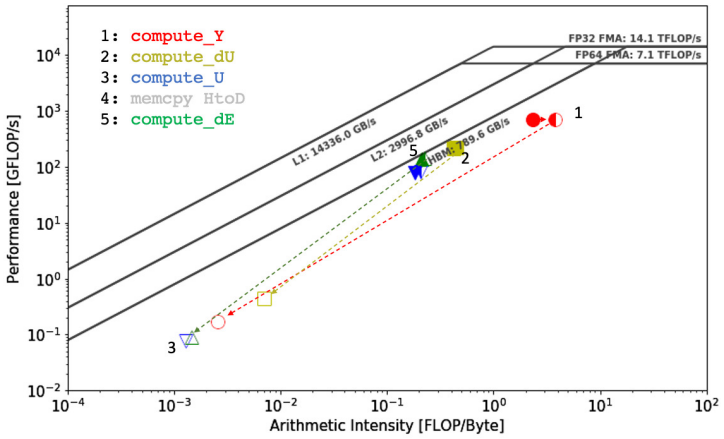


**Fig. 6.** DRAM roofline plot on **NVIDIA Volta V100**. Arrows point from 2J14->2J8->2J2. Problem sizes 2J14, 2J8, and 2J2 are represented by full, half, and open symbols, respectively.

precision add, multiply, and fused multiply and add (FMA) operations. The data movement across dram, and L2 and L1 caches is tracked for each kernel using metrics `dram__bytes`, `lts__t_bytes`, and `l1tex__t_bytes`, respectively. Metrics necessary for roofline analysis are collected using List. 1.12.

**Listing 1.12.** Command to collect metrics for V100

```
nv-nsight-cu-cli --metrics $metrics --csv ./test_snap.exe -ns 100 > metrics.
    log
```

where, `$metrics` refers to the metrics discussed above. Compute and memory bandwidth peaks were also obtained from the NSight Compute toolkit.

The DRAM memory roofline of V100, for the three problem sizes, is shown in Fig. 6. Similar to the roofline plots from other GPUs, even for V100, all kernels are positioned in the "memory-bound" regime and are close to the DRAM peak bandwidth line. However, it is possible to observe smaller differences between MI60 and V100 performance. For example, the roofline of `compute_Y`, is comparatively farther away from the DRAM peak bandwidth line in Fig. 6 than in Fig. 5. Also, the AI of this kernel is lower than that on MI60. This can potentially be attributed to better communication optimization of TestSNAP kernels on MI60.

This assessment can be made by comparing roofline differences of `compute_Y` for problem sizes 2J14 and 2J8 on these two machines. For problem size 2J8, not as much data has to be moved across the memory levels, which pushes `compute_Y` into "compute-bound" region on MI60, whereas, on V100, the kernel still stays in "memory-bound" region. This suggests that data movement was better for this kernel on MI60 compared to that on V100. Not surprisingly, kernels that are not heavily reliant only on data movement sit closer to the DRAM peak bandwidth

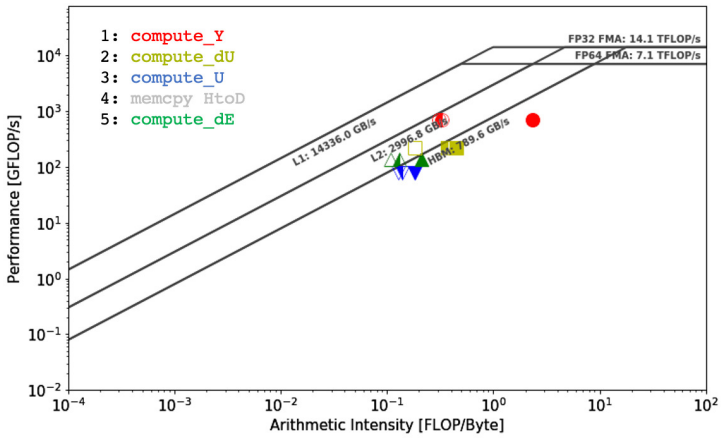


**Fig. 7.** Hierarchical roofline plot on **Volta V100**, for problem size 2J14. L1, L2, and DRAM performance are represented by open, half, and full symbols, respectively.

line on V100 compared to MI60. The rooflines of kernels for problem size 2J2 are significantly different on the two machines. On MI60, all kernels are located in the "compute-bound" region with relatively higher AI as shown in Fig. 5. In contrast, all the kernels are located in the "memory-bound" region with poor AI, as shown in Fig. 6. Looking at the raw metrics, we can observe that a lot more data is moved across DRAM memory on V100 compared to MI60, and as a consequence, the AI of 2J2 problem size is higher on MI60.

NSight Compute profiler provides an additional level of detail with the ability to capture data transfer not only across DRAM but also across the L2 and L1 cache levels. This data movement can then be used to generate cache specific AI numbers and plot roofline, which can pinpoint data reuse and kernel cache level bounds. Note that the kernel performance is ultimately a minimum of the AI obtained across all memory levels. Roofline models generated in this manner are categorized as hierarchical rooflines.

