Evaluating ISO C++ Parallel Algorithms on Heterogeneous HPC Systems

Wei-Chen Lin
Department of Computer Science
University of Bristol
Bristol, UK
wl14928@bristol.ac.uk

Tom Deakin
Department of Computer Science
University of Bristol
Bristol, UK
tom.deakin@bristol.ac.uk

Simon McIntosh-Smith
Department of Computer Science
University of Bristol
Bristol, UK
S.McIntosh-Smith@bristol.ac.uk

***Abstract*—Recent revisions to the ISO C++ standard have added specifications for parallel algorithms. These additions cover common use-cases, including sequence traversal, reduction, and even sorting, many of which are highly applicable in HPC, and thus represent a potential for increased performance and productivity.**

This study evaluates the state of the art for implementing heterogeneous HPC applications using the latest built-in ISO C++17 parallel algorithms. We implement C++17 ports of representative HPC mini-apps that cover both compute-bound and memory bandwidth-bound applications. We then conduct benchmarks on CPUs and GPUs, comparing our ports to other widely-available parallel programming models, such as OpenMP, CUDA, and SYCL.

Finally, we show that C++17 parallel algorithms are able to achieve competitive performance across multiple mini-apps on many platforms, with some notable exceptions. We also discuss several key topics, including portability, and describe workarounds for a number of remaining issues, including index-based traversal and accelerator device/memory management.

***Index Terms*—Performance Portability, Programming Models, GPUs C++17, PSTL, stdpar,**

I. Introduction

A. Background

The C++ programming language has frequently been used as the main implementation language for HPC applications. The language has several interesting properties that make it an ideal first choice for current HPC projects. From a performance perspective, C++ is an unmanaged and unsafe language (in terms of memory access) that compiles to native code. As such, the runtime performance is usually comparable to a counterpart written in a C or C-like language.

From a productivity perspective, modern C++ is a nominal and statically typed language that supports advanced language features such as lambdas, templating and object-orientation — features that help reduce duplication and improve code-base scalability. A common theme in C++ is that complex abstractions typically incur very low (zero in many cases) performance penalty when compared to managed languages that offer the same level of abstraction. For example, C++11 lambda expressions, when used together with templates, result in a commanded inlining of the lambda body directly into the receiver. This behaviour is mandated by the standard and not an opportunistic optimisation deferred to the compiler. Beyond low-cost abstractions, C++ also requires a conforming toolchain to ship with an implementation of the C++ Standard Library headers. The headers cover many common programming tasks, and are regarded as one of the main productivity features of C++.

In terms of stability and portability, ISO C++ is a committee driven language with an accompanying formal ISO specification (e.g. C++20 is also known as International Standard ISO/IEC 14882:2020). Feature additions or changes usually go through an extensive refinement process that is then voted on by committee members before final integration into the specification. The C++ committee, WG21 (Working Group 21), has been releasing new specifications on a 3-year cadence starting since C++11, bringing new features and improvements to the language. Because C++ is only a specification, there exists many production-ready compilers from familiar names, such as GCC from FSF, Clang and derivatives from multiple vendors, NVHPC from NVIDIA, and MSVC from Microsoft, to name a few. In all, C++ represents a standardisation of practical abstractions for programs that require bare-metal performance while remaining portable.

B. C++ Parallelism Technical Specification

The C++ language offers a range of algorithm-related functions in the `numeric` and `algorithm` header. These headers are intended to improve productivity by providing a predefined set of common algorithms that are easy to use, portable, and well tested. Many of the algorithms operate on C++ iterators, which ensures good compatibility across all C++ containers and any third-party containers that implement the interface. Notably, many algorithms are designed in a functional style that operate on higher-order functions (i.e. lambda expressions or functors); the implementation details in the algorithms such as the induction state of a traversal, are not exposed.

Before C++17, we see a wide range of third-party libraries and frameworks that attempt to offer a parallel or offload implementation of C++ algorithms. For example, NVIDIA's Thrust library implements an abstraction layer for CUDA where kernel invocation can be written using an API similar to the one from C++'s algorithm library. On the CPU side, we see concurrency libraries such as Intel Threading Building Blocks (TBB) which offers transparent thread management.

As C++ features are usually a standardisation of common use cases through proposals, the current parallel algorithms extension we see in C++17 also started out as a proposal to the C++ standard. The earliest revision dates back to 2013 (n3554[1]), with the final draft version completed in 2015 (n4507[2]) and subsequently merged into C++17. The proposal is done in a form of a draft Technical Specification (TS) which contains detailed considerations on the state of the art and also the shape of the proposed API. For the Parallelism TS draft, it carefully considered existing libraries and discussed their strength and weaknesses. The final API was designed as a series of overloads to the original algorithms API with an extra execution policy parameter at the beginning. Use cases such as exception handling, the potential for offload compute, and even vectorisation, are also touched upon.

C. Contribution

This study provides an evaluation of the performance and productivity properties of implementing HPC applications using ISO C++ parallel algorithms introduced in C++17 and newer revisions. This paper will first discuss methods of expressing parallelism under ISO C++ in Section III. Then, in Section IV, we provide an overview of currently available C++17 parallel algorithms implementations in the ecosystem. Where possible, the overview will include brief descriptions of the backing parallelism library each implementation is built upon. Finally, we introduce our benchmarking applications in Section V and discuss any challenges encountered during the porting process to ISO C++17. In the remainder of paper, from Section VI, we present performance and productivity results of our mini-app ports.

This paper makes the following contributions:

- We survey the state of the art for implementing HPC applications using only standard parallelism features in ISO C++17 or newer.
- We present ISO C++17 ports of three HPC mini-apps, across different styles of parallelism.
- We benchmark the ports against established programming models on a wide range of CPU and GPU platforms to compare relative overhead.

II. Related work

There have been a few studies exploring the performance of standard parallel algorithms in C++ for high-performance computing. The LULESH mini-app was ported to ISO C++17 and evaluated on NVIDIA GPUs, comparing only against OpenACC [3]. This study used an index-centric approach, developing their own counting iterator similar to ours in Listing 2. LULESH shares some similarity with the CloverLeaf code used in our study, and so we extend the analysis for standard C++ for hydrodynamics kernels significantly in this work.

The STLBM code [4] uses C++17 `for_each` for a Lattice-Boltzmann code. The kernel requires knowledge of the grid position to calculate the stencil. They follow the data-centric pattern shown in Listing 1, but use pointer arithmetic to locate the position in the iteration space. The difference of the address of the lambda-argument and the base pointer is used to recover the index as an alternative to creating a counter iterator. The study presented performance on CPUs and GPUs, but did not compare to other programming models.

Drocco et al. enabled the parallel STL on distributed systems, but did not consider heterogeneous systems [5]. Jääskeläinen et at implemented the C++17 parallel STL for heterogeneous systems with shared virtual memory [6]. Their approach uses HSA and HSAIL, but they expose the heterogeneous device via a new execution policy, although they argue that using the standard policies to execute on accelerators might also be conformant. Whilst they target different systems, these approaches both explore the idea that the iterations of the parallel algorithms can execute on some external processor.

There are a number of C++ abstraction frameworks that provide heterogeneous computing in C++. Many of these have informed the development of the parallel algorithms in C++. This study focuses on the parallel algorithms available in the standard today, so whilst we do not discuss them in detail in this study, we will compare the performance against some well known approaches.

