2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)

Evaluating the Impact of Proposed OpenMP 5.0 Features on Performance, Portability and Productivity

S. J. Pennycook, J. D. Sewall, J. R. Hammond
Intel Corporation
Santa Clara, CA 95054
Email: {john.pennycook, jason.sewall, jeff.r.hammond}@intel.com

***Abstract*—We investigate how specialization mechanisms proposed for OpenMP 5.0 – specifically, the `metadirective` and `declare variant` directives – may be deployed in a real-life code, using the miniMD benchmark from the Mantevo suite.**

Additionally, we develop an OpenMP 4.5 implementation of miniMD that achieves a performance portability of 59.35% across contemporary CPU and GPU hardware, discuss the processes of porting and enabling this code, and show that the use of specialization would enable our code to be expressed in a significantly more compact form, with implications for productivity.

***Index Terms*—high performance computing; performance portability; productivity**

I. Introduction

In our previous work [1], [2], we presented a concise, objective definition for performance portability (and a companion metric, $\mathcal{P}$) based on observed performance efficiencies for applications on different platforms. Therein, the concept of an "application" was deliberately left vague; for example, we made no restriction that a common codebase be used. The implication of this is that it is equally valid to compute a $\mathcal{P}$ for an application built from a single source using a framework like Kokkos [3] as it is to compute a score for an application built with different languages and tools for each platform (*e.g.* OpenMP* and vector intrinsics for CPUs and CUDA* for GPUs). Our intention was both to avoid the ill-defined notions around the mapping of source to executed code and to allow the flexibility to compute scores in a variety of ways.

Naturally, there are many questions that arise from examining a $\mathcal{P}$ score that is the result of such different models of development. Single-source codes may take less effort to write and maintain than multiple versions, but how does their $\mathcal{P}$ score differ? How does one evaluate all of these costs and benefits in a useful way?

Many discussions of performance portability have thus evolved to include a "third P", *productivity*, to capture these ideas. Unfortunately, productivity remains a difficult notion to quantify objectively, and this paper is not aimed at offering a definition or metric for "productive performance portability". Rather, we evaluate a solution proposed in [1] – fine-grained specialization for different architectures – and offer a preliminary evaluation of the `declare variant` and `metadirective` features proposed for OpenMP 5.0 [4].

A fine-grained, expressive specialization mechanism has the potential to offer the best of both extremes in the single- versus disparate-source dilemma mentioned above; the majority of the code can be kept as a single source to ease readability and maintenance, while allowing for specific optimizations to be deployed for performance on different platforms with a much finer granularity, and with greater ease, than would be otherwise tractable.

Furthermore, a great deal of code introduced in order to support a specific platform may be temporary in nature: workarounds for specific compiler versions or architecture bugs; hard-coded "magic constants" for optimizations such as cache-blocking; or features introduced to leverage dedicated instructions and hardware available on specific devices. Being able to identify and remove code that is no longer required (*e.g.* because the tools have improved, or the platforms are no longer relevant) is as much a productivity concern as the ability to insert new code.

It is one thing to discuss these topics at an abstract level, another to look at in the form of concrete mechanisms, and another still to see how such mechanisms may be used in a real code. Building on our work in [5], [6] and [7], we investigate how a well-known molecular dynamics proxy benchmark – miniMD – can benefit from the specialization tools proposed in OpenMP Technical Report 7 (TR7) [4] for inclusion in OpenMP 5.0.

The specific contributions of this paper are as follows:

1) We develop a version of the miniMD benchmark that utilizes OpenMP 4.5 features to execute on both CPUs and GPUs, and measure its performance portability.
2) We provide an overview of features proposed in OpenMP TR7 and discuss their potential impact on developer productivity.
3) We evaluate the utility of two proposed OpenMP features – specifically `metadirective` and `declare variant` – to reduce the complexity of our OpenMP 4.5 version of miniMD.

II. Related Work

The challenge of developing applications that achieve a high level of performance and portability across a diverse set of hardware designs has led to the creation of programming

978-1-7281-0220-7/18/$31.00 ©2018 IEEE
DOI 10.1109/P3HPC.2018.00007

frameworks that provide an abstraction of the underlying device (*e.g.* Kokkos [3], RAJA [8], SYCL [9]). Many of these frameworks are developed in C++, and rely on templated loop constructs to hide the complexity of swapping out different backends (which may or may not leverage different programming languages) for different devices. This "single-source" approach to performance portability is an attractive one, and has garnered support from many developers [10], [11], [12].

Using C++ may not be an option for the maintainers of legacy C or Fortran codes, and so more recently there have been a number of case studies that have investigated the use of directive-based models to program offload devices. Many researchers have already evaluated the use of OpenACC* to target NVIDIA* GPUs [13], [14], [15], [16] and there is growing interest in the use of OpenMP to target the same hardware. Bertolli et al. [17], [18], [19], [20] at IBM* have developed an llvm+clang-based prototype OpenMP offload compiler for NVIDIA GPUs which, although still in development, has delivered several promising results to date [21], [22], [23], and much of their work has been upstreamed into clang. A similar prototype is under development for gcc, and support for GPU offloading is also available in the Cray* compiler [24]. Supporting OpenMP on NVIDIA GPUs requires a mapping of OpenMP concepts and terminology onto that of CUDA; the exact nature of this mapping may differ between compilers, and differences between the SIMD and SIMT programming models may require different pragmas to be selected across host and device [25]. Traditional OpenMP applications for CPU typically use a two-level hierarchy of `parallel` (threads) and `simd` (vector) regions, whereas a different two-level hierarchy of `teams` (CUDA "blocks") and `parallel` (CUDA "threads") is currently recommended for NVIDIA GPUs; this naturally leads to questions about portability.

Many of the proposed OpenMP 5.0 features discussed in this paper have appeared in some form before appearing in TR7: some have been part of other programming languages [26]; others have been prototyped as vendor compiler extensions [27]; and many have been the subject of research publications [7], [28], [29]. To our knowledge, this is the first paper to investigate how these features may interact with one another in the context of a real application.

