Comparing Julia to Performance Portable Parallel Programming Models for HPC

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

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

***Abstract*—Julia is a general-purpose, managed, strongly and dynamically-typed programming language with emphasis on high performance scientific computing. Traditionally, HPC software development uses languages such as C, C++ and Fortran, which compile to unmanaged code. This offers the programmer near bare-metal performance at the expense of safety properties that a managed runtime would otherwise provide. Julia, on the other hand, combines novel programming language design approaches to achieve high levels of productivity without sacrificing performance while using a fully managed runtime.**

This study provides an evaluation of Julia's suitability for HPC applications from a performance point of view across a diverse range of CPU and GPU platforms. We select representative memory-bandwidth bound and compute bound mini-apps, port them to Julia, and conduct benchmarks across a wide range of current HPC CPUs and GPUs from vendors such as Intel®, AMD®, NVIDIA®, Marvell®, and Fujitsu®. We then compare and characterise the results against existing parallel programming frameworks such as OpenMP®, Kokkos, OpenCL™, and first-party frameworks such as CUDA®, HIP™, and oneAPI™ SYCL™. Finally, we show that Julia's performance either matches the competition or is only a short way behind.

***Index Terms*—Julia, OpenMP, OpenCL, Kokkos, CUDA, HIP, Performance Portability, Programming Models, GPUs**

I. Introduction

A. Background

HPC programming traditionally revolves around compiled languages that generate native (i.e. unmanaged) code. Unsafe programming languages such as C, C++ and Fortran have been used extensively in this field due to their excellent portability and performance characteristics; language constructs frequently map directly to the underlying hardware features. The ubiquity of C/C++ also extends to discrete accelerators such as GPUs, which typically execute code compiled from C/C++-derived kernels. These properties are highly desirable in an HPC setting where using all allocated hardware resources optimally is critical.

Unsafe languages such as C and C++ suffer from productivity pain points such as the lack of memory safety and a minefield of undefined behaviours. Programmers are required to manage memory use explicitly which is a common source of bugs, and debugging memory corruption issues is often highly challenging in large applications. In addition, for legacy and performance reasons, the compiler is unable to reject programs with illegal runtime behaviours, thus causing hard-to-catch behavioural bugs. There are existing solutions such as various compiler-supported *sanitisers* [1]; these work by instrumenting memory allocation for the whole program, and as such, are usually intended only for debugging due to the high runtime overhead. Recent iterations of C++ have made advances in offering better compile-time guaranteed safety, we also see new languages that directly address memory safety at the type-level, as seen in Rust. However, many of these advancements are a calculated tradeoff between runtime performance and programmer productivity.

By having a managed runtime that performs bounds checking and automatic memory management, we eliminate an entire class of bugs without sacrificing programmer productivity. For years, the HPC community has dismissed the use of languages that require a managed runtime from the outset. Most existing implementations, such as the Java Virtual Machine, were unable to produce code that could directly compete with C/C++ or Fortran in terms of performance. In Java's case, language semantics such as reflection and the lack of value types make it prohibitively difficult to generate optimal machine code. Beyond language semantics, the added layer of indirection also complicates support for custom accelerator hardware, as the runtime must also be able to generate accelerator-specific machine code.

B. Julia

Julia was first released in 2012 as a new dynamically-typed, general-purpose, multi-paradigm programming language designed with high performance scientific computing in mind [2]. Julia is a managed programming language featuring an LLVM-backed JIT engine. The language combines many novel ideas that are not usually found in traditional bare-metal languages such as C and C++. As a dynamically typed language, Julia supports parametric polymorphism (generics) through optional typing. Paired with multiple dispatch, these features allow enough type information to be encoded at *runtime* for the JIT compiler to specialise and create optimal code.

Many of the features and restrictions of the language are guided by the potential for net performance gains even after the overhead of JIT compilation. Julia only selectively included features from Object-Oriented and Functional paradigms to

avoid creating situations like Java where the language semantics inhibit high performance code generation. In addition, Julia also provides excellent metaprogramming and staging features along with runtime-supported foreign function interface (FFI) capabilities. With these features, Julia is in a unique position where pure Julia code can be staged and transformed to native accelerator code at runtime for execution on GPUs. Currently, all major accelerator vendors, including NVIDIA, AMD, and Intel are supported under the *JuliaGPU* initiative.

C. Contribution

This study provides an evaluation of Julia's readiness for use in implementing HPC applications. We accomplish this by porting HPC mini-apps, ones originally written in C/C++ or derivatives, to idiomatic Julia. We then compare the performance achieved in Julia to the original and provide an analysis of the results. For the best possible platform coverage, we select frequently used hardware in typical HPC clusters (as shown in Table II) to conduct our benchmarks against. CPU platforms include traditional x86 vendors Intel and AMD, and also emerging Arm platforms from Fujitsu, Marvell, and Apple®. For GPUs, we select from current vendors such as NVIDIA, AMD, and Intel. We include both HPC and consumer models as there has been a constant interest in using lower-cost consumer-grade hardware for HPC use cases.

This study makes the following contributions:

- We survey the state of the art for implementing HPC applications in Julia for both CPUs and GPUs.
- We present Julia ports of memory-bandwidth bound (BabelStream) and compute-bound (miniBUDE) benchmark mini-apps for a representative performance portability evaluation of HPC applications in Julia.
- We benchmark Julia ports of the HPC mini-apps on a wide range of representative HPC hardware platforms against first party solutions and other established parallel programming frameworks.

II. Related work

Performance portability is an important quality for HPC applications. Our previous work on surveying performance portability over a diverse range of HPC hardware showed that many languages and frameworks based on C/C++ achieve a high level of performance portability [3, 4]. This work attempts to evaluate Julia in the same way using HPC-style mini-apps. As Julia is an emerging programming language and considerably different from traditional unmanaged languages, we take a more fine-grained approach to the ecosystem and performance analysis in this study.

Julia, while being a relatively new language to the HPC scene, already has several case studies on implementing full-scale HPC applications. Regier et al. experimented with implementing bayesian inference for large datasets using Julia that runs on a cluster of 8192 Xeon Phi nodes [5]. Based on the new work of Besard et al. on bringing Julia to GPUs [6], we have seen several successful HPC application implementations in Julia. Ramadhan et al. have demonstrated an efficient fluid flow solver, *Oceananigans.jl*, that runs on both CPUs and NVIDIA GPUs [7]. In the same vein, Clements et al. have implemented a toolkit, *SeisNoise.jl*, for noise cross-correlation on both GPUs and CPUs [8].

On the machine learning side, thanks to Julia's expressiveness, an abstraction layer implemented by Innes, *Flux.jl*, showed that Julia is capable of realising a fully functional ML stack [9]. For traditional HPC applications, Ram et al. explored performance characteristic of implementing a mesh-free CDF solver in Julia [10] on the GPU, among other languages like Python and C++. Finally, Hunold et al. have explored the performance of the STREAM benchmark on CPUs implemented in Julia along with a characterisation of MPI communication overheads in Julia [11].

However, we have yet to see a comprehensive characterisation on the performance portability properties of Julia on contemporary HPC platforms. In particular, many case-studies only focus on a specific Julia package and lack direct comparison across either platform or language, or both. This study hopes to address this gap to show whether Julia is ready for production use in the HPC domain.