The L1, L2, and DRAM specific, i.e., the hierarchical roofline plot of TestSNAP kernels, for problem size 2J14 on V100, is shown in Fig. 7. The noticeable difference in the AI of `compute_Y` indicates that the actual AI of this kernel is not greater than one, but it is approximately 0.3. The proximity of L1 and L2 roofline symbols to the L2 cache peak bandwidth line suggests that `compute_Y` is L2 cache bound. Similarly, except for `compute_U`, the performance of all other profiled kernels is L2 cache bound. For all memory levels, roofline symbols of `compute_U` lie below the DRAM peak bandwidth roof, and therefore, it is considered DRAM bound.

The hierarchical roofline model provides additional details regarding the use of memory hierarchy. In Fig. 7, for `compute_Y`, `compute_dU`, and `compute_dE`, there is a large shift in their AI between DRAM and L2-L1 rooflines. This is a sign of high data reuse and good utilization of memory hierarchy. In contrast,

the shift in AI is almost negligible between L2 and L1 cache levels, representing poor utilization of hierarchy, which results from these kernels having to access data from the L2 cache to perform operations. For kernel `compute_U`, because data has to be accessed from DRAM, there is very little shift in AI across L1, L2, and DRAM rooflines, indicating little use of memory hierarchy. The performance of these kernels can be improved by having a larger bandwidth L2 cache and DRAM memory and optimizing and reducing the data movement necessary to execute the kernels.

## 5 Related work

Because of the early adoption of OpenMP directives, we were able to learn from the experiences of Vergara Larrea, *et al.* [12] who used OpenMP 4.0 directives to port codes to NVIDIA GPUs. The challenges of using OpenMP 4.5 for performance portability has been documented in detail in work by Gayatri, *et al.* [13] This study laid the groundwork for improving TestSNAP serial version using OpenMP. From this study, it was observed that the collapse clause would be better optimized using the column-major data storage format for 2D and higher dimensional arrays. This early experience helped improve the overall performance of TestSNAP on all tested GPUs. However, for the previous study, the compiler was not as mature in supporting OpenMP offload features. This study demonstrates performance analysis of a real-world application using a mature compiler that is supported by two of the three major GPU architectures.

The roofline model was introduced by Williams, *et al.* [14] in 2009, which made it possible for researchers to measure performance across multiple architectures objectively. Previous works, [15–17] in particular by Yang, *et al.* [18] have been instrumental in developing the theory of the roofline model, tabulating the metrics, and providing a recipe to generate roofline plots. While roofline models have been measured on all three GPU architectures separately, to the best of our knowledge, this is the first time a single application has been analyzed using the roofline model on three GPU architectures with no modifications to the code. This is truly unique because it provides a common standard to measure compiler and GPU architecture improvements.

## 6 Conclusions and future work

In this work, we show that it was possible to create a single source code implementation of TestSNAP using OpenMP 4.5 directives, which is portable across NVIDIA, Intel, and AMD GPUs. To our knowledge, this is the first study the same code was run on three GPU architectures without architecture specific modifications using OpenMP. We also show that standard GPU optimizations such as column-major data access patterns and exploiting more performance by collapsing loops give performance benefits across all GPUs.

TestSNAP run- and grind-times show that the NVIDIA's V100 GPU achieved the highest speed-up with a grind-time of 0.0282 ms/atom-step compared to the

serial grind-time of 9.797 ms/atom-step on Intel's Skylake architecture. However, grind-times do not show a complete picture, and this is demonstrated by the roofline models, which show that the TestSNAP kernels are memory-bound on all three GPU architectures. Roofline plots of Gen9, MI60, and V100 indicate that all significant kernels are bound by the DRAM bandwidth. These kernels are positioned in the "memory-bound" region of the roofline plot, and therefore, performance can be improved by changing the algorithm to increase the AI. The ability to collect additional cache level, data movement metrics using CUDA's NSight Compute profiler meant that hierarchical roofline models could be built for TestSNAP on V100 GPU.

As observed from the kernel roofline symbols, the majority of the TestSNAP kernels are "memory-bound". Ideally, kernels should be "compute-bound" and should be closer to the peak compute capacity line. To achieve this, we will work towards better memory access patterns and higher data reuse, particularly for kernel `compute_Y`, as it has the largest time footprint. Furthermore, we will also work towards better cache utilization to improve the overall AI of the TestSNAP code.

## 7 Acknowledgement

The TestSNAP version used in this work is a highly modified variant of the TestSNAP proxy app written by Dr. Aidan Thompson. We would like to thank Drs. Danny Perez, Noah Reddell, and Nicholas Malaya for enabling us access and providing compute resources on the DOE's Cray Tulip machine. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. We would also like to thank NERSC for providing us with compute resources.

