Heterogeneous Programming for the Homogeneous Majority

Tom Deakin*, James Cownie, Wei-Chen Lin and Simon McIntosh-Smith
Department of Computer Science
University of Bristol, Bristol, UK
*Email: tom.deakin@bristol.ac.uk

***Abstract*—In order to take advantage of the burgeoning diversity in processors at the frontier of supercomputing, the HPC community is migrating and improving codes to utilise heterogeneous nodes, where accelerators, principally GPUs, are highly prevalent in top-tier supercomputer designs. Programs therefore need to embrace at least some of the complexities of heterogeneous architectures. Parallel programming models have evolved to express heterogeneous paradigms whilst providing mechanisms for writing portable, performant programs. History shows that technologies first introduced at the frontier percolate down to local workhorse systems. However, we expect there will always be a mix of systems, some heterogeneous, but some remaining as homogeneous CPU systems. Thus it is important to ensure codes adapted for heterogeneous systems continue to run efficiently on CPUs. In this study, we explore how well widely used heterogeneous programming models perform on CPU-only platforms, and survey the performance portability they offer on the latest CPU architectures.**

***Index Terms*—CPU, Heterogeneous programming models, Kokkos, OpenMP, Performance portability, SYCL**

I. Introduction

Heterogeneous systems are becoming crucial workhorses in our search for increased computational performance. In the most recent release of the Top 500 ranking of supercomputers' performance running the LINPACK benchmark, it is clear that to get to the highest levels of floating-point performance an accelerator is required [1]; indeed all but one of the Top 10 systems (as of June 2022) uses some form of accelerator, with the majority (seven) using a GPU.

In order for our high-performance programs to survive long into the future, we need them to run on these heterogeneous systems. However the systems show variations in design, compounding the problem. As we also begin to explore how we can take advantage of the technologies in the Cloud to run our HPC codes, we see an even greater diversity in processor technology, and a faster cadence of access to the latest technologies compared with on-premises clusters whose lifetime is normally well over three years.

Our scientific simulation codes must embrace this heterogeneity of accelerators and processors. This need has led to the development of multiple heterogeneous parallel programming models and abstraction layers, both as open industry-led standards and open-source projects led by research laboratories and academic institutions. These models all aim to provide some abstraction over the nature of underlying hardware, giving varying levels of control over platform management ("What devices do I have?"), heterogeneous compute ("How can I express what my parallel and/or concurrency work is?"), and heterogeneous memory ("How can I manage memory locality?"). Performance Portability has never been so important, as rewriting and optimizing code for different machines is expensive and time consuming.

However, supercomputers using heterogeneous node architectures (those with acceleratros of some sort) are not the only technology for attaining the highest levels of performance. The Fugaku supercomputer uses an entirely homogeneous, CPU-only, node design, and was ranked 2nd in June 2022's Top 500 list [1], having held the number one spot since June 2020. Importantly, although most of the Top 10 machines use accelerators, that is not true for machines lower down the list.

Figure 1 shows the breakdown of systems in the June 2022 Top 500 list by accelerator technology, counted by accelerator vendor. It is obvious that the majority of supercomputers do not yet use accelerators. Although 61% of the aggregate performance in the list is provided by accelerated systems, 66% of the systems do not use them. It is clear that preparing to use accelerators is vital for current and near-future supercomputer designs. However, codes need to continue to run well on the majority of CPU-only systems that are available today whilst getting ready for heterogeneity. Therefore we need to write heterogeneous programs that will run well on the accelerators which the community will have access to soon, whilst maintaining performance efficiency on the homogeneous, CPU-only, systems that we use today, and may still use tomorrow as they will be needed to run the long tail of codes which may never be ported to heterogeneous technologies.

The heterogeneous programming models and abstractions give us ways to target both the CPU and GPU with our programs. But our programs are being developed increasingly with a heterogeneous design. We are writing into our programs knowledge of separate execution and memory spaces. This has even more impact on the way we write codes than NUMA-aware implementations did for multi-socket systems.

Fortran remains an important language for scientific codes,

 This work was in part funded by EPSRC through the Strategic Partnership in Computational Science for Advanced Simulation and Modelling of Engineering Systems (ASiMoV) project, EP/S005072/1.

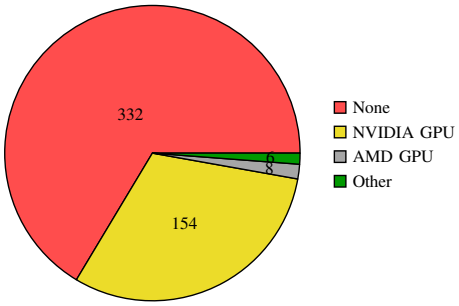

Fig. 1. Number of Top 500 supercomputers with accelerators, according to the June 2022 list [1].

TABLE I
PROGRAMMING LANGUAGES OF THE MOST-USED CODES ON ARCHER2, THE UK'S NATIONAL SUPERCOMPUTER FOR JULY 2022

Code	**Language**	**Percentage of total machine**
VASP	Fortran	25.6%
GROMACS	C++	6.9%
CP2K	Fortran	6.0%
CASTEP	Fortran	4.0%
SENGA	Fortran	3.2%
Fortran total (top 5)		38.8%
C++ total (top 5)		6.9%
Top 5 grand total		45.7%

for instance the machine usage statistics for the top 5 codes on ARCHER2 (the UK's largest centrally funded HPC machine)[2] are shown in I. You can see that just these top 5 codes, use close to 40% of the machine to run Fortran. If these top 5 are representative of all the codes on the machine, then over 80% of the machine is consumed by Fortran code.

For Fortran, OpenMP®remains the most important technique for exploiting CPU-level thread parallelism and heterogeneous, offload, parallelism. It is important to ask whether it is necessary to write OpenMP codes that have two, different, approaches to expressing the parallelism (one for the CPUs, and one for offload), or whether it is enough to write the offload code and let the compiler and runtime execute that on CPUs when there are no offload devices available.

For C++ codes, programming models such as Kokkos [3] and SYCL [4] provide a model aligned with idiomatic modern C++ styles for programming heterogeneous systems. These models provide abstractions over the platform for compute and memory, and enable portability between different classes of processor. Both models allow the CPU to be selected as the compute device, and so offer a way within the heterogeneous-first regime to target the CPU. We will further discuss their approaches in Section II.

In this paper we explore how heterogeneous programs perform on a range of the latest CPU-only platforms from a variety of vendors. We take our BabelStream and MiniBUDE benchmarks, which are main memory bandwidth bound and compute bound respectively, and explore how different heterogeneous parallel programming models expose the CPU device. We survey the current state-of-the-art performance on offer from this combination of model and platform, and discuss the impact of the models on the performance portability.

In particular, we make the following contributions:

- Describe the approach to CPU parallelism in the widely used heterogeneous programming models: OpenMP, SYCL and Kokkos.
- Explore the performance portability of programs written for heterogeneous systems on a variety of CPU systems.
- Evaluate the current state-of-the-art performance and behaviour of the compilers and runtimes for these models on CPUs with different architectures and vendors.