III. Julia ecosystem

Julia provides first class support for threads through high level OpenMP-like macros. Fig. 1 shows a side-by-side comparison of a simple parallel copy. On the right, the loop is executed in parallel with OpenMP pragmas. On the left, Julia provides the `Threads.@threads` macro that transforms the loop block into a statically scheduled [12] parallel loop. The `@inbounds` macro here simply hints that it is safe to elide bound checks at runtime.

```julia
xs = Vector{Float32}(undef, N)
ys = fill(42f0, N)
Threads.@threads for i = 1:N
  @inbounds xs[i] = ys[i]
end
```

```cpp
std::vector<float> xs(N, 0);
std::vector<float> ys(N, 42.f);
#pragma omp parallel for
for (int i = 0; i < N; i++)
  xs[i] = ys[i];
```

Fig. 1. Threaded copy. Left: Julia, right: C++ w/ OpenMP

```julia
# if ascribed: a::AbstractArray{T}, b::AbstractArray{T}, s::T
function mul(b, c, s) # type annotation optional
  i = (blockIdx().x - 1) * blockDim().x + threadIdx().x
  @inbounds b[i] = scalar * c[i]
  return
end
```

```cpp
template <typename T>
__global__ void mul(T* b, const T* c, const T s) {
  const int i = blockDim.x * blockIdx.x + threadIdx.x;
  b[i] = s * c[i];
}
```

Fig. 2. Polymorphic multiply kernel. Top: Julia (CUDA.jl), bottom: CUDA

Julia's support for GPU offload is provided by third-party packages. This is possible due to its highly flexible metaprogramming facilities along with the runtime's performant FFIs. The JuliaGPU umbrella implements support for CUDA, HIP and oneAPI with separate packages that wrap the native

interfaces provided by the respective offload solution. Table I gives an overview of the constructs employed by each of the GPU packages that provide support for writing kernels in Julia directly. Fig. 2 shows a comparison between a simple polymorphic multiply kernel written in Julia (via the CUDA.jl package) and CUDA, the Julia CUDA kernel shown here can be easily ported to other implementations using Table I. Note the high level of similarity between the two implementations: Julia is a C-influenced imperative language at heart.

CUDA.jl, as the package name suggests, provide CUDA support for running on NVIDIA GPUs. The kernel API surface here closely follows the CUDA nomenclature. The package acts as a runtime for kernel submission and memory management and is responsible for generating PTX assemblies for kernels.

Similar to CUDA.jl, *AMDGPU.jl* provides HSA support for running on AMD GCN GPUs. The API surface follows HSA nomenclature, although the package provides function aliases for CUDA nomenclature to ease transition.

OneAPI.jl provides Level Zero support for running on Intel GPUs. It largely follows the OpenCL and SYCL nomenclature. At the current state, oneAPI.jl only supports Level Zero as an offload target (i.e. Intel GPU only) as it does not mirror the full functionality of Intel's oneAPI distribution. Both AMDGPU.jl and oneAPI.jl have stated that the projects are still under development, although core functions such as kernel execution are available for testing.

Finally, *KernelAbstractions.jl* (abbreviated as *KA.jl* hereafter) is a kernel portability abstraction layer where kernels written in KA.jl can be executed on NVIDIA, AMD and CPU platforms without source changes. Currently, the package warned that no specific tuning has been done for CPU and AMD platforms so performance may be suboptimal. In addition, oneAPI for the Intel platform is a notable omission as it is considerably newer than CUDA.jl and AMDGPU.jl.

In all cases, compute kernels are implemented as plain Julia functions with some restrictions on supported types within the kernel. In this regard, Julia's GPU support is similar to OpenMP Target or Kokkos in that the driver and the kernel share the same language in a single-source format. However, for optimal performance, Julia uses different packages for each vendor. As each package is developed independently, the kernels are not directly portable because the packages expose a different kernel API, not unlike the native first-party solutions. Nevertheless, all GPU packages adhere closely to idiomatic Julia; the porting effort required to go from one API to another typically only involves mechanical substitution of the API calls by using a cross-reference table like Table I.

IV. Mini-apps in the study

We used two existing HPC mini-apps, *BabelStream* and *miniBUDE*, to cover both memory-bandwidth bound and compute bound scenarios respectively. These two mini-apps are specifically designed for performance portability studies. The codebase contains multiple implementations of the same compute kernels implemented in different parallel programming frameworks.

A. BabelStream

BabelStream is a memory bandwidth benchmark that implements the standard McCalpin STREAM benchmark [13] for both CPUs and GPUs for a wide variety of programming models. By implementing STREAM in many programming models, BabelStream has been invaluable in helping us investigate performance portability properties of common parallel programming frameworks [14]. The benchmark implements five different memory-bandwidth bound kernels: Copy, Mul, Add, Triad, and Dot. The pseudocode for these kernels are shown in Algorithm 1. The benchmark measures the execution time for each kernel and derives the average bandwidth with respect to the array size of each input using the best performing iteration.

The BabelStream codebase is designed to be an exemplar in demonstrating each framework's approach in achieving parallelism: constructs are specifically chosen to be idiomatic and generic. By comparing the different implementations side by side, one can make rough estimates on the potential productivity advantages of each framework.

Algorithm 1 BabelStream kernels

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

For this benchmark, we create an idiomatic Julia port of BabelStream, named *JuliaStream.jl*. JuliaStream.jl implements support for multicore CPUs via Julia's `Threads.@thread` macro in a straightforward manner as shown in Fig. 1. As Julia's threading macro lacks support for reductions, we have implemented the Dot kernel reduction by first summing the results to a thread local variable. We then complete the summation for each thread at the end in serial.

For GPUs, we have created implementations using CUDA.jl for CUDA, AMDGPU.jl for HIP, oneAPI.jl for SYCL, and finally KA.jl for cross-platform programming models. Each of these separate implementations is directly derived from their respective BabelStream implementations. For Julia to perform

TABLE I
JULIA GPU KERNEL API CROSS-REFERENCE

	CUDAjl	AMDGPU.jl	oneAPI.jl	KernelAbstractions.jl
Global size	`blockDim().x*gridDim().x`	`gridDim().x`	`get_global_size(0)`	(Not exposed)
Group count	`gridDim().x`	`gridDimWG().x`	`get_num_groups(0)`	(Not exposed)
Group size	`blockDim().x`	`workgroupDim().x`	`get_local_size(0)`	`@groupsize()[1]`
Index (global)	`(blockIdx().x - 1) * blockDim().x + threadIdx().x`	`(workgroupIdx().x - 1) * workgroupDim().x + workitemIdx().x`	`get_global_id(0)`	`@index(Global)`
Index (group)	`blockIdx().x`	`workgroupIdx().x`	`get_group_id(0)`	`@index(Group)`
Index (local)	`threadIdx().x`	`workitemIdx().x`	`get_local_id(0)`	`@index(Local)`
Synchronise	`sync_threads()`	`sync_workgroup()`	`barrier()`	`@synchronize`
Memory (register) size=N, type=T	`data = MArray{Tuple{N},T}`	`data = MArray{Tuple{N},T}`	`data = MArray{Tuple{N},T}`	`data = MArray{Tuple{N},T}`
Memory (private) size=N, type=T	`data = @cuStaticSharedMem(T, N)`	`data = ROCDeviceArray((N,), alloc_local(:data,T, N))`	`data = @LocalMemory(T, (N,))`	`data = @localmem T N`