III. Expressing Parallelism

Any programming model targeting heterogeneous platforms needs to consider how the complexities of the system are to be controlled. A model for the platform allows identification of the different hardware resources, be they the host CPU or an attached accelerator device. As the system can have multiple places to execute parts of the program (the host CPU, the device, etc), programming models expose some way to specify *where* the computation of a particular region should happen. When this is programmable, such as in OpenMP, SYCL, CUDA, HIP, etc., devices can be selected as a target for compute or data transfers. Just considering the threading model in C++, ignoring any heterogeneity, the current ISO standard only gives ways of computing within the current thread, or in a different thread.

Accelerator devices may have distinct memory spaces, and provide some mechanism to migrate data between those memory spaces. The C++ of today (C++17, C++20 and looking towards C++23) does not give much explicit control of data location. Most notably there is no idea of distinct memory spaces, but many hardware platforms for HPC support some form of Unified Shared Memory, often where devices can simply access all memory in the system via page faults. In this way, programming the location of data is akin to managing NUMA behaviour.

Finally, the programming model needs to express concurrent work. C++ offers a number of ways of expressing this on the host CPU, including threads, fibers, futures, etc. These approaches share more similarities with task-based programming models, where asynchronous units of work are defined and processed. For HPC however, the data parallel algorithms available in the `numeric` and `algorithm` libraries allow the expression of parallel work, including loops over data and an

iteration space. More features are coming to C++ through the `ranges` standard library to express multidimensional iteration spaces, which may be processed concurrently by an algorithm.

As of the time of writing, C++ gives us the controls to express our parallel work. However, it provides little in the way of control over where that work should take place, or over where our data may reside, especially where different memory spaces are concerned. Control of execution space is likely to arrive in the standard soon, with current implementations taking various approaches (described in Section IV) to solve this problem to accelerate adoption.

A. Data v.s. Index centric traversal

The C++ parallel algorithms are implemented as overloads to the existing algorithms API with an extra execution policy parameter. As such, the domain of our input must be expressed using the standard *begin* and *end* iterator arguments. For traversing over a sequence that holds data, this can be accomplished in a straightforward manner, as shown in Listing 1. The snippet shows two variants of sequence traversal: the first variant receives each element of the container for side effects, the second variant maps each element of the container and inserts it into another container.

Listing 1: Data-centric sequence traversal examples.

```
auto exec = std::execution::par_unseq;

std::vector<T> xs = /*...*/;
std::for_each(exec, xs.begin(), xs.end(), [](T &x){ /* ... */ });

std::vector<T> ys(xs.size());
std::transform(exec, xs.begin(), xs.end(), ys.begin(),
    [](T &x){ return /* new value, witnessing x */ });
```

This data-centric style of parallelism (data parallel) shown in Listing 1 is problematic for stencil-like applications, which usually need access to neighbouring cells in a structured grid. Similarly, accessing elements at the same index of a different sequence in a data parallel traversal is not possible unless the current index is tracked externally. This problem can be worked-around naively by generating a sequence that is filled with the indexes. Using this yields an index in the lambda term of traversal algorithms, which can then be used to access the required elements at the correct offset. However, this solution incurs a memory overhead that is proportional to the size of the input, and also runtime overhead for generating the sequence.

Recall that C++ iterators are simply an interface that can be implemented for any type. Listing 2 shows a skeleton implementation of a range iterator for a numeric type. With an index range iterator, we can realise the traditional index-centric (index parallel) traversal.

Listing 2: Index-centric sequence traversal examples with range iterator.

```
template <typename N> struct range {
  struct iterator {
    friend class range;
    using difference_type = typename std::make_signed_t<N>;
    using iterator_category = std::random_access_iterator_tag;
    using value_type = N;
    using reference = N;
    using pointer = const N*;
    // operator implementation for
    //  [],*,+,++,-,--,+=,-=,==,>=,<=,>,< omitted
    protected: explicit iterator(N start) : i_ (start) {}
    private:   N i_;
  };
  iterator begin() const { return begin_; }
  iterator end() const { return end_; }
  range(N begin, N end) : begin_(begin), end_(end) {}
  private: iterator begin_, end_;
};
std::vector<T> xs = /*...*/;
range<int> r(0, xs.size());
std::for_each(/*policy*/, r.begin(), r.end(), /*lambda*/);
```

C++20 implements a more general version of Listing 2 using ranges. For example, the index traversal in Listing 2 can now be written as shown in Listing 3.

Listing 3: C++20 ranges

```
std::for_each_n(/*policy*/,
    std::views::iota(0).begin(), N, [] (int i) {/*...*/} );
```

IV. ISO C++ Parallel Algorithm Implementations

Only recently have compiler vendors started to ship production ready parallel execution policy implementations for CPUs. With the recent release of NVIDIA's NVHPC compiler and Intel's oneDPL library, we also start to see viable offload execution policies for GPUs. This section gives an overview of currently available C++17 parallel algorithm implementations (also called Parallel STL, or *PSTL* for short).

A. libstdc++ PSTL

libstdc++ is a C++ standard library implementation developed by GNU. Similar to how glibc is the default C standard library on most Linux distributions, libstdc++ fills this role for the C++ language. The implementation is designed to be portable: non-GCC compilers such as Clang and derivatives are frequently configured to use libstdc++ for a complete toolchain.

libstdc++ implements the C++17 execution policy specification under the PSTL component, and is backed by Intel's Threading Building Blocks (TBB). TBB is an open source parallelism and concurrency abstraction library that provides frequently-used threading primitives such as thread pools, task schedulers, and affinity related utilities. The TBB-backed PSTL implementation is contributed by Intel; programs that wish to use PSTL must explicitly link against a working TBB library during compilation.

Historically, before the introduction of C++17, libstdc++ had an experimental parallel STL implementation that implemented a set of STL algorithms using OpenMP. This API is only available with non-standard headers and lives under the `__gnu_parallel` namespace, both of which have been deprecated.

B. LLVM PSTL

Similar to libstdc++ from GNU, the LLVM project contains an implementation of the C++17 execution policies, also named PSTL. The initial implementation is again contributed by Intel and uses the TBB backend, as with the libstdc++ implementation. Unlike libstdc++, this implementation also

includes an OpenMP and macOS Grand Central Dispatch (GCD) backend, both contributed by the LLVM community.

The LLVM PSTL implementation is a standalone project under active development, with integration into the LLVM repository only starting in early 2019. There are plans to integrate LLVM PSTL into *libc++*, LLVM's own C++ standard library implementation. However, as of LLVM 15, the merge appears to still be in progress.

C. Intel oneDPL

Intel oneDPL is an open source C++17 execution policy implementation for both CPUs and GPUs. The oneDPL library is part of Intel's oneAPI umbrella which provides a set of unified software components with a focus on Intel hardware. Unlike libstdc++, the oneDPL implementation is a standalone library which can be used independently of the system C++ standard library.

The oneDPL codebase is primarily a fork of the LLVM PSTL, with additions to support SYCL. For the CPU implementation, the codebase delegates directly to the forked LLVM PSTL code path as touched on in Section IV-B. For accelerators, a SYCL execution policy allows algorithms to be offloaded to GPUs that support SYCL2020. SYCL is an emerging heterogeneous parallel programming model that exposes parallelism through an idiomatic C++ API. SYCL supports GPU kernel programming in a single-source compilation model (in contrast to multi-source models such as OpenCL); SYCL's device code (i.e. kernel) can be written in C++ that is inline with the host code.