We previously developed a version of miniMD optimized for Intel® Xeon® processors and Intel® Xeon Phi™ co-processors [5], focusing on improving the code's thread and vector parallelism. This optimized version was subsequently used to evaluate the performance portability of another version of miniMD written in OpenCL [6], capable of executing on CPUs and GPUs from a number of vendors with a worst-case application efficiency of approximately 50%. This paper builds on these studies using more modern programming languages, tools and architectures.

```
// Run in parallel only when running on an offload device.
#pragma omp metadirective \
 when(device={kind(nohost)} : parallel for)
for (int i = 0; i < N; ++i)
{
  // Update atomically only when running in parallel.
  #pragma omp metadirective \
   when(construct={parallel} : atomic)
  array[i]++;
}
```

Fig. 1. An example usage of `metadirective` functionality.

III. Background

A. miniMD

miniMD is a molecular dynamics benchmark from the Mantevo benchmark suite [30], acting as a proxy for LAMMPS [31], [32]. As a proxy, the code is missing several features found in production molecular dynamics codes: it supports only short-range force calculations using the Lennard-Jones potential or Embedded Atom Method (EAM), and supports a limited number of input configurations. We refer the reader to the code's `README` for a complete breakdown of its strengths and weaknesses as a benchmark.

We selected miniMD for this study not as a representative of molecular dynamics codes, but as a representative of codes with multiple complex paths – specifically, codes where high performance is achieved by selecting different algorithms based on a combination of input parameters and features of the target device. At the time of writing, the current release of miniMD includes four different implementations of its short-range force calculation, each targeting CPUs: a serial implementation with no OpenMP ("original"); a serial implementation using OpenMP SIMD pragmas to force vectorization ("halfneigh"); a threaded implementation using OpenMP that cannot be vectorized due to its use of atomic operations ("halfneigh_threaded"); and a threaded implementation using OpenMP that computes the force between every pair of atoms twice, thus avoiding the need for atomics and ensuring vectorization ("fullneigh"). We add to this a fifth implementation using OpenMP that makes use of privatization in place of atomics ("halfneigh_threaded_privatization") based on our prior work [5].

B. OpenMP 5.0

This paper evaluates features proposed in TR7 [4] for inclusion in OpenMP 5.0. These features may or may not be part of OpenMP 5.0 when the standard is finalized, and at the time of writing there are no compliant compilers and/or runtimes that are publicly available. However, the features that we focus on here are largely productivity improvements, representing straightforward compiler transformations that are possible (but arduous) to implement by hand. Our evaluation of the syntax and semantics of these features concerns whether they are capable of expressing common patterns encountered while porting codes, rather than their direct impact on performance.

The first feature we consider is `metadirective`, which allows different directives to be used in different contexts.

```
// Base version of rsqrt compatible with all devices.
float rsqrt(float x)
{
  return 1.0f / x;
}
...
// Variant of rsqrt.
// Can only be called from a simd region.
// Requires the device to support AVX-512F instructions.
#pragma omp declare variant(rsqrt) \
 match(construct={simd}, device={isa(avx512f)})
__m512 _mm512_rsqrt14_ps(__m512 x);
...
#pragma omp simd
for (int i = 0; i < N; ++i)
{
  // Compiler chooses an appropriate call to rsqrt
  // during auto-vectorization.
  output[i] = rsqrt(input[i]);
}
```

Fig. 2. An example usage of `declare variant` functionality.

```
// Implementation guarantees host and device share memory.
#pragma omp requires \
 unified_address \
 unified_shared_memory
...
float *a = (float*) omp_target_alloc(N * sizeof(float));
float *b = (float*) omp_target_alloc(N * sizeof(float));
float *c = (float*) omp_target_alloc(N * sizeof(float));
...
// Host can access device memory without explicit transfers.
for (int i = 0; i < N; ++i)
{
  a[i] = 1.0f;
  b[i] = 2.0f;
}
...
// Device can access memory without is_device_ptr clauses.
#pragma omp target
for (int i = 0; i < N; ++i)
{
  c[i] = a[i] + b[i];
}
...
omp_target_free(c);
omp_target_free(b);
omp_target_free(a);
```

Fig. 3. An example usage of `requires` functionality.

`metadirective` is similar in many ways to certain C preprocessor directives (*e.g.* `#ifdef`), but is more expressive due to its ability to query the current OpenMP context: for example, the code in Figure 1 shows how two independent instances of `metadirective` can work together to control whether a loop is executed in parallel on certain devices and subsequently whether updates within the loop need to be atomic.

The second feature we consider is `declare variant`, which enables `metadirective`-like functionality but for function dispatch. This is necessary because there are certain transformations (*e.g.* vectorization of `simd` loops and functions) that the user cannot interact with via the preprocessor. For example, the code in Figure 2 requests that a function call (to `rsqrt`) be replaced by some device-specific vector intrinsics; the user is unable to call such intrinsics within the `simd` loop manually, since there is no representation of the vector register containing the input values in user code, and the precise definition of such a register is unknown until after vectorization has taken place. The type signature of a variant for a particular instruction set is implementation-defined, and it may be necessary to provide additional variants if substitution is desired inside of conditionals and/or vector peel/remainder loops.

The third feature we consider is `requires`, which enables OpenMP compilers to behave differently given certain assertions from the developer. For example, the code in Figure 3 shows how `requires` can be used to assert that the code will only be executed on systems where the host and device have unified address and memory spaces, overriding the default requirements and behaviors of `target` data constructs. The `requires` directive can also specify that code use optional OpenMP features (*e.g.* dynamic memory allocators inside of `target` regions) or vendor extensions.

