Julia: A competitive high-level choice for performance portability in HPC?

Jim M. R. Teichgräber

June 26, 2022

Abstract

Most attempts at achieving performance portability in high performance computing use low level languages such as C, C++ and Fortran. A number of performance portability approaches based on these languages have been developed, such as OpenCL™, SYCL™, Kokkos and OpenMP®. The Julia language aims to provide high-level language convenience, while delivering C-like performance, despite its managed runtime environment and dynamic type system.

We set out to compare Julia to existing performance portability approaches to determine if it is a competitive choice for HPC. We examine Julia's compilation process, syntax, idioms and package ecosystem to find what makes it appealing. We also study its performance in inter-node communication using the Message Passing Interface. Regarding intra-node workloads, we compare Julia's performance in memory-bandwidth bound as well as compute bound applications to that of the conventional approaches. We conclude that Julia can, in general, compete with the existing performance portable approaches to high performance computing.

1 Introduction

Dynamic programming languages such as Julia continue to be preferred for developing algorithms, solving problems and analyzing data [1]. Performing these tasks in lower-level languages such as C, C++ and Fortran requires more work and does not offer the same level of productivity [1]. Mathematical constructs map well to high-level dynamic languages, which often do not achieve the same performance as low-level languages. These, in turn, feature programming constructs which map well to the underlying hardware [2]. To achieve maximum performance in areas such as high performance computing, code developed in languages such as MATLAB®, R and Python has often been translated to highly performant lower-level languages [1]. Julia aims to provide a high-level dynamic language similar in nature to those mentioned above, but capable of achieving the performance levels of C or Fortran [1].

Julia uses a *just-in-time* (JIT) compilation model [3], which counteracts some of the overhead that a dynamic language with a managed runtime environment brings with it [4]. Julia also makes use of the *LLVM* compiler toolchain [5] for producing optimized machine code at runtime. It is also designed to enable easy translation of mathematical problems into code, as well as ease developer cooperation and code reuse through its approach to package management. JIT compilation is also related to performance portability in that its entire goal is to generate machine specific code at runtime, ideally independent of where it is run. The JIT paradigm in itself is nothing new and precedes the performance portability approaches that Julia will be compared to. This article thus also evaluates the viability of JIT compilation as an approach to performance portability, using the example of Julia.

To see if Julia's design decisions result in good performance in comparison to more conventional approaches to performance portable HPC programming, we will report on Julia's performance in serveral benchmarks, using results obtained in related work.

Julia's intra-node performance will be examined in the compute bound miniBUDE [6, 7] and memory-bandwidth bound (memory bound) BabelStream [8] benchmarks. The BabelStream benchmark is an extension of the well known STREAM benchmark [9]. Inter-node performance will be analyzed in terms of communication performance using the Julia MPI.jl wrapper [10] and the Open MPI C implementation [11] of the *Message Passing Interface* (MPI) standard [12, 13].

2 Background

2.1 Julia's Compilation Process

As mentioned before, Julia is compiled just-in-time, meaning that the compiler generates machine code at runtime of the program. Compilation happens in multiple stages. First, Julia source code is parsed into an *abstract syntax tree* (AST), then it is simplified ("lowered"), meaning higher level language features get converted into more verbose Julia code. Following this, the compiler infers types of expressions, wherever

possible. In code generation this lowered form of Julia code is converted into a portable *intermediate representation* (IR): LLVM [5] IR. LLVM then takes care of optimization in various passes, before translating the IR into platform specific machine code. [14, 4, 1]

Please note that this is a simplified view of the compilation process and that Julia's managed runtime environment can make a heuristics based decision to not JIT compile code at all and interpret it instead [14]. As this is not particularly relevant for HPC, since performance critical code should always be JIT compiled by the runtime environment, we will not look further into this.

As a language with a dynamic type system allowing arbitrary runtime type checks and dynamic multiple dispatch, Julia needs a managed runtime [15]. The managed runtime needs to take care of allocating sufficient memory for dynamically typed variables and selecting which branch of a multiple dispatch method is run based on the runtime type.