A. Related work

Vergara Larrea et al benchmarked a Jacobi OpenMP 4.0 code on the host CPU (AMD Interlagos) and Xeon Phi (Knights Corner) [5]. As processor technology and compiler maturity has considerably improved since this study was performed, it is worth revisiting to explore the latest achievable performance. We build on this study by comparing with other programming models (SYCL and Kokkos) and, additionally, consider a compute bound application. We have been unable to find any other studies investigating the performance of OpenMP code which uses `target` directives when being run solely on a CPU and comparing that with a pre-existing, non-offload, OpenMP version of the code.

In previous work, we have explored SYCL performance on CPUs [6] including for BabelStream. The present work includes updated results for targeting the CPU with SYCL, including results from DPC++ which we were previously unable to collect on Xeon. Alpay and Heuveline demonstrated the challenges of mapping the SYCL parallel compute model onto CPU devices [7], and the hipSYCL implementation has since made significant strides to improve this.

Most studies, including our own, explore the performance of Kokkos on CPUs compared to the CPU backend (e.g. [8], [9], [10]). We include Kokkos here because the way the platform heterogeneity is expressed differs from OpenMP and SYCL.

II. HETEROGENEOUS PROGRAMMING MODELS

In order to program heterogeneous systems, some way to express concurrent work on a target device, such as a GPU, is required. That device may have a discrete memory space from that of the host CPU. Heterogeneous programming has therefore involved defining a model for device discovery, management of data location in mutiple separate memory spaces, and offloading concurrent work to the device. Such a model will be familiar to those who have encountered OpenCL, where simple APIs for these concepts were defined in the first version in 2009 [11]. These fundamental ideas remain present in many of the popular heterogeneous programming models used today.

In this work we consider three such heterogeneous models and abstractions: OpenMP (4.5 and later), SYCL 2020 and Kokkos. These models all aim to abstract away variations in device (processor) type to provide, at a minimum, portability, but are lightweight enough to allow for high performance. In

this way, the models *enable* performance portable programs to be written. Our previous surveys of the performance portability landscapes (such as [9], [10], [6]) have shown that these models are proving to provide performance portability in practice, and have been improving over time.

However, we have not previously explored whether the abstractions in these heterogeneous programming models are lightweight enough to give competitive performance on homogeneous, CPU-only, platforms when compared with pre-existing, non-heterogeneous programs. If they do, this is an important result, as it would mean that a single version of a program could be developed, targeting the accelerator part of a system, and then this single, heterogeneous, version of the code would also then run well on CPU-only systems.

A. Kokkos

Kokkos is a performance portability abstraction layer for C++ applications, primarily developed by Sandia National Laboratories [3]. The model is implemented via C++ header files which provide abstractions for memory and parallel execution. The Kokkos APIs are mapped onto a variety of backend APIs, which might be other programming models, such as OpenMP and SYCL, or vendor-specific APIs such as CUDA and HIP. Kokkos provides the program with a layer of insulation from vendor specific programming models by using its own abstractions and APIs that it can then map to many different underlying interfaces. In this way, Kokkos provides functional portability across platforms.

Kokkos does not expose the complexities of the platform's model to the extent of other models we discuss. The Kokkos platform model assumes a host and a device, using an abstraction called Execution Spaces. Computational kernels are launched to be run in one of these execution spaces. In the current version of Kokkos, there may be at most three execution spaces, one for each of the following: host serial, host parallel and device parallel. A default execution space simplifies much Kokkos programming, with parallel work offloaded to this execution space; this space is usually the device parallel execution space for the backend that was specified when the Kokkos program was compiled.

The (possibly) distinct memory space of the device is exposed via a memory space. A Kokkos View is used to manage the allocation and access of multi-dimensional data arrays in the device memory space. The memory space of a view is fixed at compile time. These can be accessed on the host via a "mirror". Data transfers between the host mirror and device view are explicitly written using a deep copy API.

As a result, Kokkos programs normally use the default execution space for all parallel computation, along with a compatible default memory space, and therefore explicit control is not usually expressed in the program directly — it is hidden in the C++ templating. The exception is the explicit transfer of data between the host and device. Since the memory space is known at compile time, Kokkos can elide the transfers where the device *is* the host.

```
int N = 100'000;
Kokkos::View<double*> A{"A", N};
Kokkos::View<double*> B{"B", N};
Kokkos::View<double*> C{"C", N};

Kokkos::parallel_for(N,
  KOKKOS_LAMBDA (const int i) {
    A(i) = 1.0; B(i) = 2.0; C(i) = 0.0;
});

Kokkos::parallel_for(N,
  KOKKOS_LAMBDA (const int i) {
    C(i) = A(i) + B(i);
};

Kokkos::fence();

auto h_C = Kokkos::create_mirror_view(C);
Kokkos::deep_copy(h_C, C);
for (int i = 0; i < N; ++i)
  assert(h_C(i) == 3.0);
```

Listing 1: A typical Kokkos program

Programs must be written assuming that parallel dispatch is asynchronous, and that parallel regions will be computed in the order they appear in the program. Unlike OpenCL or SYCL, Kokkos does not expose a queue of work to be dispatched to the device. Kokkos has a tasking model abstraction, which we do not explore here, as we consider typical data-parallel workflows with large concurrent loops. We show in Listing 1 a simple but typical Kokkos program (vector addition). The programs which we use in this study follow a similar pattern.

B. SYCL

SYCL is a heterogeneous programming model from the Khronos Group standards body [4]. It is based on C++17 and provides a variety of levels of abstractions for programming heterogeneous systems, with the aim of keeping common programming patterns simple and concise. It follows a similar platform, execution, and memory model to OpenCL, on which it was orginally based; however, the latest version of SYCL (2020) is not tied exclusively to OpenCL and can be (and is) implemented by many different backends.

A SYCL queue is created in order to schedule work on a device. Although full control of the platform at runtime is available in the API, much of this is simplified by a default device selector, which is used by the queue's default constructor. The queue is out-of-order and work submission is asynchronous. This means that submitting work to the queue builds a task graph of kernels in the runtime, with dependencies specified by the access to data.

The CPU can be exposed as a device just as can any attached accelerator such as a GPU. Note that although the CPU is the host, in SYCL it must submit parallel work to itself (as the CPU device) in exactly the same manner as it would to a

GPU device (it can also explicitly submit a "host task" into the task graph to be executed on the host, although this does not express any data parallelism within the submitted task).

SYCL provides two memory abstractions: buffers and accessors; and Unified Shared Memory (USM). Where supported by the device, USM gives a pointer-based abstraction of host and device memory spaces. The user can choose which memory space to allocate memory in through different allocators: With host-only or device-only allocations, it is up to the user to move data between the host and device. With the shared allocation the implementation does this where necessary, often by migrating pages, though, if the hardware supports it, a cache-coherence protocol could also be used; something which is likely to become more common with industry adoption of the Compute Express Link™(CXL™) which explicitly supports such cache-coherent access to remote memory.