Kernel launch & Mem. allocation Groups=GN, Blocks=BN, size=N, type=T	`data = CuArray{T}(undef, N)`
	`@cuda`
	`  blocks=N÷GN`
	`  threads=GN`
	`  kernel(data)`
		`data = ROCArray{T}(undef, N)`
		`@roc`
		`  groupsize=GN`
		`  gridsize=GN*BN`
		`  kernel(data)`
			`data = oneArray{T}(undef, N)`
			`@oneapi`
			`  items=GN`
			`  groups=BN`
			`  kernel(data)`
				`device= # one of :`
				`  # CPU()`
				`  # CUDADevice()`
				`  # ROCDevice()`
				`data= # one of :`
				`  # Array{T}`
				`  # CuArray{T}`
				`  # ROCArray{T}`
				`kernel(device, GN)(`
				`  data, ndrange=N)`

well, we require it to achieve a bandwidth similar to the original BabelStream versions for each of the platforms we conduct the benchmark on.

BabelStream accepts the array size, array type, and iteration count as parameters. Array size determines how large each of the three arrays (*A, B, C*) in Algorithm 1 will be. We selected 2^{29}(≈12.9GB) elements for CPUs. This size was selected based on experiences using BabelStream on processors with very large L3 caches [3]. We selected a smaller 2^{25}(≈0.8GB) for GPUs as some devices do not support allocating arrays much larger than this value. The remainder of BabelStream's parameters are left at the defaults (FP64 array type at 100 iterations for each kernel). We make use of the built-in statistics reporting to collect our results: BabelStream includes result validation, warmup iterations, and automatically derives bandwidth of the best and worst performing iterations at the end of the benchmark.

Julia will be compared against the OpenMP and Kokkos implementations from BabelStream for all CPUs. For NVIDIA GPUs, we compare CUDA.jl and KA.jl implementations to CUDA, Kokkos and OpenCL. For AMD GPUs, we compare AMDGPU.jl and KA.jl implementations to HIP, Kokkos and OpenCL. For Intel GPUs, we compare oneAPI.jl implementations to oneAPI SYCL and OpenCL.

B. miniBUDE

MiniBUDE is a compute bound HPC mini-app derived from the full-scale Bristol University Docking Engine (BUDE) molecular dynamics application [15]. The core *virtual-screening* algorithm computes the charge interactions between molecules when docked in different poses, a process used for drug discovery.

Algorithm 2 miniBUDE Fasten Kernel

1: **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

2: **for** $il \leftarrow 0, ls$ **do**
3: $lpos_{1\times3} \leftarrow xform \cdot ligands[il].pos_{1\times3}$
4: **for** $ip \leftarrow 0, ps$ **do**
▷ Compute distance between atoms
5: $dist \leftarrow distance(lpos, proteins[ip].pos_{1\times3})$
▷ Compute the sum of the sphere radii
6: $d \leftarrow dist - R$
▷ Compute steric energy
7: $energy[i] \leftarrow energy[i] +$ $(1 - dist * (1/R)) *$ $(d < 0 ? 2 * HRD : 0)$
▷ Compute formal and dipole charge interactions
8: $e \leftarrow init *$ $(d < 0.f ? 1 : (1 - d * DST_1)) *$ $(d < DST ? 1 : 0)$
9: $energy[i] \leftarrow energy[i] +$ $(typeE ? -|e| : e) * T$
▷ Compute Nonpolar-Polar repulsive interactions
10: $dslvE = dslvInit *$ $((d < DSLV \wedge NZ) ? 1 : 0.f) *$ $(d < 0 ? 1 : (1 - d * DSLV_R))$
11: $energy[i] \leftarrow energy[i] + dslvE * 0.5$

TABLE II
PLATFORM DETAILS

Vendor	Name	Architecture	Abbreviation	Device Type	Theoretical Peak Mem. Bandwidth (GB/s)	Theoretical Peak FP32 FLOP/s (GFLOP/s)
Intel	Xeon Gold 6230	Cascade Lake	Xeon	HPC CPU (20C*2, 2S)	281.6	4096
AMD	EPYC 7742	Zen2 (Rome)	EPYC	HPC CPU (64C*2, 2S)	409.6	9216
Marvell	ThunderX2	Vulcan	TX2	HPC CPU (32C*2, 2S)	288	2560
Fujitsu	A64FX	(Custom ARMv8)	A64FX	HPC CPU (48C*2, 2S)	1024	5530
NVIDIA	Tesla A100 (SXM 80GB)	Ampere	A100	HPC GPU	2039	19490
NVIDIA	Tesla V100 (PCIe 16GB)	Volta	V100	HPC GPU	900	14130
NVIDIA	RTX2080Ti	Turing	2080Ti	Consumer GPU	616	13800
AMD	Instinct MI100	CDNA	MI100	HPC GPU	1228	23100
AMD	Instinct MI50	GCN (Vega 20GL)	MI50	HPC GPU	1024	13300
AMD	Radeon VII	GCN (Vega 20)	RadeonVII	Consumer GPU	1024	13800
Intel	UHD P630 (Xeon E2176G)	Gen9.5	UHD	Server iGPU	42.6	460
Intel	IrisPro 580 (i7 6670HQ)	Gen9	IrisPro	Consumer iGPU	34	1094
Apple	M1	Firestorm+Icestorm	M1	Consumer CPU(4+4, 1S)	68.25	673

MiniBUDE is a highly condensed application; the main (only) compute kernel attempts to dock a Cartesian set of protein and ligand atoms together using an array of predefined transformation poses, as detailed in Algorithm 2. The kernel requires computing the euclidean norm between atoms and 3D matrix transformations for positions of atoms in a tight loop. These operations involve heavy use of single-precision square roots, trigonometric functions, and absolute values.

The miniBUDE mini-app was designed with similar objectives to BabelStream, albeit with a smaller set of programming models. Similar to the JuliaStream.jl port, we create a *miniBUDE.jl* port for miniBUDE to compare performance. As miniBUDE contains all the ports needed (Kokkos, OpenMP, CUDA, HIP, OpenCL, SYCL) to achieve the same implementation coverage for BabelStream as described in Section IV-A, we conduct the same set of comparative benchmarks with *miniBUDE.jl*.

MiniBUDE accepts an input deck, number of poses, and the iteration as parameters. We left everything as default, using the `bm1` deck at 65536 poses for 8 iterations for the results to be consistent with our previous work in [15]. The mini-app contains built-in validation procedures where the computed values are checked against the known correct values of the input deck. The benchmark also includes additional warm-up iterations and procedures to derive the final arithmetic intensity.

V. RESULTS

For each mini-app, we conduct benchmarks on a diverse set of platforms selected to represent the cutting edge of CPU and GPU architectures available at the time of writing. These are listed in Table II.