For oneDPL on GPUs, the library implements templated algorithms that are backed with optimised SYCL kernels. In this context, all SYCL2020 restrictions still apply, thus references to heap memory in the kernel must be captured by value, and anything that gets passed to the GPU device must be device-copyable. This is a non-trivial restriction that complicates the use of many C++ container types such as *std::vector*.

Historically, Khronos has experimented with implementing C++ parallel algorithms using the deprecated SYCL 1.2 standard[7]. While the implementation is fully functional, no capture of pointers are allowed due to the lack of USM support.

D. NVIDIA NVHPC

The NVIDIA *NVHPC* compiler includes support for C++17 execution policies through the `-stdpar` compiler flag. Since 2020, NVIDIA has consolidated their existing compiler components (e.g. CUDA SDK, PGI compilers) into a complete HPC SDK package. The former PGI compiler (an optimising heterogeneous compiler by *The Portland Group, Inc.*, since acquired by NVIDIA) has been renamed to NVHPC which integrates components from the LLVM project. The compiler refers to its C++17 execution policy implementation as the C++ standard parallelism support, or *stdpar* for short.

NVHPC is a proprietary compiler, so we can only refer to publicly available documentations of the stdpar component. NVHPC's stdpar support for CPU is backed by OpenMP, a directive-based model that is natively supported in NVHPC with semantics and an API surface similar to NVIDIA's own OpenACC. For NVIDIA GPUs, NVHPC internally uses the NVIDIA Thrust library in conjunction with compiler-level outlining of heap pointers.

The NVHPC GPU implementation is able to track heap allocations by reference, even when they are behind multiple layers of pointer indirection. In practice, this requires almost no attention from the programmer's side on writing portable programs that run on GPUs. This is a flexibility that oneDPL (with the SYCL2020 backend) cannot support.

E. HPX

HPX is a software framework by the STE||AR group that provides portable implementations of current and future C++ standards. The framework is focused mainly on scaling across multiple nodes with a strong emphasis on asynchronous programming. Many features of C++ draft proposals related to concurrency and threading have implementations in HPX. While HPX do currently implement the C++17 execution policy APIs, the library requires explicit initialisation of the framework which is not part of the C++17 execution policy specification. In addition, HPX only has scheduling support for accelerator offload tasks; kernels must still be written in a separate source file.

F. AMD HCC

Up until 2019, AMD had experimental support for C++17 execution policies via the now deprecated open source HCC compiler. Beyond C++17 execution policies, the HCC compiler supports targeting the AMDGCN ISA for a wide range of parallel programming models, including HC, OpenMP, and even C++ AMP. Unfortunately, since the last release in mid-2019, AMD has deprecated the HCC compiler and shifted focus to HIP, an open source programming model that partially resembles the CUDA API.

V. Mini-apps in the study

To evaluate performance, we have selected two mini-apps to cover compute-bound (miniBUDE) and memory bandwidth-bound (BabelStream) application domains. To evaluate productivity, we select a larger mini-app (CloverLeaf) that has a high unique kernel count and dependency on MPI. For all mini-apps in this study, we conduct benchmarks against a vendor-supported programming model (OpenMP for CPUs, CUDA/SYCL for NVIDIA and Intel GPUs) and also the backing implementation of the C++17 execution policy (e.g. TBB or OpenMP) if one exists. Where possible, we cover both heterogeneous compute where algorithms are offloaded to GPUs, and traditional multicore CPU workloads on single and multiple socket systems using current HPC hardware platforms. To do this, we select currently maintained C++17 execution policies implementations, as shown in Table I.

Implementation	CPU	Offload (GPUs)
GNU libstdc++ PTSL	TBB	Not supported
Intel oneDPL	TBB, OpenMP	SYCL2020 with USM
NVIDIA NVHPC	OpenMP	NVIDIA GPUs (Pascal or newer)

TABLE I: C++17 execution policy implementations

Algorithm 1 BabelStream kernels

procedure COPY($A[n], C[n], n$)
  for $i \leftarrow 0$ **to** n **do** $C[i] \leftarrow A[i]$
procedure MUL($A[n], B[n], C[n], scalar, n$)
  for $i \leftarrow 0$ **to** n **do** $B[i] \leftarrow scalar * C[i]$
procedure ADD($A[n], B[n], C[n], n$)
  for $i \leftarrow 0$ **to** n **do** $C[i] \leftarrow A[i] + B[i]$
procedure TRIAD($A[n], B[n], C[n], scalar, n$)
  for $i \leftarrow 0$ **to** n **do** $A[i] \leftarrow B[i] + (scalar * C[i])$
procedure DOT($A[n], B[n], scalar, n$)
  for $i \leftarrow 0$ **to** n **do** $sum \leftarrow sum + (A[i] * B[i])$
  return sum

A. BabelStream

BabelStream is a memory-bandwidth bound mini-app that implements the original McCalpin STREAM kernels: Copy, Mul, Add, Triad [8]. The mini-app also implements an additional dot product kernel as shown in Algorithm 1. As the name suggests, BabelStream implements these kernels in multiple programming models (e.g. CUDA, SYCL, OpenMP, etc.), the coverage implies that BabelStream also runs on a wide range of hardware, including CPUs, GPUs, and accelerators [9]. Each of the implementations is written in a generic and idiomatic way to serve as exemplars for a particular parallel programming model.

The BabelStream benchmark measures the runtime of each kernel and computes the total bandwidth attained based on the elapsed time. Precautions have been taken to ensure no unrealistic compiler optimisations are possible: the data movement must take place.

We use the 2^{29} element size ($\approx$ 4GB) to validate results with past studies. For platforms that do not support this value, we use the default, which is clearly marked in Section VI. The benchmark is configured to run each kernel 100 times and records the maximum attained bandwidth.

For the C++17 port[1], we are presented with the three styles of parallelism discussed in Section III. We chose to implement both index and data based parallelism and compare performance. A C++20 range implementation was also attempted but none of the current implementations support multi-core or offload execution.

B. miniBUDE

MiniBUDE is a molecular dynamics mini-app that simulates docking of molecules to predict the resulting structure: a process frequently used for drug discovery[10]. This is a performance proxy application that implements only the compute-intensive virtual screen step of the full scale Bristol University Docking Engine (BUDE). Intuitively, the virtual screen process computes the charge interactions between a set of predefined ligand and protein molecules with different transformation poses.

Algorithm 2 miniBUDE Fasten Kernel

procedure FASTEN(const i, const $xform_{3\times3}[]$,
const $proteins[ps]$, const $ligands[ls]$, out $energy[]$)
  ▷ Values $R, DSLV, DSLV_R, NZ, DST_1, DST, HRD, T$ are part of the simulation constants
  for $il \leftarrow 0, ls$ **do**
    $lpos_{1\times3} \leftarrow xform \cdot ligands[il].pos_{1\times3}$
    for $ip \leftarrow 0, ps$ **do**
  ▷ Atom distance and sphere radii sum
      $dist \leftarrow distance(lpos, proteins[ip].pos_{1\times3})$
      $d \leftarrow dist - R$
  ▷ Steric energy, formal/dipole charge interactions
      $energy[i] \leftarrow energy[i] +$
        $(1 - dist * (1/R)) * (d < 0?2 * HRD : 0)$
      $e \leftarrow init *$
        $(d < 0.f?1 : (1 - d * DST_1)) *$
        $(d < DST?1 : 0)$
      $energy[i] \leftarrow energy[i] + (typeE? - |e| : e) * T$
  ▷ Nonpolar-Polar repulsive interactions
      $dslvE = dslvInit *$
        $((d < DSLV \wedge NZ)?1 : 0.f) *$
        $(d < 0?1 : (1 - d * DSLV_R))$
      $energy[i] \leftarrow energy[i] + dslvE * 0.5$

MiniBUDE is compute-bound mini-app that contains only a single, though non-trivial, compute kernel. A high-level pseudocode structure is shown in Algorithm 2; the kernel makes heavy use of single-precision trigonometric, square root, and absolute value operations in a partially unrolled loop. Similar to BabelStream in objective, miniBUDE is also implemented in multiple programming models. The general structure of the kernel is designed with vectorisation in mind. For all programming models, the kernel exposes a loop unrolling variable (PPWI) that controls the inner loop unroll count to support vectorisation.