Buffers manage the lifetime and location of data, and are not tied to a specific memory space. Accessors to a buffer expose that the data is required inside a kernel or in the host code, and the SYCL runtime will migrate the data on demand within the task graph. The accessors are a powerful abstraction, not unlike a Kokkos View (see Section II-A), as they also define the data dependencies between kernels submitted to the out-of-order queue. For both abstractions, they should have little overhead where the CPU device is used; for USM access is pointer based as normal, and the accessors should eventually decay to raw pointers, with a zero-copy approach taken within the SYCL runtime for any data transfers.

Mapping the `NDRange` parallelism as expressed in OpenCL and SYCL to CPUs is challenging due to the forward progress semantics specified by this model, often requiring compiler support for mapping work-groups and work-items to hardware resources appropriately [12], [13], [7], [14]. For the hipSYCL implementation of SYCL[1], OpenMP and Boost are used as the backend for executing on the CPU device. For Intel's DPC++ implementation, the OpenCL backend, itself implemented by Thread Building Blocks (TBB), is used.

The code in Listing 2 shows the same vector add program SYCL 2020, using the buffer/accessor memory abstraction.

```
int N = 100'000;
sycl::queue Q{};

sycl::buffer<double, 1> A{N};
sycl::buffer<double, 1> B{N};
sycl::buffer<double, 1> C{N};

Q.submit([&](sycl::handler& cgh) {
  sycl::accessor a_A {A, cgh,
 ↪  sycl::write_only, sycl::no_init};
  sycl::accessor a_B {B, cgh,
 ↪  sycl::write_only, sycl::no_init};
  sycl::accessor a_C {C, cgh,
 ↪  sycl::write_only, sycl::no_init};
  cgh.parallel_for(N, [=](sycl::id<1> i) {
   A[i] = 1.0; B[i] = 2.0; C[i] = 0.0;
  });
});

Q.submit([&](sycl::handler& cgh) {
  sycl::accessor a_A {A, cgh,
 ↪  sycl::read_only};
  sycl::accessor a_B {B, cgh,
 ↪  sycl::read_only};
  sycl::accessor a_C {C, cgh,
 ↪  sycl::write_only};
  cgh.parallel_for(N, [=](sycl::id<1> i) {
    C[i] = A[i] + B[i];
  });
});

sycl::host_accessor h_C {c,
 ↪  sycl::read_only};
for (int i = 0; i < N; ++i)
  assert(h_C(i) == 3.0);
```

Listing 2: A typical SYCL program

C. OpenMP

OpenMP initially began as a compiler directive standardisation for thread-based parallelism, but has expanded over the years to incorporate tasks, and, most pertinent to this study, support for heterogeneous computation and memory spaces [15]. Heterogeneous compute arrived with OpenMP 4.0, with significant amendments in 4.5, and improvements in the latest version 5.2 [16].

The evolution from threads to offload has resulted in a standard all but segmented into two distinct but related programming models: we can write an OpenMP program that will only run on the host CPU using parallel regions and tasks, and/or write an OpenMP program that uses target tasks exposing the heterogeneous aspects. It is currently typical that OpenMP programs targeting CPUs and GPUs effectively have two code-paths, one for the host only compilation and one for offloadm with only the latter using the `target` directives. Such specialisation is even built into the standard via various mechanisms, including metadirectives for pre-processor-style selection of OpenMP directives depending on the device, context, etc. As such, code authors face a moral dilemma when presenting "*the* OpenMP version". When aiming for a performance portable implementation, we want to write, as far as we can, a single code, specialising only where necessary. A typical OpenMP program which uses different APIs in the model for each kernel entry point goes against this goal.

Crucially, OpenMP presents the host CPU as a device by the very same mechanisms it uses for accelerator devices, so we do not necessarily need to write our codes in this way. The memory model allows any device allocations to share host storage, meaning the host device can benefit from zero-

[1]Note that at the time of writing, all SYCL implementations are currently classed as in development as they have not yet become conformant.

```
int N = 100000;
double *A = malloc(sizeof(double)*N);
double *B = malloc(sizeof(double)*N);
double *C = malloc(sizeof(double)*N);

#pragma omp target enter data map(alloc:
↪  A[:N], B[:N], C[:N])

#pragma omp target
#pragma omp loop
for (int i = 0; i < N; ++i) {
   A[i] = 1.0; B[i] = 2.0; C[i] = 0.0;
}

#pragma omp target
#pragma omp loop
for (int i = 0; i < N; ++i) {
    C[i] = A[i] + B[i];
}

#pragma omp target exit data map(from:
↪  C[:N]) map(release: A[:N], B[:N])

for (int i = 0; i < N; ++i)
  assert(C(i) == 3.0);

free(A); free(B); free(C);
```

Listing 3: A typical OpenMP Target program

copy memory mapping. The parallelism available in the target region uses threads (and SIMD), but also teams with their looser forward-progress guarantees (essentially equivalent to OpenCL's work-groups); these parallel entities can be assigned to CPU resources appropriately by an OpenMP runtime and compiler. Most GPU compilers now use only two of the three levels of the OpenMP hierarchical parallelism, teams and threads, ignoring SIMD (one exception is the Cray Classic compiler, now only relevant for Fortran). As a result, user code typically only exposes these constructs, without SIMD, although in the past it has been shown that including all three levels in the user code is rarely harmful [17]. It is the goal of this study to explore the feasibility of writing just one OpenMP code path which can be used for both CPUs and GPUs.

Listing 3 shows the simple vector add using OpenMP target directives. We have chosen to use the OpenMP 5 `loop` directive now fairly widely supported by many compilers to show that the loop is trivially parallelisable; previously we would have used `teams distribute parallel for simd` to express much the same description of the concurrency, noting in both cases that we rely on the runtime's defaults for sensible partitioning of the work between teams and threads, as is typical when writing performance portable programs.

III. Benchmarks

Most HPC codes are limited by memory bandwidth, while a few are compute bound. We therefore consider one benchmark at each of these extremes: BabelStream and MiniBUDE.

BabelStream is an implementation of the kernels from the McCalpin STREAM benchmark, with the addition of a dot product [18], [19]. For large input sizes, the performance of all of the kernels is bound by the throughput of main memory. BabelStream is also written to explore various parallel programming models, and as such is extremely useful in performance portability research. The benchmark has implementations in OpenMP, OpenMP target, SYCL 2020 and Kokkos (amongst many others). We will explore the CPU performance of these models, with the standard CPU-only OpenMP implementation as the baseline. BabelStream reports the sustained memory bandwidth for the fastest of each kernel, which can be compared with the theoretical peak bandwidth of the platform, so it is convenient to present the results as architectural efficiency [20].

MiniBUDE is a proxy for the Bristol University Docking Engine (BUDE), a protein docking molecular dynamics code [21]. It is compute bound, typically achieving around 50% of the peak floating point performance of any given platform; as such it is measurably performance portable. As with BabelStream, it is implemented in multiple parallel programming models. MiniBUDE itself reports the sustained GFLOP/s, which we compare with the theoretical peak floating-point performance of each platform.