Taken together, these three features provide a *prescriptive* approach to performance portability: they provide users with a set of tools for managing divergent code paths when different devices are known to prefer different expressions of the same algorithm – or even different algorithms altogether. TR7 also introduces a *descriptive* complement to this approach in the form of the `loop` directive and its associated `independent` clause, which leave all decisions regarding how a given loop should be compiled for and executed on a particular device in the hands of the implementation. We do not consider the `loop` directive in this work for two reasons: firstly, the lack of control over scheduling is incompatible with our intention to evaluate different algorithms on different devices; and secondly, since the behavior of the clause is implementation-defined, we cannot evaluate its impact without an implementation.

IV. Implementation

A. OpenMP 4.5 Offload

We were unable to find a version of miniMD using OpenMP's offload (*i.e.* `target`) directives to target GPUs, and therefore performed the necessary porting as a precursor to our OpenMP 5.0 study. The result closely mirrors the Kokkos implementation: application-level OpenMP parallel regions are pushed down to the level of individual loops in order to form kernels suitable for offloading; and functionality that is not supported by many offload devices (*e.g.* global synchronization, dynamic allocation of memory) is shifted back to the host.

1) Host-Device Transfers: As in the Kokkos version, we achieve high performance by offloading *all* computation to the device in order to minimize host-device transfers: the only transfers that occur are those required to transfer initial data to the device, or those introduced by the compiler upon transition to and from target regions. Additional transfers would be required either side of an MPI communication routine – these are present in the code, but we do not explore the behavior of multi-node execution in this paper.

```
static void *mmd_alloc(size_t bytes)
{
  char *h_ptr = (char*) malloc(bytes);
#ifdef USE_OFFLOAD
  char *d_ptr = (char*) omp_target_alloc(bytes, 0);
  omp_target_associate_ptr(h_ptr, d_ptr, bytes, 0, 0);
#endif
  return (void *) h_ptr;
}

static void mmd_free(void *ptr)
{
  if (ptr)
  {
#ifdef USE_OFFLOAD
    omp_target_disassociate_ptr(ptr, 0);
#endif
    free(ptr);
  }
}
```

Fig. 4. The wrapper used to mirror host and device allocations.

Despite our efforts to minimize transfers, inspecting the output of the GPU profiling tool `nvprof` reveals a large number of transfers to and from the device, each four bytes in length. This represents local variables referenced in target regions being automatically copied back and forth by the OpenMP runtime. While maintaining such consistency is important, one can imagine more efficient behaviors, such as identifying transfers that need not be applied repeatedly or aggregation.

2) Memory Allocation: Since all computation is offloaded to the device, we effectively mirror all host memory allocations there. To automate this, we introduce an allocation wrapper – shown in Figure 4 – which also helps to minimize the divergence between host and device code. Note that our wrapper uses API calls to associate and disassociate the pointers manually. An earlier implementation used the corresponding directives – `target enter data map(alloc:A[N])` and `target exit data map(delete:A)` – but we were surprised to find that the directives generally required the allocation size at deallocation for correctness. The specification is not abundantly clear, but our reading is that this should not be necessary. This is more than a curiosity; it is not customary to need to specify sizes in most deallocation routines found in contemporary languages (*e.g.* glibc's `free()`, `delete` in C++).

It is currently challenging to efficiently map paradigms like C's `realloc()` to OpenMP offload devices. On the host side, this routine conceptually grows allocations and allows the runtime to avoid copies by growing heap entries when possible. It is not possible to express this on the device, and this reflects the broader challenge of maintaining and using two heaps in different environments simultaneously. The `mmd_realloc` function provided by our wrapper (not shown) emulates this functionality on the device by transferring the memory's contents to the host, performing a reallocation of the host memory, and transferring the contents of the reallocated memory to a new (larger) allocation on the device.

3) Compiler Issues and Workarounds: We use a recent llvm+clang compiler toolchain with `libomptarget` to offload to NVIDIA GPUs. This is still in active development, with many features the result of upstreaming from work on the clang-ykt prototype [33], and we encountered several issues that required workarounds. The impact of these workarounds on the code is varied; some were merely cosmetic, while others impact performance. It should be noted that these observations are the result of our experimentations with the compiler and runtime during our porting and optimization efforts, and we acknowledge that they are not conclusive.

The release notes make it clear that `reduction` across teams is not supported, which we work around by using reductions within teams (on `parallel for` statements) in combination with atomic updates to global variables by the master thread of each team. This likely incurs some performance penalty relative to a structured reduction across thread blocks, but its impact here is small: the reductions are required only for the energy/virial updates, which are both scalar and infrequently called.

However, reliance on atomics to implement this workaround was complicated by uneven support on the device side; with the `nvptx64` target, using `omp atomic` on a floating-point addition would generate a compilation error. Workarounds using loops built on compare-and-swap intrinsics worked, but at the expense of performance (in the case of hardware with support for atomic floating-point reductions). Inside `parallel` regions within a `teams` region on a GPU, these compare-and-swap intrinsics would cause memory errors unless they were lexically inlined by hand or with a pre-processor macro.

In many cases, compilations of certain translation units would fail because the whole program is compiled for both the host and the target without regard for the applicability to the hardware in question. For example, host code with inline assembly fails to compile because it cannot be used on the GPU. Appropriately guarding user code is a suitable workaround, but one that is hard to employ when the assembly appears in standard headers (*e.g.* for common math functions).

Finally, we found that it was necessary to make local copies of class data members for them to be copied correctly to the device; this was true both for items implicitly qualified by `this` in member functions, and for data members of classes used in target regions.

4) Debugging: During our porting efforts we identified several best-practices for debugging offload code using the `nvptx` runtime. Although not strictly implementation details, we include them here for posterity.

Compiling with the offload device set to `x86_64` is of little value when it comes to performance, but is a great boon when debugging offload semantics and the behavior of `teams`, `distribute`, and other device-centric directives. It can also be useful in identifying whether a behavior is inherent to clang's offload capabilities, or to the `nvptx` backend in particular.