For benchmarking, the kernel driver reads in a predefined dataset that contains known results. The kernel runtime is then measured and the results validated against the control. To extend the duration of the benchmark, miniBUDE can be configured to repeat a docking run, similar to running the multiple virtual screen step back-to-back in the full scale application. We use the *bm1* dataset with the default 10 iterations to validate results with the original study[10].

For the C++17 port[2], only the index-based parallelism is considered because miniBUDE already partitions the task based on numeric groups.

C. CloverLeaf

CloverLeaf is a 2D hydrodynamic simulation application implemented using the compressible Euler equations[11]. The implementation uses the second-order accurate method with discretisation of the values on a structured grid; the simulation tries to solve PDEs (partial differential equations) on the conservation of mass, energy, and momentum using the finite-volume method.

Programmatically, CloverLeaf is implemented as a series of steps which traverse a 2D grid (e.g. `double[][]`). Like BabelStream and miniBUDE, CloverLeaf has implementations

[1]https://github.com/UoB-HPC/BabelStream/tree/option_for_vec

[2]https://github.com/UoB-HPC/miniBUDE/tree/v2

Algorithm 3 High-level CloverLeaf kernel overview

▷ Each procedure traverses the full $W \times H$ grid, some procedures may invoke multiple kernels
1: **while** $step < maxStep$ **do**
2: **procedure** IDEAL_GAS
▷ Pressure/sound speed via ideal gas equation of state with a fixed gamma
3: **procedure** VISCOSITY
▷ Artificial viscosity via the Wilkin's method to smooth out shock front and prevent oscillations
4: **procedure** PDV
▷ Cell energy/density δ via velocity gradients
5: **procedure** CALC_DT
▷ Compute the minimum timestep based on CFL conditions, velocity gradient, and velocity divergence.
6: **procedure** ACCELERATE
▷ Update velocity field via cell pressure/viscosity gradients
7: **procedure** FLUX_CALC
▷ Edge volume fluxes using the velocity fields
8: **procedure** ADVECTION
▷ Setup fields for the next iteration
9: **procedure** RESET_FIELD
▷ Edge volume fluxes based on the velocity fields
10: **procedure** FIELD_SUMMARY
▷ Total mass, internal energy, kinetic energy, and volume weighted pressure
11: $step \leftarrow step + 1$

in multiple parallel programming models. CloverLeaf is a challenging mini-app to implement ports for most programming models. At its core, CloverLeaf has over 100 unique kernels that involve 1D/2D traversal and 1D/2D reductions of multiple values.

CloverLeaf accepts input decks as the starting parameters for the simulation. An input deck includes seed parameters, simulation step count, and the expected solution for validation. For benchmarking, we have selected the `bm_16` deck for consistency with our past results [12]. For scaling results, we use the same `bm_16` deck but limit the timestep to 300 steps. In both cases, the total runtime of the simulation is measured.

For the C++17 port of CloverLeaf[3], we have implemented the index parallel variant only. The data parallel variant is not possible without a complete rewrite because CloverLeaf requires traversal of multiple grids at once based on the same index. Similar to other ports of the CloverLeaf, the C++17 porting process rewrites all kernels (excluding the top-most simulation loop) in idiomatic C++.

VI. Results

A. Platform software setup

To evaluate the performance of ISO C++ parallelism fairly and comprehensively, we have selected current HPC hardware platforms from multiple vendors and compiled our mini-apps with multiple compilers where possible. Table II shows the hardware configurations used for benchmarking. We have listed the theoretical memory bandwidth for each platform, this will be the value used Section VI-C. For floating point performance, we compare relative performance between models in Section VI-D and Section VI-E so the FLOPS value is not shown.

[3]https://github.com/UoB-HPC/cloverleaf_stdpar

On CPU platforms, we are interested in two properties: the micro architecture and memory access. Architecture wise, we consider both the traditional x86 and also AArch64 with SVE support. We also select platforms with different memory hierarchies, ranging from single NUMA regions to eight in a dual socket configuration.

For compilers on CPU platforms, we have selected the latest versions of frequently used HPC compilers, including GCC, Clang variants, and also the new NVHPC compiler. Table III shows the versions selected for each platform. For Clang, we use the vendor supplied variants which contain extra optimisations for their respective platforms.

For GPU platforms, we are limited by what our selected C++17 execution policy can support as shown previously in Table I. We use oneAPI `dpcpp` version 2022.1 on Intel GPUs and `nvc++` or `nvcc` from the NVHPC 22.7 distribution for NVIDIA GPUs. The V100 GPU is using CUDA 11.2 on driver 460.32, while the A100 GPU uses the newer CUDA 11.4 and driver 470.57.

B. Porting considerations

We have followed a set of general guidelines while porting the mini-apps to C++17 parallel algorithms. For standard conformance, we carefully avoid APIs that are not part of the C++ standard. This was done by test compiling against libstdc++ on both GCC and Clang with all C++ vendor extensions disabled (e.g. compiling with `-std=c++17`). In effect, except oneDPL, our C++17 ports require no code change for both CPUs and GPUs, and between different compilers. An exception was made for oneDPL because, as a standalone library, it makes use of non-standard headers and exposes the execution policy under a different namespace, so a small (< 6 lines) shim header is required for portability.

Special attention was made to ensure the code written is idiomatic C++. For example, we make us of *std::transform_reduce* for complex value reductions as per the documentation, with no platform or compiler specific tweaks. As discussed in Section III-A, each C++17 parallel algorithm *must* support a standard-conforming C++ iterator. We test this using the full custom iterator implementation shown in Listing 2.