A. OpenMP benchmarking

For OpenMP, we measure the performance of the two versions of the code (CPU only and offload) when running solely on the CPU. We do, however, compile each version of the code natively for the CPU only. This reflects how most HPC codes are used: we are not looking for a portable binary that can run anywhere and exploit any available offload hardware, but rather a single, portable, source code that can be recompiled for whichever machine is available. We will use appropriate OpenMP environment variables to limit the number of threads to one per physical core (`OMP_PLACES=cores`) and affinity placement settings (`OMP_PROC_BIND=true`) for the baseline result, and use similar flags when running the offload build.

B. SYCL benchmarking

The host CPU is made available to SYCL programs at runtime via the `cpu_selector_v`. Many implementations also provide control of the default selector with environment variables, or by selecting the CPU device from an enumerated list. In all cases, the CPU device is chosen at runtime; the other programming models in this study make this decision at compile time. For SYCL we therefore select the CPU device when running the application.

We will explore two SYCL implementations which target the CPU: DPC++ and hipSYCL. We built hipSYCL with the OpenMP backend in order to enable the CPU device to be

available, and set the same environment variables for thread count and placement as with the usual OpenMP build. DPC++ uses an OpenCL backend, for which we use a recent binary release. As the OpenCL backend does not allow us to limit the number of threads to one per physical core, we will use `taskset` to ensure thread placement and affinity.

At the time of writing, only DPC++ supports the SYCL 2020 reduction interface, so when testing with hipSYCL we remove the Dot kernel from execution. This has no impact on the validity of the benchmark result, as each iteration of each kernel is timed in isolation.

C. Kokkos benchmarking

When benchmarking Kokkos, we build the Kokkos library with the OpenMP backend. This sets the default execution and memory spaces to be the CPU, and so all parallel work will execute using OpenMP threads, and all deep copies will be elided. As with the other models using OpenMP (directly or indirectly), we set appropriate environment variables.

D. Hardware platforms

Table II details the CPUs used to evaluate the performance portability of the heterogeneous programs. With the exception of the Graviton 3, only available via AWS, the CPUs are all hosted in the Isambard supercomputer. The table also includes the theoretical peak main memory bandwidth and peak 64-bit floating point performance which we use to calculate the architectural efficiency of the benchmarks. The CPUs represent the high-end processors typically found in supercomputers and the cloud, across both x86 and Arm architectures.

To ensure BabelStream runs from main memory, rather than Last Level Cache, we set the size of each array to 2^{26} FP64 elements, for a total memory footprint of 1.6 GB.

For MiniBUDE, we use the `bm_1` input deck. The number of poses per work-item (PPWI) is a tunable parameter that defines the scheduling of the 65,536 poses across the compute resources. It is usual to select the parameter as $2^i, i \in [0, 7]$ which results in the best, correct, performance.

IV. Results

We ran the BabelStream and MiniBUDE benchmarks on our CPU platforms using OpenMP target, SYCL and Kokkos, all targeting the CPU device. We also show results from a typical CPU-only OpenMP implementation as a baseline. For BabelStream, we show the Triad kernel, since the other kernels show almost identical trends.

The results are presented using Cascade plots [22], adapted to show results for each model obtained from multiple compilers. In these figures, the lines show the architectural efficiency of the benchmark on a given CPU platform for each implementation of a programming model. Each line, read left to right, is sorted by decreasing efficiency, with each point relating to a compiler; the coloured bar underneath indicates the compiler used to obtain the result shown above. The final data point is augmented with a dashed vertical line dropping to the x-axis to show that no further data exists. The cascade plots here contain results from multiple models built with multiple compilers, so it is clearer to present one figure per platform to make reading the charts as simple as possible.

Where the implementation in a given programming model sustains a high value across the chart, it is getting good performance across a range of compilers. In contrast, when a line drops across the chart, certain compilers are achieving worse performance than others from the same source-code. By comparing the trend for each model across the figures for each platform, we can benchmark the current performance portability of a given model when targeting the CPU across a range of different CPUs.

A. BabelStream

1) Icelake: The architectural efficiency of the Triad kernel on the Icelake CPU is shown in Figure 2. The OpenMP baseline attains around 77% of the theoretical peak bandwidth across all compilers. When using an OpenMP target task to target the CPU we find that the LLVM and Intel compilers attain similar performance to the baseline OpenMP, while the GCC compiler has a noticeable performance penalty for this mode of execution. Using the `perf c2c` Linux tool, the number of remote DRAM accesses is significantly higher for the target version, suggesting NUMA issues with mapping host to device data for the host device.

The Cray compiler has significant issues with thread creation, causing over-subscription of the cores. A workaround is to edit the source to include the `num_threads(64)` clause on each OpenMP construct, and set the environment variable `OMP_NUM_THREADS=1`, obtaining 47% efficiency. However, use of the clauses is a performance portability anti-pattern which should be avoided! [17].

When running on top of the open-source compilers, Kokkos performs well, showing little overhead over the OpenMP backend it is implemented with; however the Intel and Cray compilers show reduced performance.

We show results for SYCL using hipSYCL wit the GCC and LLVM backend OpenMP compilers (labeled as such on the figure) along with DPC++ (labelled as Intel). For hipSYCL, the performance is once again close to that of OpenMP. However DPC++ does not perform well on CPUs; note that we get around 70% efficiency on a single-socket execution, but under 20% efficiency for the same problem run across two sockets, indicating severe issues with NUMA and thread pinning in the runtime. This is not a new observation, as it is a well known property of Intel's OpenCL implementation on which their CPU SYCL is based.

2) Milan: The BabelStream Triad results on the AMD Milan CPU are shown in the cascade plot in Figure 4. The OpenMP baseline is again performance portable across all compilers tested.

The trend for the OpenMP target version is the same as for Icelake: for LLVM and the vendor compiler (AOCC in this case), the performance of the target version is similar to the native OpenMP code, but for Cray and GCC the performance

TABLE II
HARDWARE PLATFORMS

CPU	**Sockets**	**Cores per socket**	**Main memory bandwidth** (GB/s)	**Peak FP64 FLOPS** (TFLOPS/s)
Fujitsu A64FX 1.8 GHz	1	48	1024	2.7648
Graviton 3 @ 2.6 GHz	1	64	300	2.6624
Intel Xeon Gold 6338 @ 2.0 GHz (Icelake)	2	32	409.6	4.096
AMD EPYC 7713 @ 2.0 GHz (Milan)	2	64	409.6	4.096

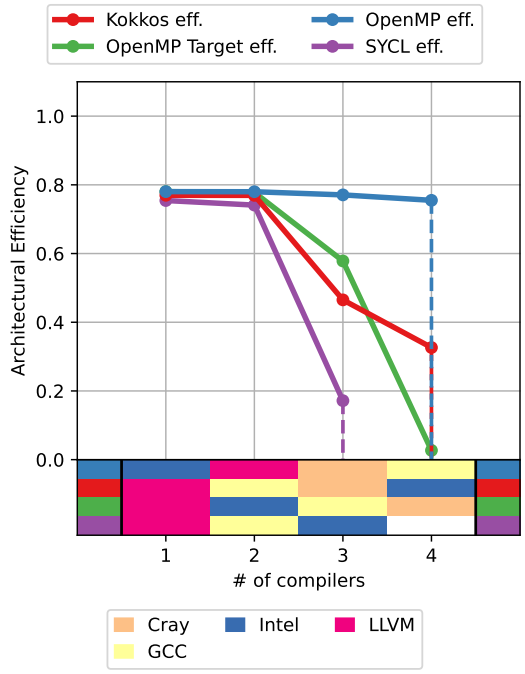

Fig. 2. Cascade Plot of BabelStream architectural efficiency on Intel Icelake for different programming models across a variety of compilers

is reduced. The workaround with the `num_threads(128)` clause did not improve the performance on this platform.