For both backends, we observed that the runtime was very eager to fall back to the host in the event of errors on the device. It was often necessary to run external profiling tools, or

```
// Enable offload based on compile-time constant.
// Desirable to avoid using "teams" on host.
#pragma omp metadirective \
 when(user={condition(USE_OFFLOAD)} : \
      target teams distribute) \
 when(device={kind(host)} : parallel for)
for (int i = 0; i < nlocal; ++i)
{
  ...
  // Parallelize inner loop based on outer loop construct.
  // Note the repeated reduction() clause.
  #pragma omp metadirective \
     when(construct={target} : parallel for \
          reduction(+:fix, fiy, fiz, t_eng_vdwl, t_virial)) \
     when(construct={parallel} : simd \
          reduction(+:fix, fiy, fiz, t_eng_vdwl, t_virial))
  for (int k = 0; k < numneighs; ++k)
  {
    ...
  }
}
```

Fig. 5. An example usage of `metadirective` within miniMD.

to query `omp_is_initial_device()` regularly, to ensure that the device was actually being used.

B. OpenMP 5 Features

Due to their novelty, there are few examples of the specialization-focused directives proposed for OpenMP 5.0 available, and those are of small scale [7]. Our intention in working with the mature codebase found in miniMD is to mimic the experience of developers with legacy codebases; the code is functional and was designed – some time ago, by a different developer – to be decomposed into a certain pattern of classes and functions.

This is relevant because of its influence on the approach taken in refactoring: a developer is faced with a choice of making minimal changes to the code's existing code structure and relying heavily on `metadirective` or making more significant changes to refactor loops into standalone functions that are amenable to specialization with `variant`.

Interestingly, our experience suggests that any instance of `metadirective` can be replaced by a suitable use of `variant`, but not vice versa. Converting a `metadirective` to a `variant` consists of converting the scoped block affected by the directive into a function, and creating a function variant for each directive variant; each `when` clause corresponds to a `match` clause, and the `default` clause to the base function. Reversing the transformation is possible in trivial cases, but not in cases where each variant differs by more than simple directive changes.

1) Metadirective: The `metadirective` is highly effective in cases where the only change(s) necessary are directly tied to the loop schedule. For example, selecting between `target teams distribute` on outer loops and `parallel for` on inner loops (for GPUs) and `parallel for` on outer loops and `simd` on inner loops (for CPUs) is a common pattern that can be handled this way – see Figure 5. We also found the `user` selector to express directives as a function of compile-time constants (such as template parameters) to be very useful.

```
// Base version that uses atomics to guard updates
// of neighboring atoms.
void neighbor_force_update(MMD_float* f,
                           int j,
                           MMD_float fx,
                           MMD_float fy,
                           MMD_float fz)
{
  #pragma omp atomic
  f[j * PAD + 0] -= fx;

  #pragma omp atomic
  f[j * PAD + 1] -= fy;

  #pragma omp atomic
  f[j * PAD + 2] -= fz;
}

// Variant specialized for privatized implementations
// Assumes privatization will always be used on the host.
#pragma omp declare variant(...) \
 match(device={kind(host)})
void private_neighbor_force_update(MMD_float* f,
                                   int j,
                                   MMD_float fx,
                                   MMD_float fy,
                                   MMD_float fz)
{
  f[j * PAD + 0] -= fx;
  f[j * PAD + 1] -= fy;
  f[j * PAD + 2] -= fz;
}

// Variant specialized for privatized implementations
// executing on Intel(R) Xeon Phi(TM) processors.
// Assumes privatization will always be used on the host.
#pragma omp declare variant(...) \
 match(construct={simd(uniform(f), simdlen(16), inbranch)}, \
       device={kind(host), isa(avx512f), arch(knl)}, \
       implementation={vendor(intel)}, \
       user={condition(PAD==4)})
void _mm512_private_neighbor_force_update_ps(MMD_float *f,
                                             __m512i j,
                                             __m512 x,
                                             __m512 y,
                                             __m512 z,
                                             __mmask16 mask)
{
 /* AoS-to-SoA transpose using AVX-512 instrinsics */
}
```

Fig. 6. An example usage of `variant` functionality within miniMD. The type signature of the base function repeated in each variant is omitted.

One issue, however, is that each `metadirective` clause must be a complete, valid directive in its own right – there is no way to conditionally enable or disable only *part* of a directive. In practice this can lead to undesirable code duplication, particularly in light of the length and complexity found in some OpenMP directives. For example, choosing between two directives that include the same reduction requires that the reduction clause be specified each time. It is unclear how to extend the syntax to handle such cases more elegantly.

It is straightforward to use a `metadirective` to enable specific behavior under certain conditions, but *disabling* certain behavior under certain conditions is less so. One strategy to express negation is to employ the `default` clause alongside empty `when` clauses to describe the conditions in which the directive should *not* be generated. However, it is unclear whether this is compliant.

2) Variant: We have found that variants work very well when differences between implementations are self-contained and can be encapsulated in a single function call. This should not be surprising, as this is the focus of the proposal. A good example is the specialization employed to inject vector intrinsics into the force computation; these intrinsics replace independent gathers and scatters for each component of the force with a more efficient instruction sequence that gathers structs and converts their data layout via shuffles [5]. These intrinsics are device-specific and may not provide performance boosts in all cases, so a variant works well – see Figure 6.

Unfortunately the number of such self-contained cases (in miniMD, at least) is small. Many of the broader algorithmic specializations (*e.g.* whether forces should be privatized per thread/team and then reduced, or updated directly via atomics) are spread across several functions. In limited cases it is possible to express such behavior via variants by ensuring that each of these multiple functions is declared as a variant with the same context-selector; however, in general the context-selection mechanism cannot guarantee that another variant is not selected (*e.g.* because a different developer has introduced another compatible variant with a higher $\mathcal{P}$ score).

One potential solution to this requires that the `user` context specifier be extended to include additional *user-defined* traits. For example, in the case above, adding an additional user-defined `algorithm` parameter would prevent alternative implementations from being selected *unless* the developer had declared that they were compatible.