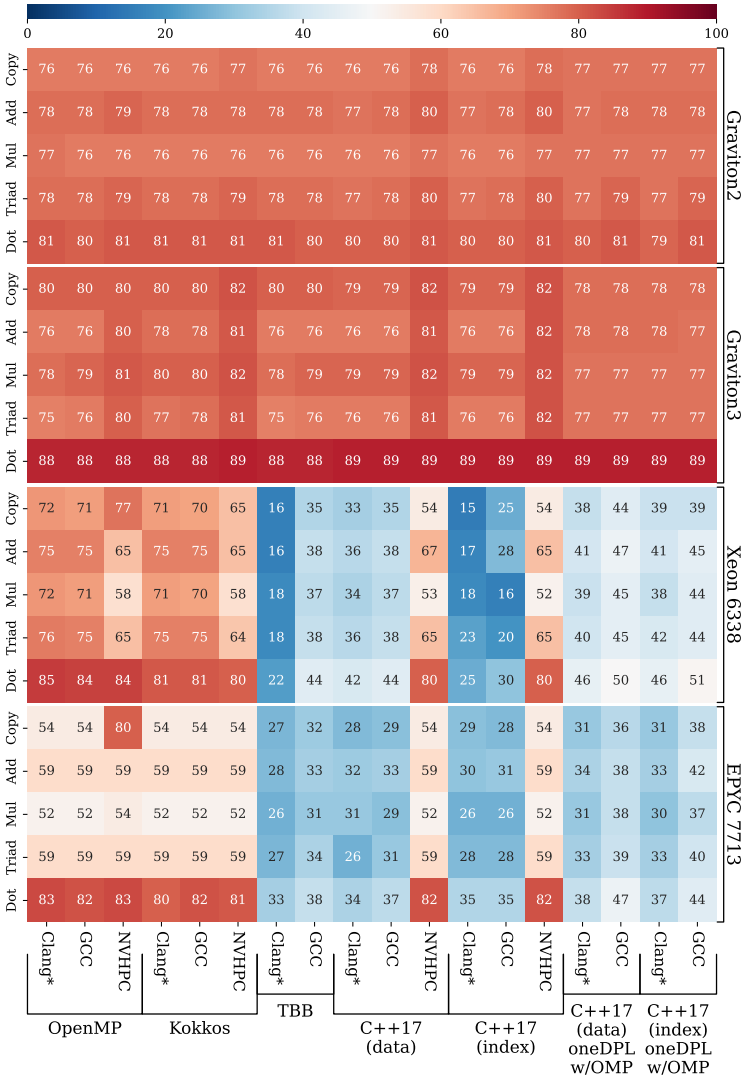
For NUMA awareness, we only allocate memory using `malloc` or equivalent. This was done to avoid any initialisation which allows first-touch affinity for each thread. Experimentation with C++ containers such as `std::vector` shows that it is non-trivial to leave the allocation uninitialised which leads to poor performance. There are source-level workarounds available such as custom allocators, however, these are not attempted to limit the scope of the study.

C. BabelStream

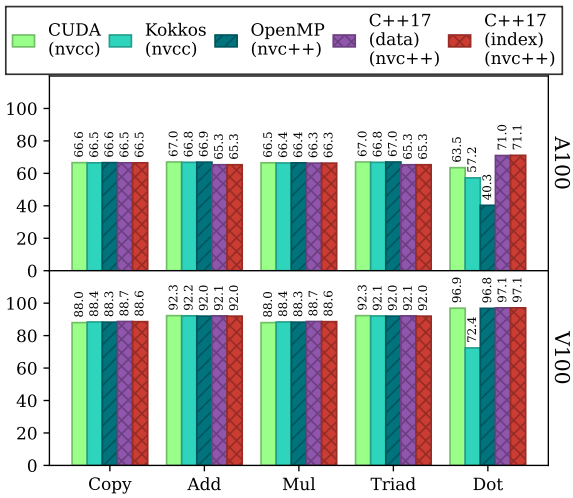
1) CPUs: The BabelStream port implements both the data and index parallel variant discussed in Section III. Bandwidth results for CPUs are shown in Fig. 1a. On CPU platforms, we compare the bandwidth to the backing implementation of the respective C++ execution policy: TBB for libstdc++ and

TABLE II: Platform details

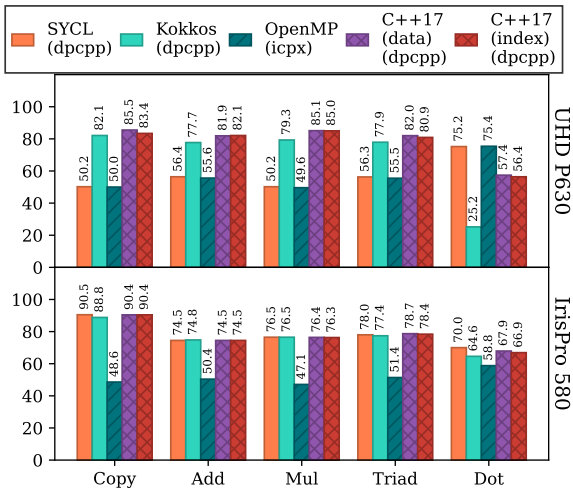
Vendor	Name	Architecture	Abbreviation	Device Type	Total NUMA nodes	Theoretical Peak Mem. Bandwidth (GB/s)
Intel	Xeon Gold 6338	x86, Ice Lake	Xeon	HPC CPU (32C*2)	2 (1 per socket)	409.56
AMD	EPYC 7713	x86, Zen3 (Milan)	EPYC	HPC CPU (64C*2)	8 (4 per socket)	409.56
AWS	Graviton 2	AArch64, Neoverse N1	Graviton2	HPC CPU (64C*1)	1	204.8
AWS	Graviton 3	AArch64, Neoverse V1	Graviton3	HPC CPU (64C*1)	1	307.2
NVIDIA	Tesla A100 (SXM 40GB)	Ampere	A100	HPC GPU	N/A	2039
NVIDIA	Tesla V100 (PCIe 16GB)	Volta	V100	HPC GPU	N/A	900
Intel	UHD P630 (Xeon E2176G)	Gen9.5	UHD	Server iGPU	N/A	42.6
Intel	IrisPro 580 (i7 6670HQ)	Gen9	IrisPro	Consumer iGPU	N/A	34



(a) AArch64 and x86 CPU results



(b) NVIDIA GPU results



(c) Intel GPU results (2^{25} elements $\approx$ 0.3GB)

Fig. 1: BabelStream results as % of peak memory bandwidth *per platform*; **higher is better**

Compiler	AWS Graviton2/3	AMD EPYC	Intel Xeon
Clang*	ACfL 22.1 (armclang, LLVM 13)	AOCC 3.2.0 (LLVM 13)	oneAPI ICPX 2022.1 incl. `dpcpp`, `icpx` (LLVM 14)
GCC	12.1.0		
NVHPC	22.7, incl. `nvcc`, `nvc`, `nvc++`		
oneDPL	2021.7		

TABLE III: Compiler/library version for CPU platforms

OpenMP for NVHPC and oneDPL. We also include results for Kokkos, a C++ performance portability library[13] that delegates to OpenMP on CPUs.

On AArch64 CPUs, we observe very similar performance across Graviton2 and Graviton3, in the 75% to 90% of peak range. The performance scales well with the core count, as shown in Fig. 2. This highlights the overall maturity of the AArch64 codegen in all the compilers we tested. Based on this,

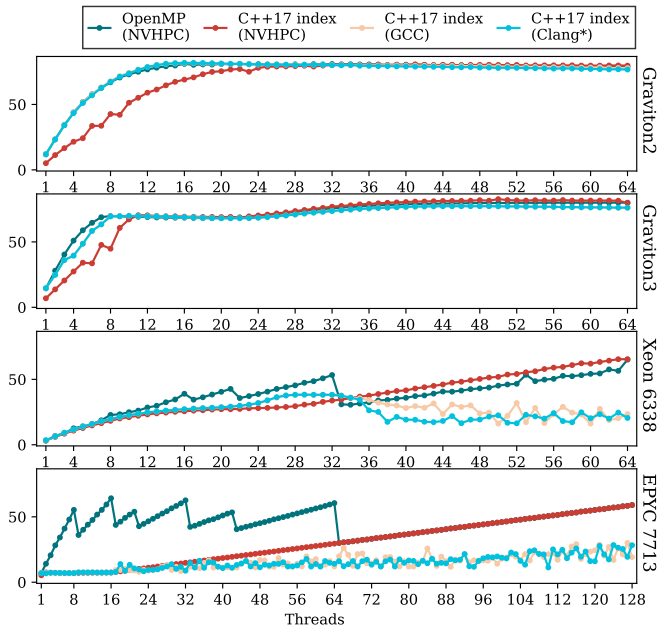


Fig. 2: BabelStream CPU scaling results as % peak bandwidth *per platform*, **higher is better**

we can also infer that the C++ parallel algorithms abstraction did not incur any noticeable overhead. Here, the NVHPC compiler managed to outperform other compilers by around 4%. Given that Graviton3 was only released in May 2022, it is remarkable to see a new micro architecture performing this well.