For SYCL, we show results with hipSYCL using GCC, LLVM, and the DPC++ compilers. The results show the same behaviour as on the Intel platform.

Kokkos does not perform on this platform without a minor change to the benchmark source. By default, when a Kokkos View is constructed the data is initalised to zero with `std::memset`. On this platform this results in a non-first-touch-aware implementation creating significant NUMA issues and very poor performance. The result presented in Figure 4 uses an alterntive View constructor provided by Kokkos to remove the initalization, as shown in Figure 3.

We also observed that the OpenMP backend on Milan chooses a static schedule with a chunk size of 4,096. In comparison, our native OpenMP version does not set a chunk size and so has a more optimal distribution of work between the threads. This issue only appears in the development branch of Kokkos, and we have notified the repository author of this regression.

3) Graviton 3: Figure 5 shows the Triad kernel on the AWS Graviton 3 processor. The OpenMP baseline and Kokkos show good performance with all three compilers. Again GCC shows a performance degradation when offload-style OpenMP is targeting the host device, strengthening the observation from the other platforms of overheads in this runtime. Note that as a single socket platform, Graviton 3's NUMA effects should be small, especially as it has a single NUMA domain [23].

4) A64FX: Figure 6 shows the results for BabelStream on the Fujitsu A64FX processor. We observed the same issues with Kokkos here as with Milan, so applied the same fix shown in Figure 3. The results on this platform show a similar story to the others, but we note that the Cray Compiler (LLVM-based) is somewhat older than on the other platforms.

B. MiniBUDE

1) Icelake: The architectural efficiency based on the theoretical peak floating-point performance for MiniBUDE on the Intel Icelake CPU is shown in Figure 7. The PPWI used is 128 in all but three cases: LLVM SYCL where 4 was best, Cray OpenMP target where 64 was best, and Intel SYCL where 1 was best.

The OpenMP baseline attains 68% efficiency with the Intel and LLVM compilers, dropping to around 55% with GCC and Cray. Kokkos attains similar performance when using LLVM, but is slower for the other compilers.

When using the OpenMP target code on the Icelake CPU, it appears that Intel's compiler maps the target directive `teams num_teams(N)` to thread count while ignoring the `OMP_NUM_THREADS` environment variable. Once this was established, setting the workgroup size for MiniBUDE on OpenMP target to the core count showed nearly identical performance to that of OpenMP. The LLVM compiler and its OpenMP runtime handles this mode of operation well.

The SYCL results again confirm that its results on the CPU tend to lag behind the other models. For MiniBUDE, this was observed previously [21], although the performance of DPC++ for this benchmark on CPUs seems to have regressed.

In addition, we were unable to compile hipSYCL using the GCC compiler. When GCC is invoked from hipSYCL's compiler driver, GCC reports errors on ambiguous overloads on lines that involve the use of `local_ptr` and `multi_ptr`. We suspect there is a system configuration issue with our Milan node where out-of-date headers from the Cray software stack were installed in *system* locations.

2) Milan: The MiniBUDE results on AMD Milan are shown in Figure 8. The 70% efficiency figure is calculated based on the observed CPU frequency; recording the frequency scaling during program execution shows a sustained 2.6Ghz clock, a 1.3x increase over the 2Ghz base. Compared with

```
// Original:
Kokkos::View<T*> d_a("d_a", ARRAY_SIZE);
// Alternative:
Kokkos::View<T*> d_a(Kokkos::ViewAllocateWithoutInitializing("d_a"), ARRAY_SIZE);
```

Fig. 3. Benchmark source change required to prevent Kokkos causing NUMA-issues on some platforms

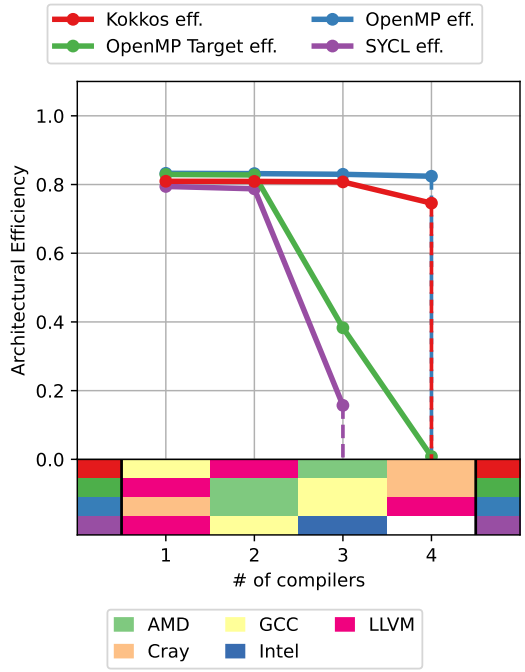

Fig. 4. Cascade Plot of BabelStream architectural efficiency on AMD Milan for different programming models across a variety of compilers

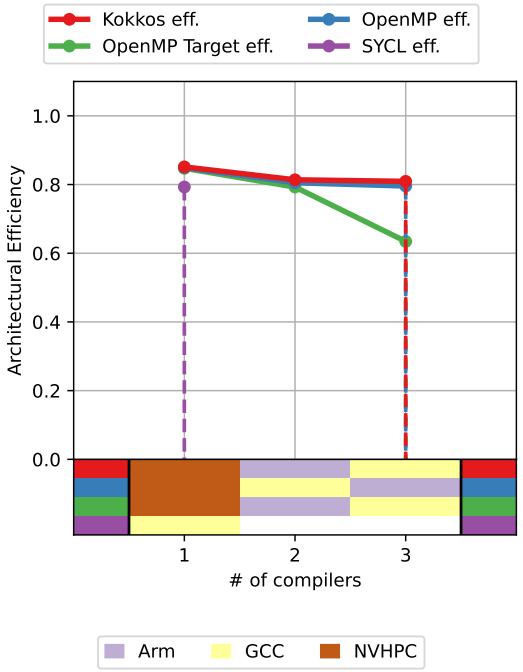

Fig. 5. Cascade Plot of BabelStream architectural efficiency on AWS Graviton 3 for different programming models across a variety of compilers

Icelake, the Milan processor has double the core count but half the vector width, which results in the same peak floating point performance.

The overhead in all compilers for the OpenMP target version is quite clearly visible, however it is relatively small, and even negative with LLVM, where the OpenMP target code is marginally faster than the native OpenMP code.