## 8 Reproducibility

The results shown in this work are reproducible by downloading the code from the git repository: `https://github.com/FitSNAP/TestSNAP/tree/OpenMP4.5`. However, as mentioned previously, if the code does not compile due to regression test failure, an alternate version of the TestSNAP code without the array of structs is available from github repository: `https://github.com/namehta4/TestSNAP/tree/mod_OpenMP4.5`. Both these versions have similar compute performance. The compiler flags used to compile TestSNAP OpenMP offload on Intel Gen9, AMD MI60, and NVIDIA V100 are provided in listings 1.13, 1.14, and 1.15, respectively.

**Listing 1.13.** Compiler flags for Intel Gen9

```
icx -O3 -fstrict-aliasing  -Wno-openmp-target -Wall -Wno-unused-variable -std
    =c++11 -qnextgen -fiopenmp -fopenmp-targets=spir64 *.cpp  -o test_snap.
    exe
```

**Listing 1.14.** Compiler flags for AMD MI60

```
clang++ -O3 -fstrict-aliasing  -Wno-openmp-target -Wall -Wno-unused-variable
    -std=c++11 -lm -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa -Xopenmp-
    target=amdgcn-amd-amdhsa -march=gfx906 -ffp-contract=fast *.cpp  -o
    test_snap.exe
```

**Listing 1.15.** Compiler flags for NVIDIA V100

```
clang++ -O3 -fstrict-aliasing  -Wno-openmp-target -Wall -Wno-unused-variable
    -std=c++11 -lm -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda --cuda-path=
    $(CUDA_PATH) -I/$(CUDA_LIB) -ffp-contract=fast *.cpp  -o test_snap.exe
```

## References


1. H. Carter Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014.
2. R. Reyes and V. Lomüller. SYCL: Single-source C++ accelerator programming. In *PARCO*, pages 673–682, 2015.
3. ROCm HIP. ROCm HIP. `https://github.com/ROCm-Developer-Tools/HIP`.
4. S. Plimpton. Fast Parallel Algorithms for Short-Range Molecular Dynamics. *Journal of Computational Physics*, 117(1):1–19, 1995.
5. A.P. Bartók, M.C. Payne, R. Kondor, and G. Csányi. Gaussian Approximation Potentials: The Accuracy of Quantum Mechanics, without the Electrons. *Physical Review Letters*, 104(13):136403, 2010.
6. A. Thompson, L. Swiler, C. Trott, S. Foiles, and G. Tucker. Spectral neighbor analysis method for automated generation of quantum-accurate interatomic potentials. *Journal of Computational Physics*, 285(1):316–330, 2015.
7. S. Plimpton, A. Kohlmeyer, A. Thompson, S. Moore, and R. Berger. LAMMPS Stable Release 3 March 2020. `https://zenodo.org/record/3726417#.Xz2NMS2z3Vu`, Mar 2020.
8. C. Trott, S. Hammond, and A. Thompson. SNAP: strong scaling high fidelity molecular dynamics simulations on leadership-class computing platforms. In *International Supercomputing Conference*, pages 19–34. Springer, 2014.
9. Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.
10. N. Ding and S. Williams. An instruction roofline model for GPUs. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 7–18. IEEE, 2019.
11. D. Richards, O. Aaziz, J. Cook, J. Kuehn, S. Moore, D. Pruitt, C. Vaughan, and The ECP Proxy App Team. Quantitative Performance Assessmentof Proxy Apps and Parents. In *LLNL-TR-809403. Report for ECP Proxy App Project Milestone ADCD-504-9*. Exascale Computing Project, Apr 2020.
12. L.V.G. Vergara, J. Wayne, M.G. Lopez, and O Hernández. Early Experiences Writing Performance Portable OpenMP 4 Codes. In *Proceedings of Cray User Group Meeting, London, England*. Cray User Group, 2016.
13. R. Gayatri, C. Yang, T. Kurth, and J. Deslippe. A Case Study for Performance Portability Using OpenMP 4.5. In *International Workshop on Accelerator Programming Using Directives, 2018*, pages 75–95. Springer, 2018.
14. S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.
15. C. Yang, T. Kurth, and S. Williams. Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system. *Concurrency and Computation: Practice and Experience*, page e5547, 2019.
16. E. Konstantinidis and Cotronis Y. A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. *Journal of Parallel and Distributed Computing*, 107:37–56, 2017.
17. N. Ding and S. Williams. An Instruction Roofline Model for GPUs. In *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 7–18, 2019.

18. C. Yang, R. Gayatri, T. Kurth, P. Basu, Z. Ronaghi, A. Adetokunbo, B. Friesen, B. Cook, D. Doerfler, L. Oliker, J. Deslippe, and S. Williams. An Empirical Roofline Methodology for Quantitatively Assessing Performance Portability. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 14–23, 2018.