A. Platform software setup

For all x86 CPU platforms, we select the latest version of Julia: 1.6.2. For the Arm 64 bit (AArch64) machines, we use Julia 1.7.0 on the Apple M1, since that is the first release that supports this platform without Rosetta (Apple's x86 to AArch64 translation layer), while for the TX2 and A64FX we used both Julia 1.6.2 and 1.7.0. As Julia 1.7.0 is not yet officially released at the time of writing, we select the beta 4 release of this version.

For Julia's GPU support, we provide isolated `Project.toml` build files for each platform in both BabelStream and miniBUDE. By doing this, we reduce the chance of each implementation's transient dependencies from preventing the latest possible version being selected. At the time of writing, by inspecting `Manifest.toml`, we got 3.4.2 for CUDA.jl, 0.2.12 for AMDGPU.jl, 0.2.0 for oneAPI.jl, and 0.7.0 for KA.jl.

We used LLVM11 to compile OpenMP and Kokkos for all of our CPUs. This matches the version used internally in Julia 1.6.2. For A64FX, we use Arm's distribution of Clang (Arm Compiler for Linux), version 21.0, which is based on LLVM11. For Kokkos, we select the latest publicly available version, 3.4.01 (released May 2021), across all platforms.

Every effort has been made to match up driver and software versions on GPU platforms. However, many of these platforms are production clusters that operate on a time-sharing basis so we do not have the appropriate permissions to change driver versions.

For Nvidia's V100 GPU, the driver is reported as version 460.32 with CUDA 11.2. For A100, we got version 470.57 with CUDA 11.4, and finally, for 2080Ti we got 418.39 with CUDA 10.1.

For AMD's MI50 and MI100 GPU, the system uses version 4.2.0 of the ROCm stack. For RadeonVII, the system uses a slightly older ROCm 3.10.

For Intel's IrisPro GPU, the driver self-reported as version 20.49.18626. For UHD, we get version 21.28.20343. For compiling SYCL implementations, we used oneAPI version 2021.3.0 on UHD and 2021.1.0 for IrisPro.

B. BabelStream

In this section we present performance results for the BabelStream benchmark.

Early on in the porting process, it was discovered that KA.jl has a bug with private memory use in for-loops when used in

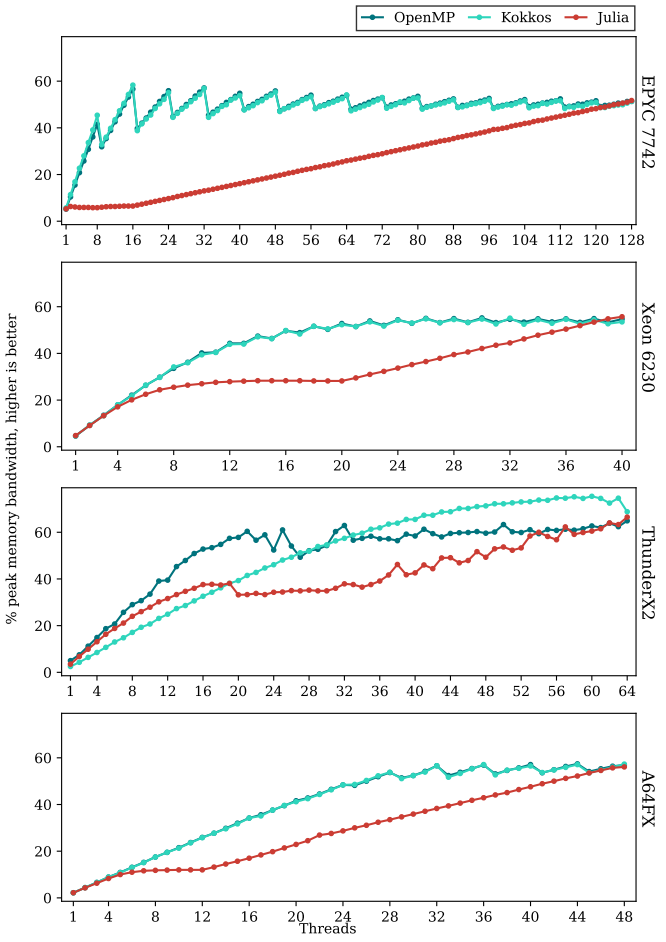

Fig. 3. BabelStream CPU scaling (Triad kernel)

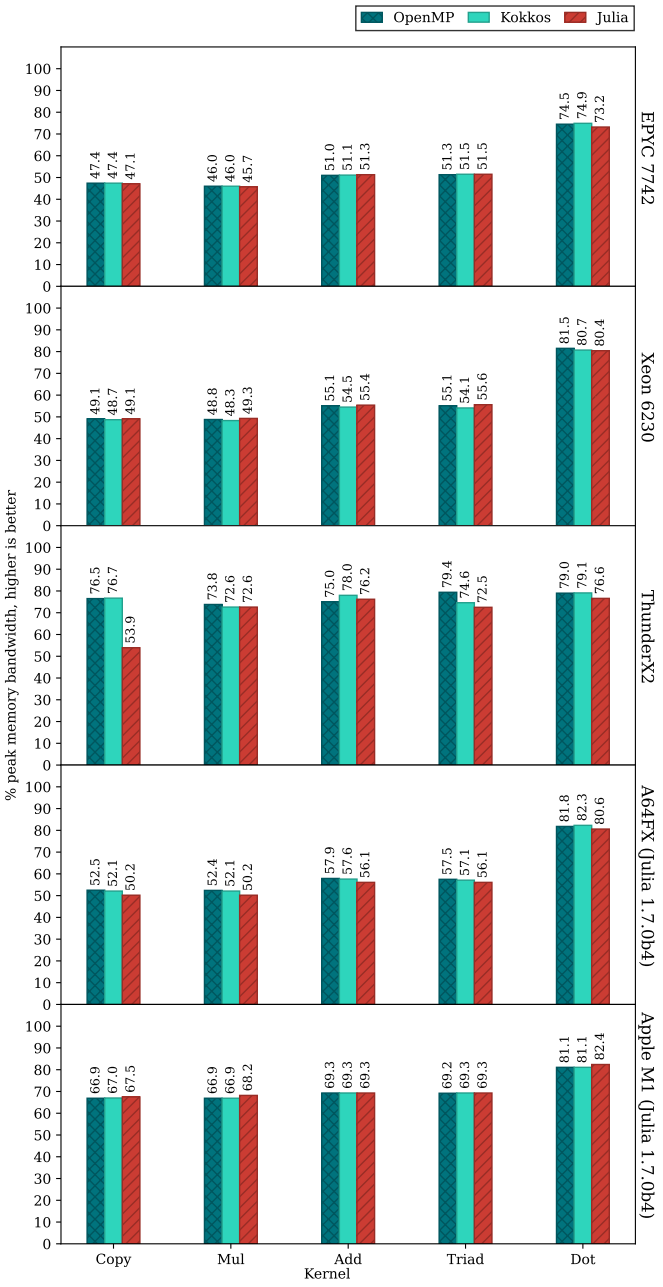

Fig. 4. BabelStream CPU results

conjunction with `@synchronize`, though only on CPUs. The bug occurred at macro-expansion time so we were unable to collect any results for BabelStream's KA.jl implementation on the CPU [16].