The binary built with the Cray compiler for both OpenMP versions failed with a segmentation fault at runtime. However, the Kokkos version built with the Cray compiler, which sits on top of OpenMP, did not.

SYCL performed well with hipSYCL using the GCC compiler, but, as with BabelStream, the DPC++ compiler did not produce high performance.

Unfortunately, due to system configuration issues with our Milan node, we were unable to build hipSYCL with the LLVM compiler. The cause appears to be bad linker interactions with the Cray software stack, and solving this would require root permissions which we do not have.

3) Graviton 3: The MiniBUDE results on AWS Graviton 3 are shown in Figure 9. The colour bar at the bottom of the figure is regular for all models, unlike for all of the other plots shown in this study. As MiniBUDE is a compute bound code, this indicates that the GCC compiler is producing the highest quality binary for this benchmark.

OpenMP, OpenMP target and Kokkos all give similar performance. With the NVHPC compiler Kokkos with the OpenMP backend performs better than writing OpenMP directly. NVHPC performs badly with either OpenMP execution model, achieving only 21% of peak, whereas the worst other OpenMP case (Arm, OpenMP target) achieves 46%. GCC again shows high overhead for the OpenMP targe code.

While compiling MiniBUDE with NVHPC, we encountered verification errors on results with certain loop unroll counts. Further reduction of the MiniBUDE kernel revealed that NVHPC appears to have *miscompiled* combinations of the `sinf` and `cosf` calls when handling two or four-wide vector lengths. The snippet in 4 shows the overall structure that causes the miscompilation. We have opened a bug report [24] and Nvidia have promptly provided a workaround for this issue.

4) A64FX: The MiniBUDE results on A64FX are shown in Figure 10. On the whole, the results are a slight improvement on our previous findings on this platform [25].

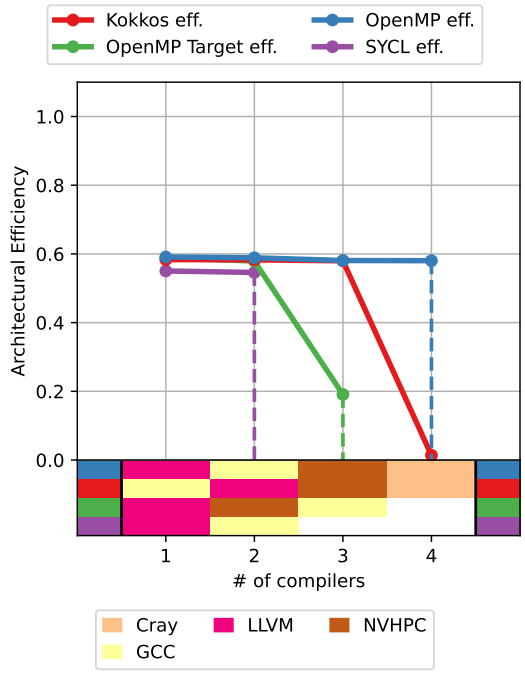

Fig. 6. Cascade Plot of BabelStream architectural efficiency on Fujitsu A64FX for different programming models across a variety of compilers

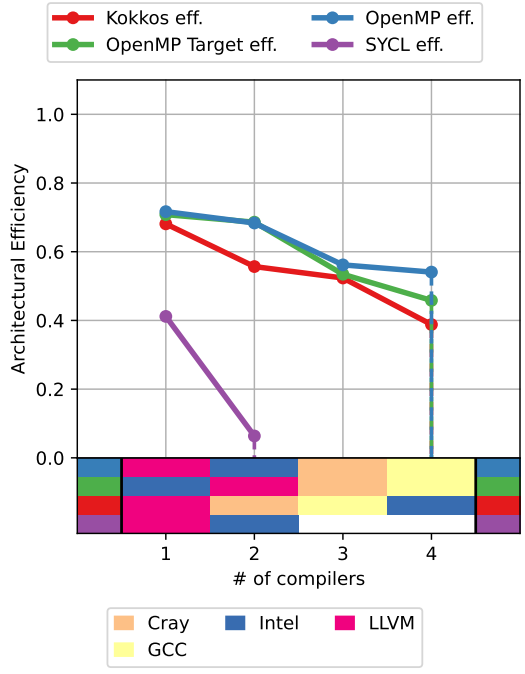

Fig. 7. Cascade Plot of MiniBUDE architectural efficiency on Intel Icelake for different programming models across a variety of compilers. The PPWI value is tuned for each result.

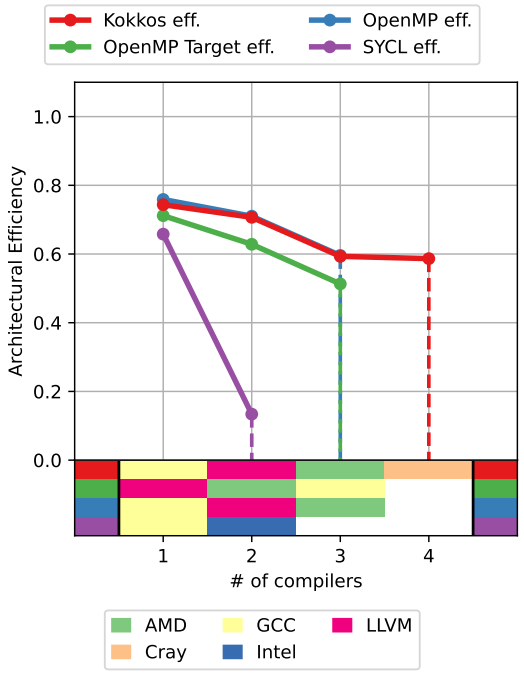

Fig. 8. Cascade Plot of MiniBUDE architectural efficiency on AMD Milan for different programming models across a variety of compilers. The PPWI value is tuned for each result.

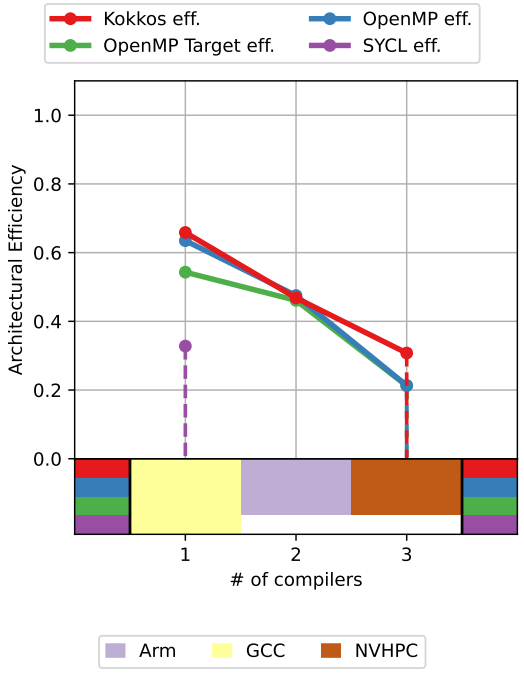

Fig. 9. Cascade Plot of MiniBUDE architectural efficiency on AWS Graviton 3 for different programming models across a variety of compilers. The PPWI value is tuned for each result.

```
void f(int X, int G, const float *xs,
 ↪  const float *ys, float *zs) {
  for (int i = 0; i < X; i++) {
    int ix = G * X + i;
    zs[ix] = cosf(xs[ix]) * cosf(ys[ix]);
  }
}
```