For x86 CPUs, the results are less clear. On models that are NUMA-aware such as OpenMP and Kokkos, the results are inline with previous literatures[12]. Since the C++17 execution policy in NVHPC is backed by OpenMP, we see a similar performance for both the data and index parallel implementations.

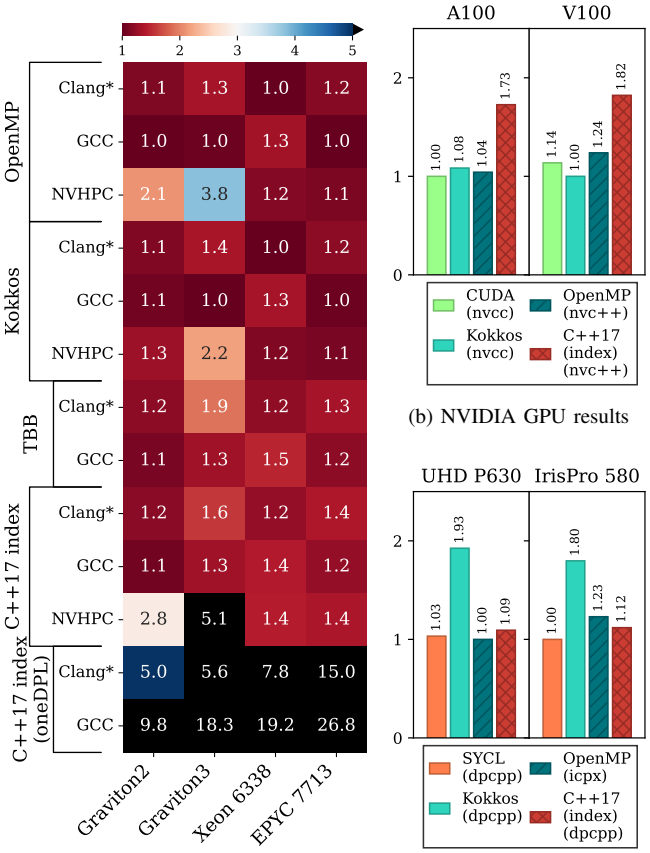
On GCC and Clang, the C++17 execution policy from libstdc++ is backed by TBB, and we observe an equally poor bandwidth on both models. This is supported by the scaling results shown in Fig. 2. Looking at TBB's parallelism implementation, it appears to use a thread pool with no special consideration for NUMA awareness, which leads to suboptimal data placement. Interestingly, TBB offers a *partitioner* variable for a static or dynamic division of tasks; experimenting with all partitions only shows very minor improvement (<5%) compared to the default.

Finally, we compare the oneDPL implementation of C++17 execution policies with plain OpenMP. Surprisingly, despite delegating to OpenMP for parallelism like Kokkos, oneDPL with OpenMP performed only marginally better than the TBB-backed implementations. Looking at oneDPL's implementation, it internally re-chunks the input into a hardcoded 2048 element chunk and then executes the chunk in an OpenMP `taskloop` directive. This effect of this is similar to using a non-NUMA-aware thread pool.

We noticed the Copy kernel compiled with NVHPC reported an above-average bandwidth. Decompilation reveals NVHPC replacing the parallel copy with a direct call to `__c_mcopy4`, NVIDIA's hand-optimised copy routine.

2) GPUs: On both Intel and NVIDIA GPU platforms, the data and index parallel implementations attained bandwidth that is identical or close to the vendor supported model, as shown in Fig. 1c and Fig. 1b. For NVIDIA GPUs, we observe a lower bandwidth for Kokkos and OpenMP target with the Dot kernel. Since the Dot kernel implements a tree-based reduction, optimal performance requires the runtime to pick an optimal block size. We suspect Kokkos and the OpenMP target backend selected a suboptimal block value.

D. miniBUDE



(a) AArch64 and x86 CPU results (b) NVIDIA GPU results (c) Intel GPU results

Fig. 3: miniBUDE results as normalised runtime *per platform*; **lower is better**

The miniBUDE port only implements the index parallel variant discussed in Section III. Normalised runtime results for CPUs are shown in Fig. 3a with scaling results in Fig. 4. Similar to Section VI-C, we compare C++ parallel algorithms implementations to their backing implementation and alternative models.

As miniBUDE is a compute-bound application, the performance characteristics become decoupled from the memory layout; the heatmap in Fig. 3a now highlight how well each compiler optimises away abstraction layers. Overall, the TBB-backed libstdc++ implementation is directly comparable to

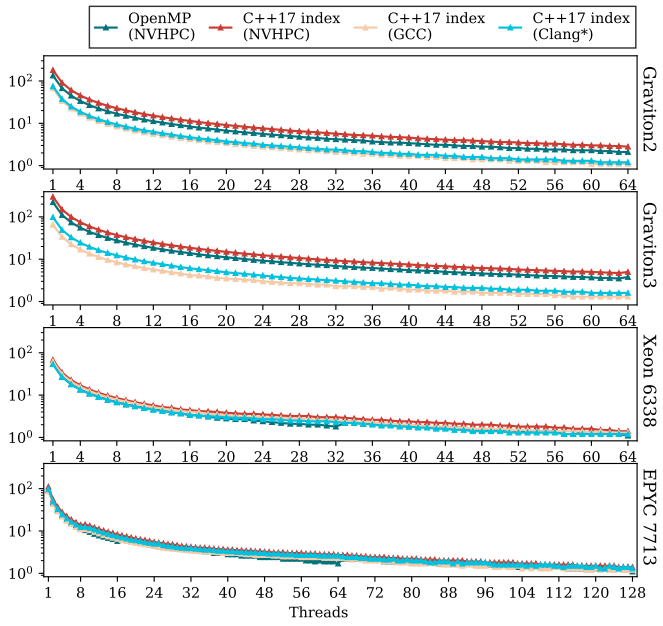


Fig. 4: miniBUDE CPU scaling results as normalised runtime *per platform*; **lower is better**

OpenMP and Kokkos, with a slightly slower average runtime across all platforms.

Unexpectedly, the OpenMP-backed oneDPL performed badly, with runtime almost proportional to the core count. Profiling reveals low processor utilisation (<50%) caused by bad task scheduling with the 2048 chunk size — the same issue we discussed in Section VI-C.