But with the use of type inference, the compiler can statically allocate stack memory at compile time[1], if a section of code is perfectly typed by inference [4, 1]. This reduces the need for the environment or the JIT compiler to interfere with program execution. Similarly, in cases where the types of method calls are known at compile time, the compiler can select an appropriate method, and thus avoid dynamic multiple dispatch at runtime. These effects become even more pronounced when considering, that the compiler is realistically able to perfectly type the majority of all expressions [1].

The managed runtime is therefore only necessary for dynamic code that needs recompilation or evaluation at runtime and specific language features such as garbage collection [4]. This means that the advantages of dynamic languages can be used in Julia code which is not performance critical. However, in the carefully written parts of the program that do require peak performance, it is possible to maintain the performance of statically compiled languages [1].

2.2 Idiomatic Julia

As a high level dynamic language, Julia allows for concise and expressive syntax, where lower-level languages like C need more explicit code [1]. It has become standard for code in high performance computing and technical computing to use a two-tiered architecture for writing code. Higher-level logic is expressed in a dynamic language such as R or Python and performance critical sections are written in C or Fortran [1]. Instead of programming in two different languages, Julia allows for high performance and convenience:

[1] This is specifically referring to the time in which the JIT compiler acts

1. The Julia runtime environment takes care of memory management. This not only improves programmer productivity, but also prevents memory related bugs which can often be hard to track down [2]. As explained in subsection 2.1, the JIT compiler can take care of efficiently allocating memory on the stack in many situations, while dynamic memory allocations at runtime are still possible if needed. [4]

2. Array bounds checking is done by default. This improves code safety as well as convenience for the programmer, though it can be circumvented with the use of the `@inbounds` macro to ensure maximum performance. [16]

3. Unicode symbols can be used to translate mathematical descriptions of algorithms more easily into code. This means constants or functions using greek letters such as `Δ` or `Φ` can be used in source code. LaTeX symbols can be used too and are often defined as aliases to existing functions or operators. For example `≥` can be used instead of `>=` and `∈` serves as an alias for the `in` keyword. Most Julia programming environments enable the use of the LaTeX `\` commands plus a `Tab` to type these special characters. [17]

4. Type inference allows for convenient and readable code, i.e. `Δ = 1.5` can be specified without any explicit type annotation, while not compromising on performance. [18]

5. Operators and functions can be defined on more complex types such as vectors and matrices. Adding two vectors a and b is simply expressed as `a+b`. Solving the system of linear equations $Ax = b$ for x can be done with `A\b` [18]. The resulting LLVM IR and assembly of such an operation can be inspected using the `@code_llvm` and `@code_native` macros. This reveals that Julia emits vectorized code when appropriate. On x86 derived architectures, instructions from the *Streaming SIMD Extensions* (SSE) and *Advanced Vector Extensions* (AVX) instruction set families are used.

6. The `@threads` and `@simd` macros allow for fine tuning of multi-threading and vectorization behavior. `@threads` in particular enables OpenMP-like parallelization of for loops:

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

```
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];
```

Figure 1: Comparison of Julia `@threads` (left) against C++ OpenMP parallelized for loop (right).
Taken from [2]

2.3 Package Ecosystem

In addition to Julia's syntax, its package ecosystem encourages developers to exchange and reuse code to easily build software. For HPC it is particularly interesting, as it encourages API uniformity: The *CUDA.jl*, *AMDGPU.jl*, *oneAPI.jl* and *KernelAbstractions.jl* (KA.jl) packages, all developed by the JuliaGPU group [19], provide functionality to program GPUs. They all share similar APIs for retrieving information about the environment that the compute-kernel is executed in as well as modifying said environment, as can be seen in Table I. This enables developers to adapt code to different GPU architectures with relative ease. [2]

In particular, CUDA.jl and its predecessor *CUDANative.jl* base their compilation processes Julia's own, without needing to change the Julia compiler itself [4]. CUDANative.jl, for instance, reuses large parts of the original Julia compiler and augments its functionality through *extension interfaces*, which hook into different parts of the compilation process to tailor it to generating GPU device code. For example, exceptions can be disallowed through the use of so-called `CodegenParam`-parameters, as they do not generally make sense in device code. [4] Because Julia uses LLVM to generate CPU code in the very last step of the compilation process, LLVM can also be used to convert the IR into machine code for GPU architectures instead. This means that there is no need to distribute a separate GPU compiler with the package. In the case of CUDANative.jl, the LLVM PTX backend [20] included with the Julia LLVM distribution, can generate well performing PTX code for CUDA devices from LLVM IR [4].