The overall porting process is straightforward compared to other C/C++ based BabelStream implementations. All GPU packages share the same semantics for memory management and kernel execution; the kernels themselves are identical across ports if we simply substitute the kernel API calls by referencing Table I. In addition, for GPUs, we find that we are still able to write idiomatic Julia with no major syntactical changes from the CPU port. Because of Julia's execution model, testing a new GPU port only requires an additional entry in the project's dependency declaration. This is a sharp contrast to the native solutions where compiling the kernels in a separate compiler is the norm.

1) CPUs: For CPUs, Julia on x86 platforms showed an impressive, near-identical performance, compared to OpenMP and Kokkos implementations, as shown in Fig. 4.

On the AArch64 CPU platforms, Julia's performance is again very similar to the OpenMP and Kokkos results.

While benchmarking, we observed an anomaly with Julia, OpenMP, and Kokkos results for the Dot kernel on A64FX, and to a lesser extent, TX2. As the Dot kernel implements a reduction, it is unique from the rest of the four kernels.

Experimenting with several reduction implementations showed that, for OpenMP, the compiler was unable to re-order the summation and multiplications in the hot-loop in presence of OpenMP's reduction clause, thus inhibiting vectorisation. We verified this by adding the following compiler flags: `-fno-signed-zeros -fno-trapping-math -fassociative-math` which showed performance similar to what we got on x86 platforms. As adding these unsafe math flags changes semantic for the whole program, we opted to add a Clang pragma that controls floating point associativity

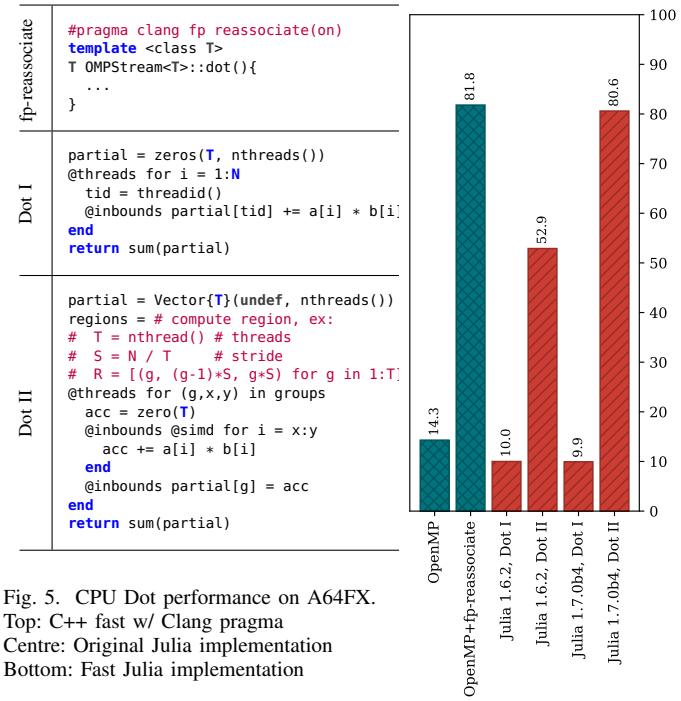

Fig. 5. CPU Dot performance on A64FX.
Top: C++ fast w/ Clang pragma
Centre: Original Julia implementation
Bottom: Fast Julia implementation

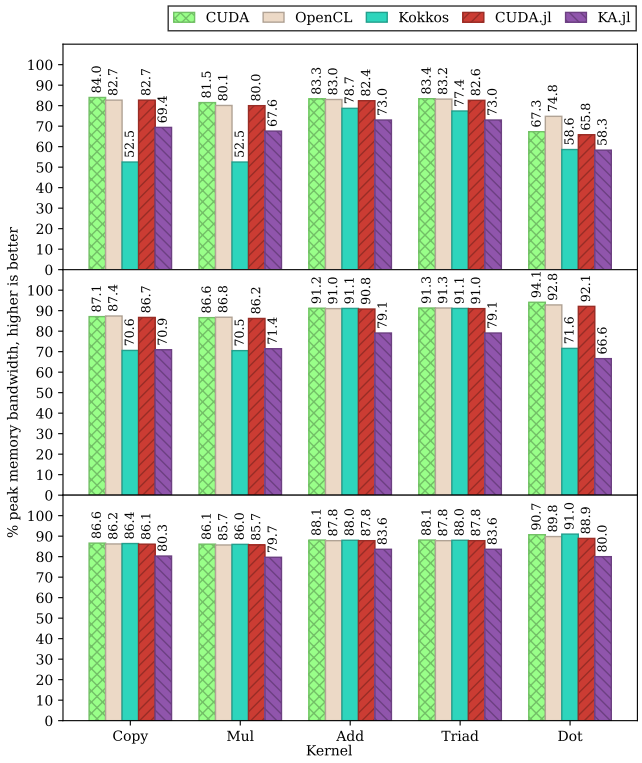

Fig. 6. BabelStream Nvidia GPU results

of a specific block. The pragma and effect of this change is shown in Fig. 5. Kokkos inherited OpenMP's behaviour and using the same pragma resolved the issue as Kokkos delegates to OpenMP for parallelism here too.

For Julia, we experimented with adding localised `@fastmath` macros but with no meaningful improvement. In the end, we rewrote the reduction to expose the dot product reduction as a thread private operation as shown in the Dot II implementation in Fig. 5. With the help of the `@simd` macro, Julia was able to successfully produce vectorised code. When tested on version 1.7.0 of Julia, we were able to match the performance of ArmClang on A64FX.

Another anomaly we have identified is the suboptimal performance of the Copy kernel on the TX2 as shown in Fig. 4. The issue is corrected by inserting additional arithmetic operations (e.g. `c[i] = a[i] + 0`) in the assignment expression. Looking at the generated ARM assembly, we suspect LLVM's cost model failed to emit optimal code.

We present scaling results in Fig. 3 on CPU platforms, the M1 platform is omitted here because the cores are not symmetrical due to the use of Arm's *big.LITTLE* architecture. Here, Kokkos and OpenMP results use the *spread* placing (via `OMP_PROC_BIND=spread`) for the best possible memory bandwidth utilisation. It seems that Julia, by default (with only the following environment variables: `JULIA_EXCLUSIVE=1` and `JULIA_NUM_THREADS=$(nproc)`) scales similarly to the effect of using the *close* placing in OpenMP where threads are allocated close to the parent thread. Looking into the source of the `Threads.@threads` macro, we find that it invokes the C function `jl_set_task_tid` inside a for-loop to initialise each thread; threads are assigned a linear id hence similar to OpenMP's close binding. It is not clear whether Julia currently implements any mechanism for configuring thread affinity policies, tools like `numactl` may be able to counteract this as a workaround.

2) GPUs: For GPUs, both CUDA.jl (Fig. 6) and oneAPI.jl (Fig. 8) managed to match the first-party implementation. This is particularly impressive for oneAPI.jl which is still in early stages of development. For KA.jl on Nvidia GPUs, the performance achieved is only a small step behind (~13%) Kokkos even for the worst case. For AMD GPUs, AMDGPU.jl performed slightly worse (~20%) than the HIP and OpenCL implementation for simple kernels, as shown in Fig. 7. The dot kernel presented a challenge even for HIP; we see similar levels of reduction in performance going from HIP to AMDGPU.jl and KA.jl. We attribute this to AMDGPU.jl's development status, as the project is still under heavy development and not ready for production use. For KA.jl on AMDGPUs, the performance is identical to the vendor-specific AMDGPU.jl package, indicating KA.jl's API surface maps more directly to AMDGPU.jl, and in turn, HSA semantics. Independent of programming language and framework, there appears to be a lack of optimisation on AMD's software stack as the overall performance is slightly lower than on Nvidia and Intel platforms.