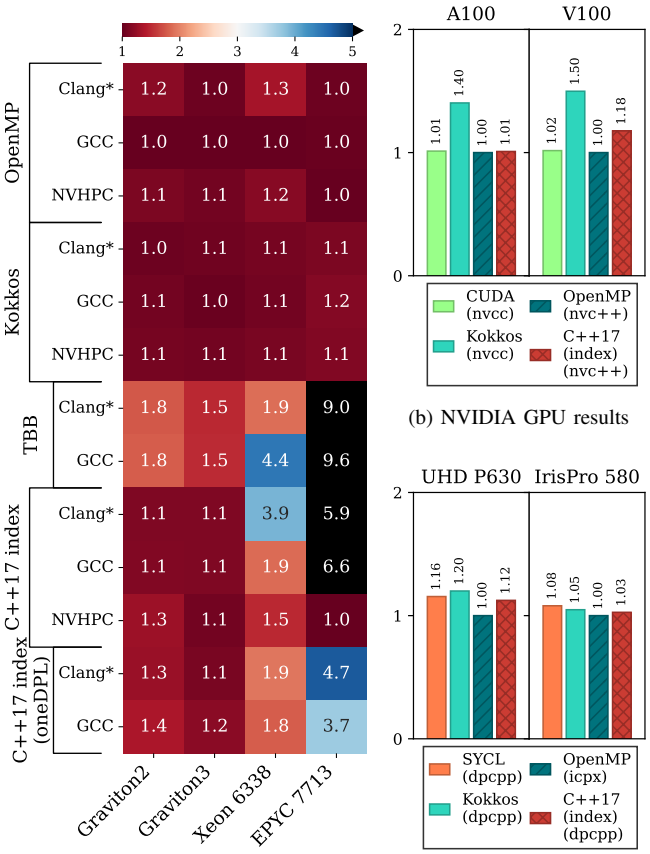
Across models compiled with NVHPC on AArch64 platforms, we see a notable degradation compared to Clang and GCC. We suspect the vectorisation backend in NVHPC is still a work in progress for AArch64; PGI, the compiler NVHPC is based on, did not support AArch64 up until the transition to NVHPC in 2020.

On NVIDIA GPUs, we observe a higher runtime compared to CUDA, OpenMP, and Kokkos. Past exchanges with NVIDIA highlighted the probable cause to suboptimal block sizes and a potential failure to emit an approximated square root intrinsic[14]. Finally, Intel GPUs performed well with little variance across all models, showcasing mature vectorisation support.

E. CloverLeaf

The CloverLeaf port only implements the index parallel variant discussed in Section III. Normalised runtime results for CPUs are shown in Fig. 5a and scaling results in Fig. 6. Similar to BabelStream, CloverLeaf is a memory-bandwidth bound application but with a much higher kernel count. On CPU platforms, we again see NUMA-aware models do well on x86 CPUs, all of which have more than one NUMA node. The overall performance degradation is proportional to the number of NUMA nodes, with the EPYC platform performing the worst at 8 NUMA nodes. In line with BabelStream results for NVHPC, the OpenMP-backed execution policy is the only combination that is comparable to OpenMP and Kokkos.

For GPUs, there are no NUMA nodes so the focus shifts to whether each model can safely and correctly elide data transfers between the host and device in a complex execution graph. For both Intel and NVIDIA GPUs, the C++ index parallel model is comparable to vendor models such as SYCL and CUDA. Kokkos unfortunately introduced an extra kernel launch overhead which increased runtime.



(a) AArch64 and x86 CPU results

(b) NVIDIA GPU results

(c) Intel GPU results

Fig. 5: CloverLeaf results as normalised runtime *per platform*; **lower is better**

F. Optimisation attempts

The C++17 parallel algorithms API does not expose any further optimisation parameters, as touched on in Section III-A and Section III. This section explores whether some performance issues identified in previous sections can be further minimised by patching the implementations themselves.

The OpenMP-backed oneDPL implementation performed poorly on multiple platforms due to the hardcoded chunk size and the use of OpenMP *taskloop*s. The oneDPL codebase reveals an additional re-chunking step that splits ranges into a fixed 2048 element sub-range. Under this structure, we simply removed the re-chunking and replaced the OpenMP directives to operate over a plain for-loop. Fig. 7 and Fig. 8 shows the results of this change compared to the native OpenMP and

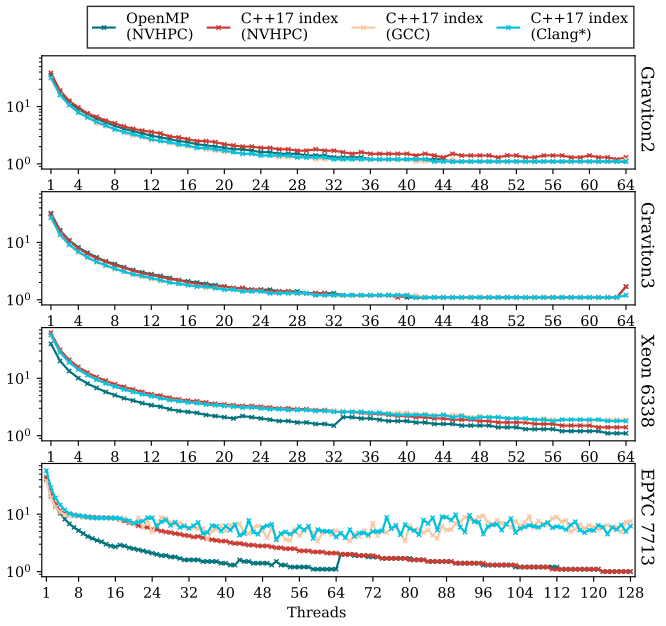


Fig. 6: CloverLeaf results as normalised runtime *per platform*; **lower is better**

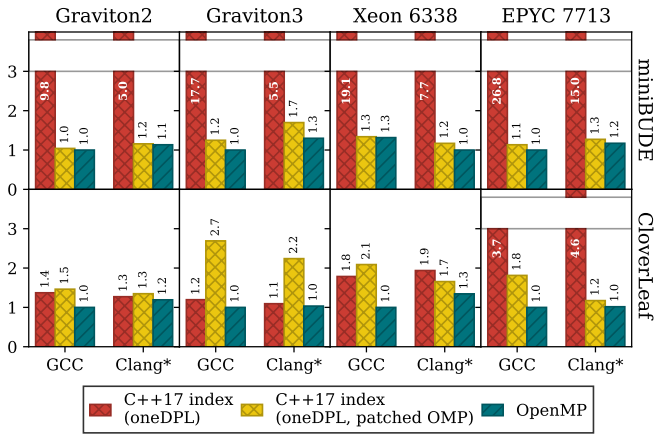


Fig. 7: CloverLeaf/miniBUDE uplift with patched oneDPL

the unpatched oneDPL implementation. Overall, this change brings the oneDPL implementation up to par with the native OpenMP version.

For miniBUDE on NVIDIA GPUs, we have identified a lower than usual occupancy based on profiler output. By patching the NVIDIA *Thrust* headers (NVHPC uses Thrust internally) to use a more optimal block size, we achieved performance equal to CUDA, as shown in Fig. 9.

In both cases, we intend to provide our findings as feedback to the respective vendors.

VII. Conclusion

We implemented three ISO C++17 ports of representative HPC mini-apps that cover both compute-bound and memory bandwidth-bound application spaces. The resulting ports are almost completely portable (only 6 lines away in the worst case) across different compilers for both CPUs and GPUs. In

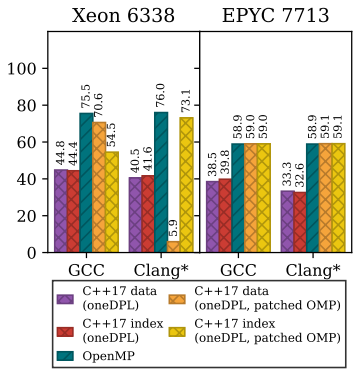


Fig. 8: BabelStream Triad uplift with patched oneDPL

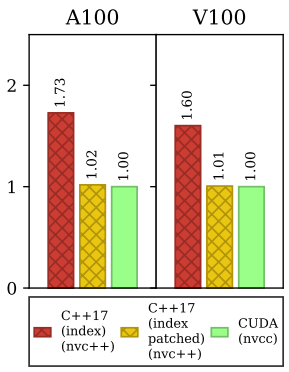


Fig. 9: MiniBUDE uplift with patched NVHPC

effect, we show how HPC codes written in idiomatic ISO C++ can be both performance portable and require no third-party libraries.