In future work, we would like to investigate how variants may be used in Fortran and in different idioms found in C/C++ codes; the mechanism is highly flexible, and may admit many styles of use.

3) Requires: The lack of explicit transfers in our completed OpenMP 4.5 code means that there are few opportunities for us to evaluate the use of `omp requires`. Introducing this functionality would replace only a handful of explicit transfer directives, and would therefore have very little impact on the size or readability of the code.

However, we believe that being able to use this functionality would have increased our productivity significantly *during development*. Many of the bugs that we introduced during development were a result of a missing transfer or mirrored allocation, resulting in stale data or invalid memory accesses on the device. Furthermore, our initial implementation contained a significant number of transfers either side of each kernel to enable us to develop the offload functionality in an incremental fashion and to divide work between developers. Relying on unified virtual memory and shared memory would have enabled us to focus on the correctness of individual kernels without consideration of dataflow.

It is impossible for us to quantify this improvement retroactively, since we did not track our development effort as suggested by Wienke et al. [34]; the best that we can do is to provide a qualitative assertion that a significant amount of the development time was spent debugging these transfers, and that getting them right was by far the most frustrating part of this exercise.

	Intel® Xeon® Gold 6148 Processor	**NVIDIA* P100**
Platform	CPU	GPU
Sockets	2	1
Cores[a]	20	56
Clock (GHz)	2.4	1.3
Peak GFLOP/s[b]	6144	9300
Peak GB/s[c]	176	551
DRAM (GB)	384	16
Compiler	Intel® C++ Compiler 2018.3.22	clang[d]
MPI	Intel® MPI Library 2018	N/A
Runtime	N/A	CUDA 8.0
Driver	N/A	375.20

[a]Reported for GPUs as the number of streaming multiprocessors.
[b]Single precision peak from vendor-provided hardware specification.
[c]Measured by STREAM [35] or BabelStream [36].
[d]Built from https://github.com/llvm-project/llvm-project-20170507/tree/c784e7 with `LIBOMPTARGET_NVPTX_COMPUTE_CAPABILITIES=60`.

TABLE I
HARDWARE AND SOFTWARE CONFIGURATION.

Implementation	**Compiler Flags**
Original	`-O3 -xHost -fasm-blocks -DMPICH_IGNORE_CXX_SEEK -restrict -qopenmp -DUSE_SIMD -DPAD=4 -DPRECISION=1`
mxhMD	`-O3 -xHost -fno-alias -ipo -restrict`
Kokkos (CPU)	`-O3 -xHost KOKKOS_DEVICES=OpenMP KOKKOS_ARCH=SKL -DPAD=4 -DPRECISION=1`
Kokkos (GPU)	`-O3 KOKKOS_DEVICES=Cuda KOKKOS_ARCH=Pascal60 -DPAD=4 -DPRECISION=1`
OpenMP 4.5 (CPU)	`-O3 -xHost -fasm-blocks -DMPICH_IGNORE_CXX_SEEK -restrict -qopenmp -DUSE_SIMD -DPAD=4 -DPRECISION=1`
OpenMP 4.5 (GPU)	`-O3 -fopenmp -fopenmp-targets= nvptx64-nvidia-cuda -MD -DPAD=4 -DPRECISION=1`

TABLE II
COMPILER FLAGS USED FOR EACH IMPLEMENTATION.

V. RESULTS

A. Experimental Setup

The system configuration used by our experiments is given in Table I. We would liked to have used a more modern NVIDIA GPU (*i.e.* one using the Volta microarchitecture) in our experiments, but we experienced compatibility issues with the `nvptx` backend of `libomptarget` in the development version of clang available at the time of publication, perhaps due to the independent warp scheduling introduced in Volta.

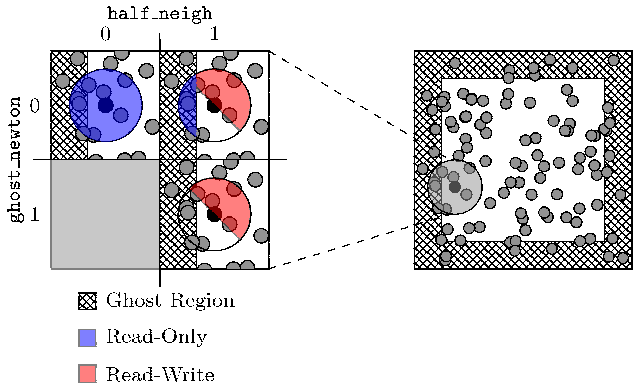

Fig. 7. A two-dimensional representation of a single atom's neighbor list maintained by a thread, for each meaningful combination of the `half_neigh` and `ghost_newton` options. The blue shaded area indicates atoms that only have their data read for force updates to the center atom, while the red shaded area indicates atoms that are read for force updates and that receive the center atom's force contribution. The angle of the red shading indicates the neighbor list construction convention in miniMD.

Implementation	Configuration		App. Eff. (%)	
	half_neigh	ghost_newton	CPU	GPU
Original	0	0	60.54	-
Original	1	0	25.27	-
Original	1	1	25.85	-
mxhMD	0	0	100.00	-
mxhMD	1	1	80.69	-
Kokkos	0	0	35.49	78.12
Kokkos	1	0	26.67	100.00
Kokkos	1	1	27.15	95.69
OpenMP 4.5	0	0	62.82	56.24
OpenMP 4.5	1	0	24.26	5.53
OpenMP 4.5	1	1	26.00	5.49

TABLE III
APPLICATION EFFICIENCY FOR MINIMD'S DEFAULT INPUT PROBLEM.

We compare four different implementations of miniMD: the reference code ("Original"); our CPU-optimized code from [5] ("mxhMD"); the Kokkos version ("Kokkos"); and the offload implementation developed for this paper ("OpenMP 4.5"). Compilation flags specific to each implementation are listed in Table II.