C. miniBUDE

In this section we present performance results for the miniBUDE benchmark.

Porting miniBUDE to Julia showed just how close the mapping is from Julia to the underling LLVM semantics. In

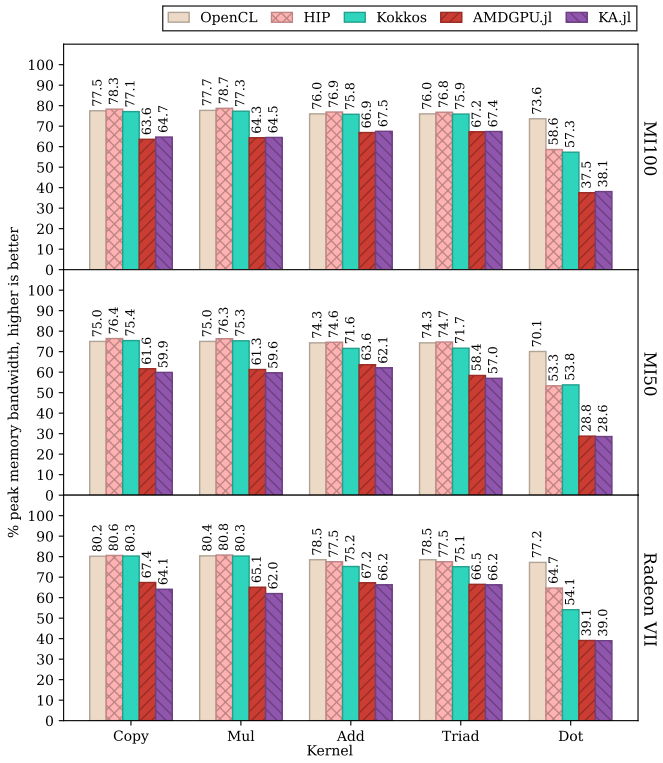

Fig. 7. BabelStream AMD GPU results

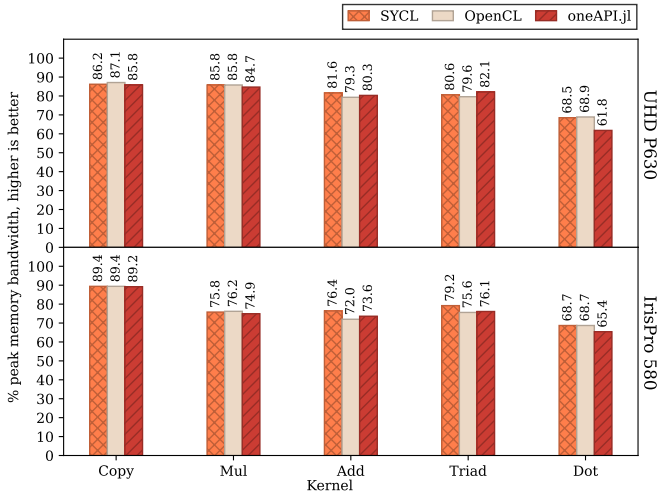

Fig. 8. BabelStream Intel GPU results

the extreme case, certain optimisations used in optimising C/C++ code remain effective in Julia. For example, the use of unsigned integers for induction in the hot loop still produces suboptimal kernels for both CPUs and GPUs. As a side effect of this proximity to LLVM internals, performance is easily affected by the lack of type annotations, something that is handled by the Julia runtime with dispatches based on dynamic types.

1) CPUs: For CPUs, Julia performed relatively well for the x86 architecture, as shown in Fig. 9. On EPYC, upon inspecting the generated native code via the `@code_native` macro,

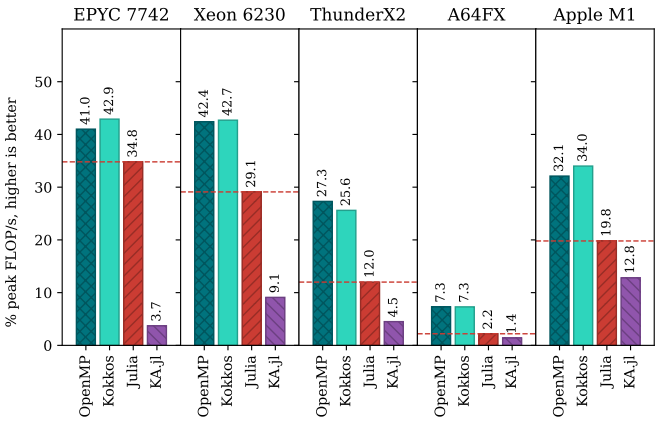

Fig. 9. miniBUDE CPU results

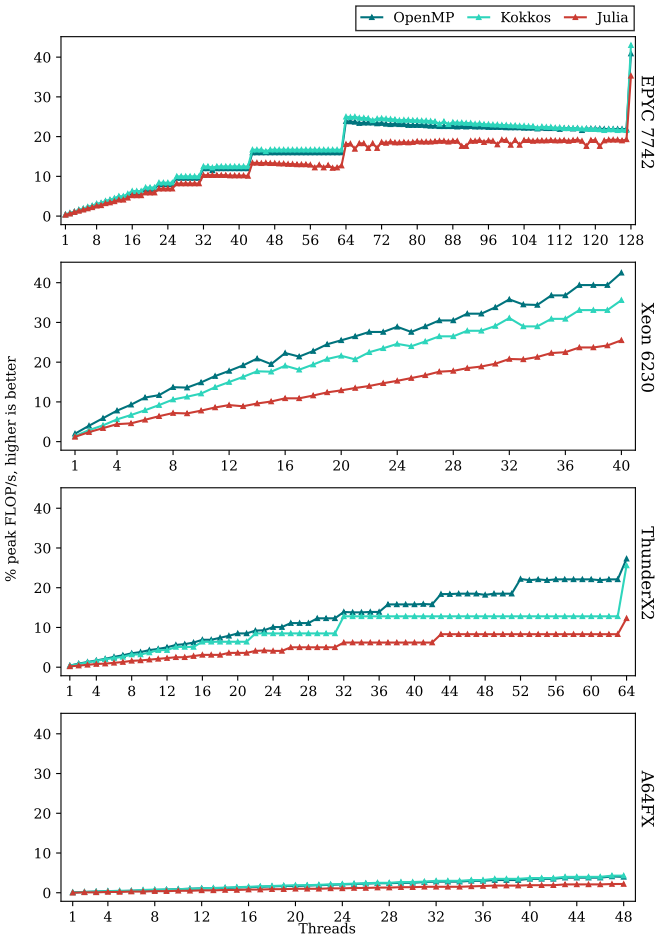

Fig. 10. miniBUDE CPU scaling results

Julia successfully emitted AVX2 instructions. Comparing this with LLVM's assembly, we see highly similar structure with only a handful of cases where Julia uses the packed version of the vector instructions. After further investigation, it is not clear why Julia showed a ~15% reduction in performance compared to the OpenMP implementation. For Xeon, we identified that Julia did not emit AVX512 instructions. On OpenMP, and by extension, Kokkos, both implementations successfully emitted AVX512 instructions when compiled using the correct set of optimisation flags. We were able to confirm that the lack of AVX512 contributed to the significantly lower performance (>30% difference) by replacing `-march=skylake-avx512 -mprefer-vector-width=512` with just `-march=skylake` on both the OpenMP and Kokkos implementation. With the non-AVX512 version of OpenMP and Kokkos, Julia showed nearly identical performance.