Beyond performance portability, this study has demonstrated how traditional HPC programming techniques such index-based traversal are well-supported use cases in ISO C++17. In general, none of the C++17 implementations impose unreasonable requirements on algorithm use: captured-pointers are allowed. Based on the three ports, we conclude that only minimal code transformation is required coming from either a vendor-supported programming model such as CUDA or a portability layer such as Kokkos. In particular, C++17 parallel algorithms implement tuned, non-trivial kernels that greatly improve productivity. For example, CloverLeaf contains multiple complex reduction kernels that reduce a structure. This was easily accomplished with the `std::transform_reduce` algorithm in the documented way.

Finally, we note the absence of explicit device and memory management interfaces in the C++17 API; we anticipate newer C++ standards to address this. Moreover, certain idiomatic C++ patterns, such as using *std::vector*s for memory allocation, are not performance-portable on all platforms. We hope C++20 or newer additions such as `std::span` and `std::mdspan` can fill this gap.

There is a consensus in the HPC community that picking programming models for greenfield projects is difficult: predicting which model will outlive your project is mostly a gamble. As we have shown in this study, we may be closer to a world where heterogeneous systems can be programmed in plain ISO C++.

Acknowledgment

This work makes use of the following services: *Isambard UK National Tier-2 HPC Service* (https://gw4.ac.uk/isambard) operated by GW4 and the UK Met Office, funded by EPSRC (EP/P020224/1); *HPC Zoo*, a research cluster managed by the HPC Group at the University of Bristol (https://uob-hpc.github.io/zoo); *Intel DevCloud*, an online cluster for developers (https://devcloud.intel.com); *EC2 Graviton instances*, with access supported by AWS.

References

[1] J. Hoberock, J. Marathe, M. Garland, O. Giroux, V. Grover, A. Laksberg, H. Sutter, and A. Robison, "A Parallel Algorithms Library | N3554," Mar 2013. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3554.pdf

[2] ISO, *ISO/IEC JTC1 SC22 WG21 N4507 — Programming languages — Technical Specification for C++ Extensions for Parallelism*, May 2015. [Online]. Available: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4507.pdf

[3] D. Olsen, G. Lopez, and B. A. Lelbach, "Accelerating Standard C++ with GPUs Using stdpar," 08 2020. [Online]. Available: https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/

[4] J. Latt, C. Coreixas, and J. Beny, "Cross-platform programming model for many-core lattice Boltzmann simulations," *PLoS ONE*, vol. 16, 2021.

[5] M. Drocco, V. G. Castellana, and M. Minutoli, "Practical Distributed Programming in C++," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 3539. [Online]. Available: https://doi.org/10.1145/3369583.3392680

[6] P. Jääskeläinen, J. Glossner, M. Jambor, A. Tervo, and M. Rintala, "Offloading C++17 Parallel STL on System Shared Virtual Memory Platforms," in *High Performance Computing*, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Cham: Springer International Publishing, 2018, pp. 637–647.

[7] Khronos, "SYCL Parallel STL," Apr 2019. [Online]. Available: https://github.com/KhronosGroup/SyclParallelSTL

[8] J. D. McCalpin *et al.*, "Memory bandwidth and machine balance in current high performance computers," *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19-25, 1995.

[9] 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.

[10] 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*, B. L. Chamberlain, A.-L. Varbanescu, H. Ltaief, and P. Luszczek, Eds. Cham: Springer International Publishing, 2021, pp. 332–350.

[11] A. Mallinson, D. A. Beckingsale, W. Gaudin, J. Herdman, J. Levesque, and S. A. Jarvis, "Cloverleaf: Preparing hydrodynamics codes for exascale," *The Cray User Group*, vol. 2013, 2013.

[12] T. Deakin, A. Poenaru, T. Lin, and S. McIntosh-Smith, "Tracking Performance Portability on the Yellow Brick Road to Exascale," in *Proceedings of the Performance Portability and Productivity Workshop P3HPC*. United States: Institute of Electrical and Electronics Engineers (IEEE), Sep. 020.

[13] 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.

[14] W.-C. Lin, T. Deakin, and S. McIntosh-Smith, "How to Develop Performance Portable Codes using the Latest Parallel Programming Standards | NVIDIA On-Demand," Mar 2022. [Online]. Available: https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41618/

APPENDIX

Artifact Description

We ran BabelStream, miniBUDE, and CloverLeaf on a wide range of hardware platforms listed in Table II.

Artifacts Available: Source code for BabelStream, with all the models presented in this study, is available at https://github.com/UoB-HPC/BabelStream. The *std-data* and *std-indices* models used in this study uses the `option_for_vec` branch which contains considerations for oneDPL portability and NUMA-awareness adjustments.

Likewise, source code for miniBUDE, with all the models presented in this study, is available at https://github.com/UoB-HPC/miniBUDE/tree/v2, with plans to merge the *v2* branch into *main* shortly.

For CloverLeaf, due to the code size, each model resides in a separate repository:

- CUDA:https://github.com/UK-MAC/CloverLeaf_CUDA
- TBB:https://github.com/UoB-HPC/cloverleaf_tbb
- SYCL:https://github.com/UoB-HPC/cloverleaf_sycl
- C++17:https://github.com/UoB-HPC/cloverleaf_stdpar
- Kokkos:https://github.com/UoB-HPC/cloverleaf_kokkos
- OpenMP:https://github.com/UoB-HPC/cloverleaf_openmp_target/tree/omp-plain
- OpenMP target:https://github.com/UoB-HPC/cloverleaf_openmp_target/tree/omp-target

We have created scripts to help make the results in this paper reproducible. The source code can be found at https://github.com/UoB-HPC/performance-portability/tree/2022-benchmarking. The script handles setting platform specific optimisations flags for each compiler and also includes job scripts for launching the experiments. The job script includes optimal platform and model specific thread pinning strategies (e.g. `OMP_PROC_BIND` or direct taskset/numactl, etc).

Additional benchmarking for oneDPL experiments with and without patch can be found at https://github.com/UoB-HPC/performance-portability/tree/2022-benchmarking-dplomp-experiment.

Experimental setup: See Table II for a list of hardware platforms used and Section VI-A for versions on the software stack.

Artifact Evaluation

Performed verification and validation studies: Each mini-app contains built-in verification for correctness. For BabelStream, the results are validated against a simple host version that implements all the kernels. For miniBUDE and CloverLeaf, the results are validated against the known values of the input deck.

Validated the accuracy and precision of timings: For BabelStream, benchmark measurements use the best result over 100 runs. For miniBUDE, benchmark measurements contain warm-up and measurements are obtained using the best result over 8 runs. For CloverLeaf, benchmark measurements is set at 3k (0.3k for scaling results) timesteps and measurement is done using the built-in profiler for accuracy.

In all benchmark runs, we cross-check results with existing literature where possible.

Used manufactured solutions or spectral properties: N/A

Quantified the sensitivity of your results to initial conditions and/or parameters of the computational environment: Both BabelStream and miniBUDE contained warm-up iterations or equivalent. CloverLeaf is designed to only measure runtime during simulation time steps.

Describe controls, statistics, or other steps taken to make the measurements and analyses robust to variability and unknowns in the system: N/A