Listing 4: NVHPC miscompilation on sinf/cosf use

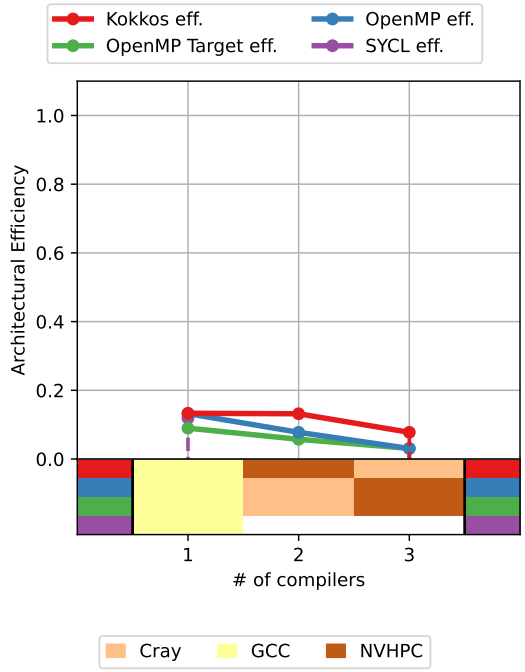

Fig. 10. Cascade Plot of MiniBUDE architectural efficiency on Fujitsu A64FX for different programming models across a variety of compilers. The PPWI value is tuned for each result.

The GCC compiler produced the best results for all programming models, albeit with the overhead for the OpenMP target version seen throughout this paper.

C. Comparison of the models

1) OpenMP: If we focus our attention only on the results achieved by the different OpenMP implementation styles and compilers, we can see that in the four tests where LLVM was used the performance of native OpenMP and OpenMP target code is effectively indistinguishable, with the target code sometimes even being marginally faster than the native implementation. The Intel compiler was only benchmarked for OpenMP on the Icelake machine, but showed similar performance to LLVM there. The GCC target code is consistently slower than native code on all machines, achieving between 46% and 86% of the native code performance. The Cray compiler is extremely erratic, but the target code effectively fails completely on BabelStream achieving less that 5% of the performance of the native code, whereas the MiniBude case achieved 95% relative performance. The AMD compiler showed good relative performance for both codes. The NVHPC compiler showed comparable performance between the two codes (100% relative performance for both), however its performance on MiniBude was poor in both cases (achieving 10% of peak compared with the best OpenMP code [GCC] at 63%).

2) Kokkos: For the majority of the cases, Kokkos attains similar performance to native OpenMP. This indicates that in general the abstractions in Kokkos follow the C++ goals for no overhead for abstraction. On some platforms, Milan and A64FX, the implementation choices within Kokkos had different behaviour than on other platforms. These platforms both have strong NUMA behaviour which exacerbated these issues. The Cray compiler was usually unable to produce competitive results for Kokkos on all platforms, which is perhaps an indication of work to do as their compiler is migrated from Classic to LLVM.

3) SYCL: For SYCL we tested with the hipSYCL compiler's OpenMP backend using GCC and LLVM, along with Intel's DPC++ compiler. The hipSYCL compiler always outperformed DPC++ on the CPU platforms where both were available. In addition, the results are similar to native OpenMP, indicating that for these codes, the abstractions in SYCL, did not produce a significant overhead most of the time. However, there is still some overhead over native OpenMP.

V. Conclusion

While our applications certainly do not exercise all possible OpenMP features (for instance there are no codes using OpenMP tasks), the results from most of the LLVM based compilers (AMD, Intel, LLV itself) in IV-C1 show that it may be possible to write a single, target style, OpenMP code and then achieve good performance on machines both with and without accelerators. However some compilers still need more work to achieve this. Although the Kokkos and SYCL, models were designed in the first instance for heterogeneous programming the can also offer compelling performance on CPUs when built with the GCC and LLVM compilers. Across all platforms, Kokkos and SYCL showed only a small overhead over the native OpenMP base case.

The lack of benchmarks to test heterogeneous codes on homogeneous, CPU-based systems means that many compiler teams may only have tested "compile target for host" style programs for correctness, rather than performance. We hope that our work will encourage compiler teams to pay this use case more attention, and developers to consider the viability of committing to developing a single, performance portable version of their code, with the significant advantages in reduced complexity that this brings.

Acknowledgment

This work used the Isambard 2 UK National Tier-2 HPC Service (http://gw4.ac.uk/isambard/) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/T022078/1). The University of Bristol is an Intel oneAPI Center of Excellence, which helped support this work. Thanks to AWS for supporting access to Graviton 3.

REFERENCES

[1] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "Top 500 June 2022," 6 2022. [Online]. Available: https://top500.org/lists/top500/2022/06/

[2] A. S. Team, "Archer2 Machine Usage," 7 2022. [Online]. Available: https://raw.githubusercontent.com/ARCHER2-HPC/usage-data/main/allusers/2022/07/2022-07_stats_by_uasge.csv

[3] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[4] The Khronos SYCL Working Group, *SYCL 2020 Specification*, 5 2022.

[5] V. G. Vergara Larrea, W. Joubert, M. G. Lopez, and O. Hernandez, "Early Experiences Writing Performance Portable OpenMP 4 Codes," in *Proceedings of the Cray User Group*, ser. CUG, 2016.

[6] T. Deakin and S. McIntosh-Smith, "Evaluating the performance of HPC-style SYCL applications," International Workshop on OpenCL and SYCLCon (IWOCL/SYCLCon). ACM, 2020.

[7] A. Alpay and V. Heuveline, "Hipsycl in 2021: Peculiarities, unique features and sycl 2020," in *International Workshop on OpenCL*, ser. IWOCL'21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3456669.3456691

[8] V. Artigues, K. Kormann, M. Rampp, and K. Reuter, "Evaluation of performance portability frameworks for the implementation of a particle-in-cell code," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 11, p. e5640, 2020, e5640 cpe.5640. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5640

[9] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance Portability Across Diverse Computer Architectures," International Workshop on Performance, Portability and Productivity in HPC held in conjunction with Supercomputing (P3HPC). IEEE, 2019.

[10] T. Deakin, A. Poenaru, T. Lin, and S. McIntosh-Smith, "Tracking Performance Portability on the Yellow Brick Road to Exascale," International Workshop on Performance, Portability and Productivity in HPC held in conjunction with Supercomputing (P3HPC). IEEE, 2020.

[11] The Khronos OpenCL Working Group, *OpenCL OpenCL 1.0 API and C Language Specification*, 10 2009.