KA.jl on CPUs for miniBUDE achieved much lower performance though KA.jl mentioned in the documentation that execution on CPUs is not currently a priority. At the time of writing, KA.jl states that no specific optimisation has been done for CPUs beyond verification that the code produces the correct results. The inability for KA.jl to correctly compile JuliaStream.jl's Dot kernel due to the use of synchronisations as discussed in Section V-B further solidifies the untested state of running KA.jl kernels on CPUs.

For AArch64 platforms, both LLVM and Julia performed poorly on A64FX whereas Julia is significantly slower than LLVM on TX2 and M1. For A64FX and TX2, ArmClang's optimisation report and Julia assembly output showed no obvious issues, both implementations were emitting vectorised code. Testing with the non-Arm variant of LLVM and also Julia 1.7.0 also showed no major improvement. Investigating this further, we found that in A64FX's case, only Fujitsu's compiler was able to produce performance that is closer to the 40% theoretical FP32 performance obtained on x86 platforms. We understand that the coming LLVM 13 will support auto vectorisation with SVE on AArch64 platforms, and we expect that LLVM 13 will perform much better on A64FX, though this is yet to be validated.

For miniBUDE, there is a third level of for-loop after the cartesian iteration of proteins and ligands; this is done to facilitate vectorisation and not part of the algorithm shown in Algorithm 2. This structure is highly sensitive to missed optimisations as the effects are further amplified by the workgroup size. Given that both Julia and LLVM had some quirks on optimising BabelStream's Dot kernel on AArch64, we suspect this structure, along with the math operations in the miniBUDE kernel, contributed in making optimisation for miniBUDE even more challenging. This problem is also made worse on newer platforms such as A64FX. In general, we find that current compilers still produce less optimised code for AArch64 compared to the more ubiquitous x86 platform.

We present scaling results in Fig. 10 on CPU platforms, M1 is omitted for the same reason as discussed in Section V-B. As miniBUDE is compute bound, Julia scales similarly to Kokkos and OpenMP, demonstrating Julia's mature threading support.

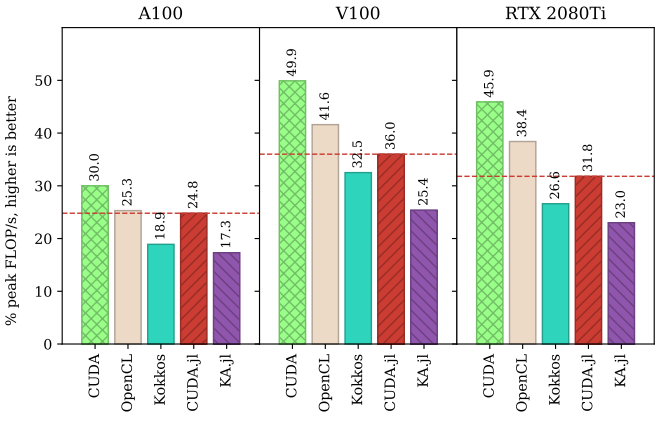

Fig. 11. miniBUDE Nvidia GPU results

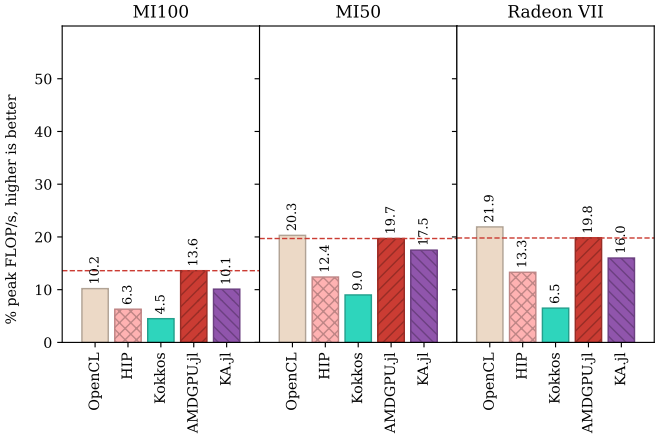

Fig. 12. miniBUDE AMD GPU results

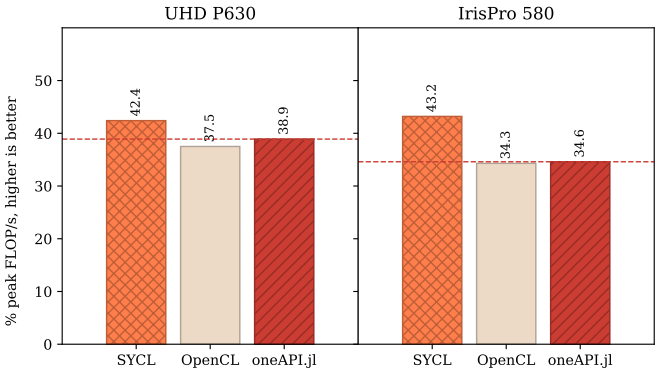

Fig. 13. miniBUDE Intel GPU results

2) GPUs: For GPUs, CUDA.jl is within ~30% of the first-party implementations, as shown in Fig. 11. In this case, Julia was able to produce code that achieved performance between Kokkos and OpenCL, two C/C++ derived third-party frameworks. Although slower than all other implementations, KA.jl is only slightly behind Kokkos.

Results for AMD GPUs are shown in Fig. 12 where AMDGPU.jl was able to outperform most other implementations, tying with OpenCL. This is surprising given that AMDGPU.jl fell short of first-party results from Section V-B.

We again attribute this to the immaturity of the ROCm platform as the best performing framework was only able to achieve 20% of the theoretical FP32 performance, in contrast to Intel and Nvidia where 40% ~50% is the norm. As expected, KA.jl followed closely behind AMDGPU.jl thanks to the close mapping.

Finally, Julia performed similarly to both SYCL and OpenCL on Intel GPUs, as shown in Fig. 13. This is particularly encouraging for the oneAPI.jl project as performance seems to be on-par with first-party implementations across memory-bandwidth bound (as shown in Fig. 8) and compute bound scenarios.

VI. Conclusion

We ported two mini-apps to Julia to show how it is effective at achieving high performance on a range of devices for both memory-bandwidth and compute bound applications.

For BabelStream, we observed nearly identical performance to the OpenMP and Kokkos versions on the CPU, with the exception of the Dot kernel on A64FX. As A64FX is a relatively new platform, even LLVM required extra compiler options for optimal performance. We were able to verify that with the latest beta release of Julia, along with a transformation of the reduction kernel to expose the dot product expression, Julia was able to match LLVM's performance. For GPUs, Julia's various GPU packages again performed very close (~15%) to first-party frameworks. On AMD platforms, Julia's AMDGPU.jl package is about 40% slower than the best performing framework, we attribute this to the ROCm stack's immaturity and AMDGPU.jl's beta status.