The Kokkos implementation of miniMD is the highest performing and most up-to-date version that exists for NVIDIA GPUs. Although an OpenACC version exists, it is part of the older miniMD 1.2 distribution and has not been maintained. When it was introduced, its authors reported a slow-down of 3× relative to the Kokkos implementation [37], and in our own efforts to compile and use the code we saw worse results still (including some correctness errors); whether this is due to lack of maintenance or some configuration issue on our part is unclear. No CUDA version of miniMD exists; the closest proxy is the (now deprecated) USER-CUDA package from LAMMPS, which (as mentioned previously) includes additional complexity not present in miniMD.

Unless otherwise stated, all performance numbers are collected using miniMD's default configuration: a small problem (131,072 atoms) using the Lennard-Jones potential. All experiments were repeated for larger problem sizes (256,000 atoms and 2,048,000 atoms), but no significant difference was present in the trends.

For readability, the performance analysis in this section focuses on configuring miniMD only by way of its `half_neigh` and `ghost_newton` options. These options control the use of Newton's third law (N3) when building neighbor lists and computing forces between atoms, as shown in Figure 7: when both options are 0, each atom reads the positions of all of its neighbors but updates only its own force; when both options are 1, each atom reads the positions of half of its neighbors and updates their forces; and otherwise each atom only updates neighbors belonging to the same MPI rank. The options thereby trade away redundant computation for parallel force updates, which must be carefully managed with atomics, reductions or MPI communication. Note that the `ghost_newton` option is ignored when `half_neigh` is set to 0, and we therefore exclude this combination from our experiments.

The `half_neigh` and `ghost_newton` options are exposed (in some form) by all of the implementations considered, and are sufficient to demonstrate the complexity inherent in supporting multiple algorithms within a single code. The naïve privatization introduced by our OpenMP 4.5 version is discussed only with regards to productivity (in Section V-D), since it never improves performance.

B. Application Efficiency

Table III compares the application efficiency (*i.e.* the performance of an application on a platform for a given input, relative to the best-known performance achieved on that platform for that input) of each implementation of miniMD executing the default input problem on the CPU and GPU.

The best performing implementations on the CPU and GPU (marked by an application efficiency of 100%) are mxhMD and Kokkos, respectively, as expected. Of the performance-portable implementations, our OpenMP 4.5 version performs best on the CPU (closely matching the original implementation) – unlike with the Kokkos implementation, adding support for the GPU has not degraded performance on the CPU.

Unexpectedly, completely disabling N3 (*i.e.* setting both `half_neigh` and `ghost_newton` to 0) gives the best performance in almost every case: the only exception is the Kokkos implementation on GPU, which sees better performance with `half_neigh` set to 1. It is striking that this same configuration gives the *worst* performance for the OpenMP implementation on GPU, which we attribute to compiler immaturity; exploiting N3 relies on atomics, which we could only use by way of a manual compare-and-swap loop expressed using the `__sync_bool_compare_and_swap` builtin. We expect that clang will eventually support `omp atomic` correctly for floating-point types in the `nvptx` backend, and will generate an instruction sequence closer to that used by CUDA's `atomicAdd`.

Implementation	Application Efficiency (%)		$\mathcal{P}$
	CPU	GPU	
Original	60.54	-	0.00
mxhMD	100.00	-	0.00
Kokkos	35.49	100.00	52.39
OpenMP 4.5	62.82	56.24	59.35

TABLE IV
APPLICATION EFFICIENCY AND PERFORMANCE PORTABILITY FOR MINIMD'S DEFAULT INPUT PROBLEM, WHEN USING THE BEST CONFIGURATION ON EACH PLATFORM.

Implementation	Total SLOC	Workaround SLOC
Original	4,396	-
mxhMD	14,988	-
Kokkos	4,896	-
OpenMP 4.5	5,410	257
OpenMP 5.0	4,936	300

TABLE V
SLOC COUNTS FOR EACH IMPLEMENTATION OF MINIMD.

The remaining differences in performance are due in part to additional specializations that are not currently present in our OpenMP 4.5 version of miniMD, but which could be explored in future work: specifically, mxhMD employs a faster neighbor list build algorithm and uses a force kernel optimized for simulations with identical force parameters for all atom pairings; and the Kokkos implementation employs shared (local) memory and a force kernel optimized for simulations with a small number of atom types.

C. Performance Portability

The results in this section measure performance portability as defined in our previous work [1], [2]:

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

that is, the harmonic mean of application a's performance efficiency when executing problem p on a set of platforms H.

In order to compute $\mathcal{P}$ for an application with multiple algorithms, we first assume that the developer (or the application itself) is smart enough to select the configuration with the highest performance on each platform. Whether this is derived via some heuristic applied to device parameters, hard-coded by the developer or discovered at run-time via auto-tuning is immaterial. The result of this selection and the summarized application efficiencies are shown in Table IV. The Original and mxhMD versions are incapable of executing on the GPU, which is reflected in the final $\mathcal{P}$ scores.

The $\mathcal{P}$ scores for Kokkos and OpenMP 4.5 are very similar, with OpenMP pulling out in front because of its higher application efficiency on the CPU. This is very promising, and suggests that OpenMP is a viable directive-based alternative to frameworks like Kokkos – at least when it comes to performance and portability. However, we note that neither number is as high as might be hoped, and there remains a clear need to incorporate different algorithms and/or improve the tools in order to close the gap between performance-portable and highly-specialized implementations.

D. Productivity

Although there are many problems with using source lines of code (SLOC) or similar (*e.g.* words of code) as a measurement of productivity and/or development effort [34], it is a simple objective metric that has been used by similar productivity studies [38], [39]. Even if SLOC does not reflect missteps in the development process or capture time spent on non-coding tasks, we argue that the size of an application is an important factor in determining maintenance cost. It stands to reason that larger code bases will be harder for new developers to become familiar with and that they may be harder to modify substantially – one of the reasons that benchmarks like miniMD exist in the first place is that their "parent" codes are too large for exploratory optimization studies [30].

In order to ensure that we count SLOC in a consistent manner, all source codes are formatted with `clang-tidy` before SLOC data are generated using David A. Wheeler's `SLOCCount` tool [40]. The results are shown for each implementation in Table V.