[12] T. Baumann, M. Noack, and T. Steinke, "Performance evaluation and improvements of the pocl open-source opencl implementation on intel cpus," in *International Workshop on OpenCL*, ser. IWOCL'21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3456669.3456698

[13] J. Meyer, A. Alpay, H. Fröning, and V. Heuveline, "Compiler-aided nd-range parallel-for implementations on cpu in hipsycl," in *International Workshop on OpenCL*, ser. IWOCL'22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3529538.3530216

[14] T. Deakin, S. McIntosh-Smith, A. Alpay, and V. Heuveline, "Benchmarking and Extending SYCL Hierarchical Parallelism," Workshop on Hierarchical Parallelism for Exascale Computing held in conjunction with Supercomputing (HiPAR). IEEE, 2021.

[15] B. R. de Supinski, T. R. W. Scogland, A. Duran, M. Klemm, S. M. Bellido, S. L. Olivier, C. Terboven, and T. G. Mattson, "The ongoing evolution of openmp," *Proceedings of the IEEE*, vol. 106, no. 11, pp. 2004–2019, 2018.

[16] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*.

[17] M. Martineau, J. Price, S. McIntosh-Smith, and W. Gaudin, "Pragmatic performance portability with openmp 4.x," in *OpenMP: Memory, Devices, and Tasks*, N. Maruyama, B. R. de Supinski, and M. Wahib, Eds. Cham: Springer International Publishing, 2016, pp. 253–267.

[18] 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, Dec 1995.

[19] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via BabelStream," *International Journal of Computational Science and Engineering*, vol. 17, no. 3, pp. 247–262, 2018, special Issue on Novel Strategies for Programming Accelerators.

[20] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019. [Online]. Available: https://doi.org/10.1016/j.future.2017.08.007

[21] A. Poenaru, W.-C. Lin, and S. McIntosh-Smith, "A performance analysis of modern parallel programming models using a compute-bound application," in *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24 – July 2, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 332–350. [Online]. Available: https://doi.org/10.1007/978-3-030-78713-4_18

[22] J. Sewall, J. Pennycook, D. W. Jacobsen, T. Deakin, and S. McIntosh-Smith, "Interpreting and Visualizing Performance Portability Metrics," International Workshop on Performance, Portability and Productivity in HPC held in conjunction with Supercomputing (P3HPC). IEEE, 2020, all authors have equal contribution to this work.

[23] T. P. Morgan, "Inside Amazon's GRAVITON3 Arm Server Processor," 1 2022. [Online]. Available: https://www.nextplatform.com/2022/01/04/inside-amazons-graviton3-arm-server-processor/

[24] W.-C. Lin, "Nvc/nvc++ miscompiles if cosf/sinf is called ," 8 2022. [Online]. Available: https://forums.developer.nvidia.com/t/nvc-nvc-miscompiles-if-cosf-sinf-is-called/223954

[25] W.-C. Lin and S. McIntosh-Smith, "Comparing julia to performance portable parallel programming models for hpc," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2021, pp. 94–105.

APPENDIX A
ARTIFACT DESCRIPTION APPENDIX: HETEROGENEOUS PROGRAMMING FOR THE HOMOGENEOUS MAJORITY

A. Abstract

This paper explores the performance of heterogenenous programming models when targetting CPUs vs a typical homogeneous approach to parallel programming. We tested the performance of two benchmarks, BabelStream and MiniBUDE, written in OpenMP target, Kokkos and SYCL and compared the performance on a range of x86 and Arm CPUs to a baseline OpenMP implementation. The codes are available as open-source software on GitHub with a permissive license.

B. Description

1) Check-list (artifact meta information):

- **Program:** BabelStream and MiniBUDE
- **Compilation:** For our MiniBUDE experiments, we produced and included scripts to build the code on the platforms tested. They may be used as an example to configure on your systems. For BabelStream, the commands to build and run the programs in each compiler on each platform is listed below.
- **Data set:** The benchmarks auto-generates its own inputs based on a command line argument. For BabelStream we used a problem size of 2^{26}. We used the `bm_1` input for MiniBUDE.
- **Hardware:** The Intel® Xeon® CPU, AMD EPYC and Fujitsu A64FX CPU were available in the GW4 Isambard system. The AWS Graviton 3 CPU is available in the Amazon Web Services (cloud). Further details of the hardware was present in the main body of the paper in Table II.
- **Publicly available?:** Yes, on GitHub.

2) How software can be obtained (if available): The BabelStream code is available on GitHub at https://github.com/UoB-HPC/babelstream. The MiniBUDE code is available on GitHub at https://github.com/UoB-HPC/minibude.

All other compilers and software was either pre-installed, or obtained using the usual means.

3) Software dependencies: We used Kokkos 3.6.01 and hipSYCL commit `03de8f8e` dated August 9th, 2022.

On the Intel® Xeon® Gold 6338 ("Icelake") CPU we used the Intel®oneAPI DPC++/C++ Compiler 2022.1.0, LLVM 14, GCC 12.1, and Cray CCE Clang version 11.0.4.

On the AMD EPYC 7713 ("Milan") CPU we used AMD AOCC 3.2.0, GCC 12.1, LLVM 14, Cray CCE Clang version 13.0.0.

On the Fujitsu A64FX, we used GCC 12.1, NVIDIA NVHPC 22.7. For MiniBUDE we used the Cray CCE-SVE compiler version 10.0.1. For BabelStream we used the Cray CCE compiler version 10.0.3.

On AWS Graviton 3, we installed compilers via the Spack package manager. We used GCC 12.1, LLVM 14, Armclang 22.0.1 and NVIDIA NVHPC 22.7.

C. Installation

MiniBUDE used its CMake build system, along with build and run scipts available on GitHub at https://github.com/UoB-HPC/performance-portability/tree/p3hpc22/benchmarking/2022/bude.

BabelStream built and ran the codes via direct command line invocation. We show all steps in the GitHub Gist at https://gist.github.com/tomdeakin/89386d62f5581d073751ecfb9c4c6060.

We note the complexities of running a study like with high multiplicative factors of multiple systems, compilers and programming models.

D. Experiment workflow

The binaries built using the steps in the previous section were executed on the different CPUs with one thread per physical core. OpenMP environment variables (or `taskset` in the case of DPC++) were used to pin threads to cores.

On A64FX we also set the following environment variable: `XOS_MMM_L_PAGING_POLICY=demand:demand:demand`.

In order to identify the NUMA issues with BabelStream GCC OpenMP target, we used `perf c2c` invoked as in Figure 11.

E. Evaluation and expected result

Each benchmark self-validates with an expected input. If the results are not within a suitable tolerance, they are reported as incorrect.

The benchmark outputs the runtime and estimated sustained memory bandwidth or sustained peak floating point performance of the application. In this study we presented architectural efficiency, calculated as the ratio of the sustained perforamnce and the theoretical peak of the processor.

```
OMP_PLACES=cores OMP_PROC_BIND=true OMP_NUM_THREADS=64
 ↪  LD_PRELOAD="/home/br-wlin/usr/lib64/libcrypto.so.10" perf c2c record ./a.out -s
 ↪  $((2**26))
```

Fig. 11. perf c2c invocation