For miniBUDE, we get a better view on how well Julia handles floating point optimisations. In general, x86 CPU platforms performed well, although Julia was not able to emit AVX512 on platforms that support it. On AArch64, we observe difficulties for both LLVM and Julia to achieve a high percentage of the theoretical FP32 performance with Julia significantly slower than LLVM results. We believe compiler backends targeting AArch64 have yet to reach the same level of maturity for x86 platforms at the current stage. On GPUs, Julia performed similarly to what OpenCL is getting and it is usually less than 25% difference from the best performing framework. Which is to say, Julia is competitive for compute bound applications on GPUs.

Julia largely follows Python's *batteries included* motto when it comes to productivity. Many of the GPU packages even handle downloading software dependencies and configuring the host system for use with Julia's GPU support. For example, CUDA.jl retrieves the appropriate CUDA SDK from Nvidia on first launch.

In addition, because of Julia's JIT execution model, along with an ergonomic package system, creating a program that supports multiple accelerators from different vendors is straightforward. Traditionally, mixing frameworks that require different host compilers (e.g. `nvcc` for CUDA or `hipcc` for HIP) requires special attention to the overall project design to avoid compilation issues; programmers frequently have to resort to compiler-specific workarounds in the codebase and implement fragile and complex build scripts. In fact, the Kokkos framework was designed specifically to abstract this complexity away with highly sophisticated build scripts. Julia was able to avoid all this.

Currently, except KA.jl, Julia's kernel portability is tied to each of the GPU packages. In effect, writing optimised kernels for multiple vendors still requires a manual port. However, as each of the GPU packages share similar capabilities, the effort required is usually limited to basic API call substitutions. We look forward to seeing KA.jl support more platforms under the JuliaGPU umbrella.

Thanks to Julia's approach on reusing large parts of the LLVM project, Julia programs enjoys comparable performance to native C/C++ solutions. And thanks to the concentrated effort from the open-source communities on improving LLVM, Julia gets the unique opportunity to provide best-in-class performance on some of the latest hardware platforms. In general, we find Julia's language constructs map closely to the underlying LLVM Intermediate Representation under ideal conditions with precisely *ascribed* types; various conventional optimisation techniques and pitfalls in C/C++ still hold.

To this end, Julia offers us a glimpse of what is possible in terms of performance for a managed, dynamically-typed programming language. Given the overall performance-guided design of Julia, the LLVM-backed runtime, and comparable performance results shown here, we think Julia is a strong competitor in achieving a high level of performance portability for HPC use cases.

Acknowledgment

This work used Intel's DevCloud online cluster for developers (https://intelsoftwaresites.secure.force.com/devcloud/oneapi). This work used the Isambard UK National Tier-2 HPC Service (https://gw4.ac.uk/isambard) operated by GW4 and the UK Met Office, and funded by EPSRC (EP/P020224/1). This work used the HPC Zoo, a multi-platform research cluster managed by the High-Performance Computing Group at the University of Bristol (https://uob-hpc.github.io/zoo). This work used the DiRAC@Durham facility managed by the Institute for Computational Cosmology on behalf of the STFC DiRAC HPC Facility (www.dirac.ac.uk). The equipment was funded by BEIS capital funding via STFC capital grants ST/P002293/1,ST/R002371/1 and ST/S002502/1, Durham University and STFC operations grant ST/R000832/1. This work was also performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (www.csd3.cam.ac.uk), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/P020259/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk). DiRAC is part of the National e-Infrastructure.

References

[1] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 28.

[2] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A Fast Dynamic Language for Technical Computing," *CoRR*, vol. abs/1209.5145, 2012. [Online]. Available: http://arxiv.org/abs/1209.5145

[3] 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.

[4] T. Deakin, S. McIntosh-Smith, J. Price, A. Poenaru, P. Atkinson, C. Popa, and J. Salmon, "Performance portability across diverse computer architectures," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 2019, pp. 1–13.

[5] J. Regier, K. Fischer, K. Pamnany, A. Noack, J. Revels, M. Lam, S. Howard, R. Giordano, D. Schlegel, J. McAuliffe, R. Thomas, and Prabhat, "Cataloging the visible universe through Bayesian inference in Julia at petascale," *Journal of Parallel and Distributed Computing*, vol. 127, pp. 89–104, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731518304672

[6] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: Unleashing julia on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 4, pp. 827–841, 2019.

[7] A. Ramadhan, G. L. Wagner, C. Hill, J.-M. Campin, V. Churavy, T. Besard, A. Souza, A. Edelman, R. Ferrari, and J. Marshall, "Oceananigans.jl: Fast and friendly geophysical fluid dynamics on GPUs," *Journal of Open Source Software*, vol. 5, no. 53, p. 2018, 2020. [Online]. Available: https://doi.org/10.21105/joss.02018

[8] T. Clements and M. A. Denolle, "SeisNoise.jl: Ambient Seismic Noise Cross Correlation on the CPU and GPU in Julia," *Seismological Research Letters*, 09 2020. [Online]. Available: https://doi.org/10.1785/0220200192

[9] M. Innes, "Flux: Elegant Machine Learning with Julia," *Journal of Open Source Software*, 2018.

[10] N. R. Mamidi, K. Prasun, D. Saxena, A. Nemili, B. Sharma, and S. M. Deshpande, "On the performance of GPU accelerated q-LSKUM based meshfree solvers in Fortran, C++, Python, and Julia," 2021.

[11] S. Hunold and S. Steiner, "Benchmarking Julias Communication Performance: Is Julia HPC ready or Full HPC?" in *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020, pp. 20–25.

[12] JuliaLang, "Multi-Threading." [Online]. Available: https://docs.julialang.org/en/v1/base/multi-threading/#Base.Threads.@threads

[13] 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.

[14] 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.

[15] 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.

[16] JuliaGPU, "@synchronize inside while loops doesn't work on CPUs," https://github.com/JuliaGPU/KernelAbstractions.jl/issues/262, 2021.

APPENDIX

Artifact Description

We ran the BabelStream and miniBUDE mini-app on a wide range of hardware platforms listed in Table II.

Artifacts Available: Source code for JuliaStream.jl is currently in the process of being merged into BabelStream. The Pull Request, along with reviews from members of the Julia community is available at https://github.com/UoB-HPC/BabelStream/pull/106. Source code for miniBUDE.jl is now part of the miniBUDE benchmark, available at https://github.com/UoB-HPC/miniBUDE.

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/2021-benchmarking.

Experimental setup: See Table II for a list of hardware platforms used and Section V-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. Benchmark measurements uses the best result over 100 runs. For miniBUDE, the results are validated against the known values of the input deck. Benchmark measurements contains warm-up and measurements are derived over an 8 iteration average.

Validated the accuracy and precision of timings: For BabelStream, benchmark measurements use the best result over 100 runs. We also cross-check results with existing literature where possible. For miniBUDE, benchmark measurements contain warm-up and measurements are derived over an 8 iteration average. We also 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.

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