With the exception of mxhMD, the code bases are all of approximately the same size – especially when lines attributed to workarounds for offload are removed from our new OpenMP versions. The mxhMD source includes a lot of code that is ultimately not used here, including separate implementations of multiple functions in SSE and AVX intrinsics; this is a clear case of a code that has been developed for performance rather than portability.

The OpenMP 5.0 version contains 34 instances of `metadirective` (each of which is typically 3 lines) and 8 instances of `declare variant` (each of which is typically 1 line). Including the bodies of the variants, the amount of code specialized for different devices amounts to only 141 lines. To make a more direct comparison: introducing privatization by duplicating the entire force computation function requires 157 lines, whereas introducing the same functionality through variants and metadirectives requires only 66 lines. Although a similarly sized solution could also have been written using C++ templates, the fact that it can be expressed using directives alone is a boon for C and Fortran programmers.

VI. CONCLUSION

The offload features present in OpenMP 4.5 guarantee functional portability of an application across CPUs and GPUs, which is key to obtaining positive performance portability scores as measured by $\mathcal{P}$. As demonstrated in this paper, the level of performance that can be achieved using OpenMP – and subsequent performance portability scores – is comparable and may even surpass that achieved by other performance portability frameworks like Kokkos. More detailed investigation of

the remaining performance differences across all platforms is required if we hope to improve these Φ scores further.

One of the biggest usability gaps between OpenMP 4.5 and its alternatives lies in its reliance on functionality outside of OpenMP (*e.g.* pre-processors) to implement common patterns that execute different code on different devices. The `metadirective` and `declare variant` features proposed in OpenMP TR7 close this gap significantly and are sufficient to express several common idioms; however, complications arise when device-specific behaviors span multiple functions and it may be necessary to refactor existing codebases before these features can be used effectively.

A. Future Work

The introduction of new OpenMP functionality provides both a need and opportunity to develop new tools. `metadirective` and `declare variant` choose between functionality based on "context-selectors", and tools that help to identify and debug the interaction of context-selectors with the compiler's contextual information are required if developers are to adopt a fine-grained specialization approach in production codes.

Since these directives already demarcate sections of code that may be device-specific, we believe that they are a good starting point for the development of source analysis tools that measure divergence across platforms. Such tools could provide more insight, more efficiently, than the manual investigation undertaken in Section V-D.

Runtime-level dispatch for specialization mechanisms is something that would be broadly useful for expressibility and productivity, but it is far from clear how to enable this functionality without serious performance ramifications; this is an area of research that may offer a large return on investment.

ACKNOWLEDGEMENTS

The authors wish to thank Alex Duran for his help in defining the `declare variant` directive; Carlo Bertolli and Doru Bercea for their help in configuring clang for offload to NVIDIA GPUs; and the anonymous reviewers for their suggestions on how to improve the paper.

DISCLAIMERS

REFERENCES

[1] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "A Metric for Performance Portability," *CoRR*, vol. abs/1611.07409, 2016. [Online]. Available: http://arxiv.org/abs/1611.07409

[2] ——, "Implications of a Metric for Performance Portability," *Future Generation Computer Systems*, aug 2017. [Online]. Available: https://doi.org/10.1016/j.future.2017.08.007

[3] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling Manycore Performance Portability Through Polymorphic Memory Access Patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[4] OpenMP Architecture Review Board, *OpenMP Technical Report 7: Version 5.0 Public Comment Draft*, July 2018. [Online]. Available: https://www.openmp.org/wp-content/uploads/openmp-TR7.pdf

[5] S. J. Pennycook, C. J. Hughes, M. Smelyanskiy, and S. A. Jarvis, "Exploring SIMD for Molecular Dynamics, Using Intel Xeon Processors and Intel Xeon Phi Coprocessors," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1085–1097. [Online]. Available: https://doi.org/10.1109/IPDPS.2013.44

[6] S. J. Pennycook and S. A. Jarvis, "Developing Performance-Portable Molecular Dynamics Kernels in OpenCL," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov 2012, pp. 386–395.

[7] S. J. Pennycook, J. D. Sewall, and A. Duran, "Supporting Function Variants in OpenMP," in *Evolving OpenMP for Evolving Architectures*, B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, and J. Labarta, Eds. Cham: Springer International Publishing, 2018, pp. 128–142.

[8] R. D. Hornung and J. A. Keasler, "The RAJA Portability Layer: Overview and Status," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-661403, September 2014.

[9] Khronos OpenCL Working Group, *SYCL Specification, Version 1.2.1*, July 2018. [Online]. Available: https://www.khronos.org/sycl

[10] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and S. A. Jarvis, "Achieving Performance Portability for a Heat Conduction Solver Mini-Application on Modern Multi-core Systems," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2017, pp. 834–841.

[11] M. Martineau, S. McIntosh-Smith, and W. P. Gaudin, "Assessing the Performance Portability of Modern Parallel Programming Models using TeaLeaf," *Concurrency and Computation: Practice and Experience*, vol. 29, 2017.

[12] A. Afzal, C. Schmitt, S. Alhaddad, Y. Grynko, J. Teich, J. Förstner, and F. Hannig, "Solving Maxwell's Equations with Modern C++ and SYCL: A Case Study," in *29th Annual IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, July 2018.

[13] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 136–143.

[14] A. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter, "Evaluating Performance Portability of OpenACC," in *Languages and Compilers for Parallel Computing*, J. Brodman and P. Tu, Eds. Cham: Springer International Publishing, 2015, pp. 51–66.

[15] J. M. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3D into an Exascale Application Using OpenACC: An Approach for Moving to Multi-petaflops and Beyond," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 15:1–15:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389017

[16] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC — First Experiences with Real-World Applications," in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 859–870.

[17] C. Bertolli, S. F. Antao, A. E. Eichenberger, K. O'Brien, Z. Sura, A. C. Jacob, T. Chen, and O. Sallenave, "Coordinating GPU Threads for OpenMP 4.0 in LLVM," in *Proceedings of the 2014 LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 12–21. [Online]. Available: http://dx.doi.org/10.1109/LLVM-HPC.2014.10

[18] C. Bertolli, S. F. Antao, G.-T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, and K. O'Brien, "Integrating GPU Support for OpenMP Offloading Directives into Clang," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: ACM, 2015, pp. 5:1–5:11. [Online]. Available: http://doi.acm.org/10.1145/2833157.2833161

[19] S. F. Antao, A. Bataev, A. C. Jacob, G. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, T. Chen, H. Sung, C. Bertolli, and K. O'Brien, "Offloading Support for OpenMP in Clang and LLVM," in *2016 Third Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*, Nov 2016, pp. 1–11.

[20] G.-T. Bercea, C. Bertolli, A. C. Jacob, A. Eichenberger, A. Bataev, G. Rokos, H. Sung, T. Chen, and K. O'Brien, "Implementing Implicit OpenMP Data Sharing on GPUs," in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM-HPC'17. New York, NY, USA: ACM, 2017, pp. 5:1–5:12. [Online]. Available: http://doi.acm.org/10.1145/3148173.3148189

[21] M. Martineau, S. McIntosh-Smith, C. Bertolli, A. C. Jacob, S. F. Antão, A. E. Eichenberger, G. Bercea, T. Chen, T. Jin, K. O'Brien, G. Rokos, H. Sung, and Z. Sura, "Performance Analysis and Optimization of Clang's OpenMP 4.5 GPU Support," in *7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, PMBS@SC 2016, Salt Lake, UT, USA, November 14, 2016*, 2016, pp. 54–64. [Online]. Available: https://doi.org/10.1109/PMBS.2016.011

[22] I. Karlin, T. Scogland, A. C. Jacob, S. F. Antão, G. Bercea, C. Bertolli, B. R. de Supinski, E. W. Draeger, A. E. Eichenberger, J. Glosli, H. Jones, A. Kunen, D. Poliakoff, and D. F. Richards, "Early Experiences Porting Three Applications to OpenMP 4.5," in *OpenMP: Memory, Devices, and Tasks - 12th International Workshop on OpenMP, IWOMP 2016, Nara, Japan, October 5-7, 2016, Proceedings*, 2016, pp. 281–292. [Online]. Available: https://doi.org/10.1007/978-3-319-45550-1_20

[23] M. Martineau and S. McIntosh-Smith, "The Productivity, Portability and Performance of OpenMP 4.5 for Scientific Applications Targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs," in *Scaling OpenMP for Exascale Performance and Portability*, B. R. de Supinski, S. L. Olivier, C. Terboven, B. M. Chapman, and M. S. Müller, Eds. Cham: Springer International Publishing, 2017, pp. 185–200.

[24] V. V. Larrea, W. Joubert, M. G. Lopez, and O. Hernandez, "Early Experiences Writing Performance Portable OpenMP 4 Codes," in *Cray Users Group (CUG)*, 2016.

[25] J. Larkin, "Early Results of OpenMP 4.5 Portability on NVIDIA GPUs & CPUs," August 2017, DOE CoE Performance Portability Workshop 2017.

[26] A. Duran, E. Ayguade, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures," *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011.

[27] Intel Corporation, "`vector variant`." [Online]. Available: https://software.intel.com/en-us/node/523350

[28] J. Lee, F. Petrogalli, G. Hunter, and M. Sato, "Extending OpenMP SIMD Support for Target Specific Code and Application to ARM SVE," in *Scaling OpenMP for Exascale Performance and Portability (IWOMP 2017)*. Springer, 2017, pp. 62–74.

[29] J. D. Sewall, S. J. Pennycook, A. Duran, X. Tian, and R. Narayanaswamy, "A Modern Memory Management System for OpenMP," in *Proceedings of the Third International Workshop on Accelerator Programming Using Directives*, ser. WACCPD '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 25–35.

[30] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-Applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, September 2009.

[31] S. Plimpton, "Fast Parallel Algorithms for Short-range Molecular Dynamics," *J. Comput. Phys.*, vol. 117, no. 1, pp. 1–19, Mar. 1995. [Online]. Available: http://dx.doi.org/10.1006/jcph.1995.1039

[32] S. Plimpton *et al.*, "LAMMPS Molecular Dynamics Simulator," http://lammps.sandia.gov/, August 2018.

[33] "clang-ykt / openmp / libomptarget," retrieved August 31, 2018. [Online]. Available: https://github.com/clang-ykt/openmp/tree/master/libomptarget

[34] S. Wienke, J. Miller, M. Schulz, and M. S. Müller, "Development Effort Estimation in HPC," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 10:1–10:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014918

[35] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, December 1995.

[36] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "GPU-STREAM v2.0: Benchmarking the Achievable Memory Bandwidth of Many-Core Processors Across Diverse Parallel Programming Models," in *High Performance Computing: ISC High Performance 2016 International Workshops*, ser. Lecture Notes in Computer Science, M. Taufer, B. Mohr, and J. Kunkel, Eds. Springer, 5 2016.

[37] C. Trott, "OpenACC and C++: An Application Perspective," Sandia National Laboratories, Tech. Rep. SAND2015-2057 C, 2015.

[38] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Accelerating Hydrocodes with OpenACC, OpenCL and CUDA," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov 2012, pp. 465–471.

[39] A. Tabuchi, M. Nakao, H. Murai, T. Boku, and M. Sato, "Performance Evaluation for a Hydrodynamics Application in XcalableACC PGAS Language for Accelerated Clusters," in *Proceedings of Workshops of HPC Asia*, ser. HPC Asia '18. New York, NY, USA: ACM, 2018, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/3176364.3176365

[40] D. A. Wheeler, "More than a Gigabuck: Estimating GNU/Linux's Size," 2001. [Online]. Available: https://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html