This makes it possible to distribute new accelerator device packages through the Julia package manager, independent of the Julia implementation that is used. In their work on CUDANative.jl, Besard et al. have also created the *LLVM.jl* package to allow for high-level interactions with LLVM, internally using Julia's *Foreign Function Interface* (FFI) to interface with the LLVM C API. [4]

As a result of this work, accelerator support for a particular architecture or device can be implemented compactly. CUDANative.jl itself only requires about 1500 lines of code. Thus, software portability is encouraged by design: firstly through the use of the LLVM compiler toolchain, abstracted away by the LLVM.jl package and secondly by integrating into the main Julia compiler. [4]

There are also additional benefits to using this heavily interconnected compiler architecture for device code. CUDA and host calls to a function are only distinguished by the use of the `@cuda` macro, which also means that dynamic code can be written and efficiently executed on the GPU. Because there is no differentiation between device code and host code, much of the Julia standard library can be used in writing device code and device code can be reused on the host.

2.4 BabelStream and miniBUDE

In order to measure intra-node performance, the variations on the STREAM and miniBUDE benchmarks will be used.

A. BabelStream: The STREAM benchmark [9] consists of 4 operations which are meant to emulate typical vector operations in HPC codes. The specific benchmark that will be used is the BabelStream benchmark [8] which additionally implements the dot product to benchmark a reduction operation [2].
The benchmark is best described by the pseudocode in Algorithm 1.

Algorithm 1 BabelStream kernels, taken from [2]

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

The pseudocode highlights the memory bound nature of the benchmark, the computations themselves are trivial, the number of array accesses is much more significant.

B. miniBUDE: The Bristol University Docking Engine (BUDE) [7] is a highly compute bound molecular dynamics based application. For benchmarking compute bound applications, the miniBUDE [6, 2] benchmark, which is derived from the full scale BUDE application will be used. This benchmark requires trigonometric function evaluations, square roots and absolute values over multiple iterations [2].

3 Performance Results

To measure the performance of Julia in comparison to other performance portability frameworks and approaches, we will report on Julia's intra-node performance in compute bound as well as memory bound

TABLE I
JULIA GPU KERNEL API CROSS-REFERENCE

	CUDA.jl	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)`

Taken from [2]

applications. In doing so, we will compare CPU and GPU performance, to determine if Julia is able to perform well in highly heterogeneous systems. On the CPU side, we will compare the standard Julia compiler and the KA.jl package against OpenMP and Kokkos. As for GPUs, depending on the architecture, CUDA or HIP, CUDA.jl or AMDGPU.jl and OpenCL, Kokkos and KA.jl will be compared. Specifically, the results of Lin et al. [2] will be presented[1].

Please note that the results that Lin et al. obtained on consumer CPUs and GPUs will not be considered, as they are not relevant for HPC. This also fully excludes the results comparing SYCL, OpenCL and oneAPI.jl on Intel Integrated Graphics GPUs.

We will analyze MPI performance as a measure for communication performance in highly distributed inter-node workloads. More specifically, we will compare the MPI.jl package [10] to the Open MPI [11] C library, using the results of Hunold et al. [21] who have analyzed their performance in a subset of the *ReproMPI* [22, 23] benchmark suite[1]. It should be noted that, while Open MPI is an implementation of the MPI Standard, MPIl.jl is merely a wrapper, which can use a number of different MPI implementations, including Open MPI. In this case, MPI.jl using Open MPI will be compared against the direct use of Open MPI functions.

3.1 Intra-Node Performance

The results of the compute bound miniBUDE benchmark will be examined first. Afterwards, we will analyze performance in the memory bound BabelStream benchmark, before continuing on to the communication performance of inter-node applications.

3.1.1 Compute bound — miniBUDE

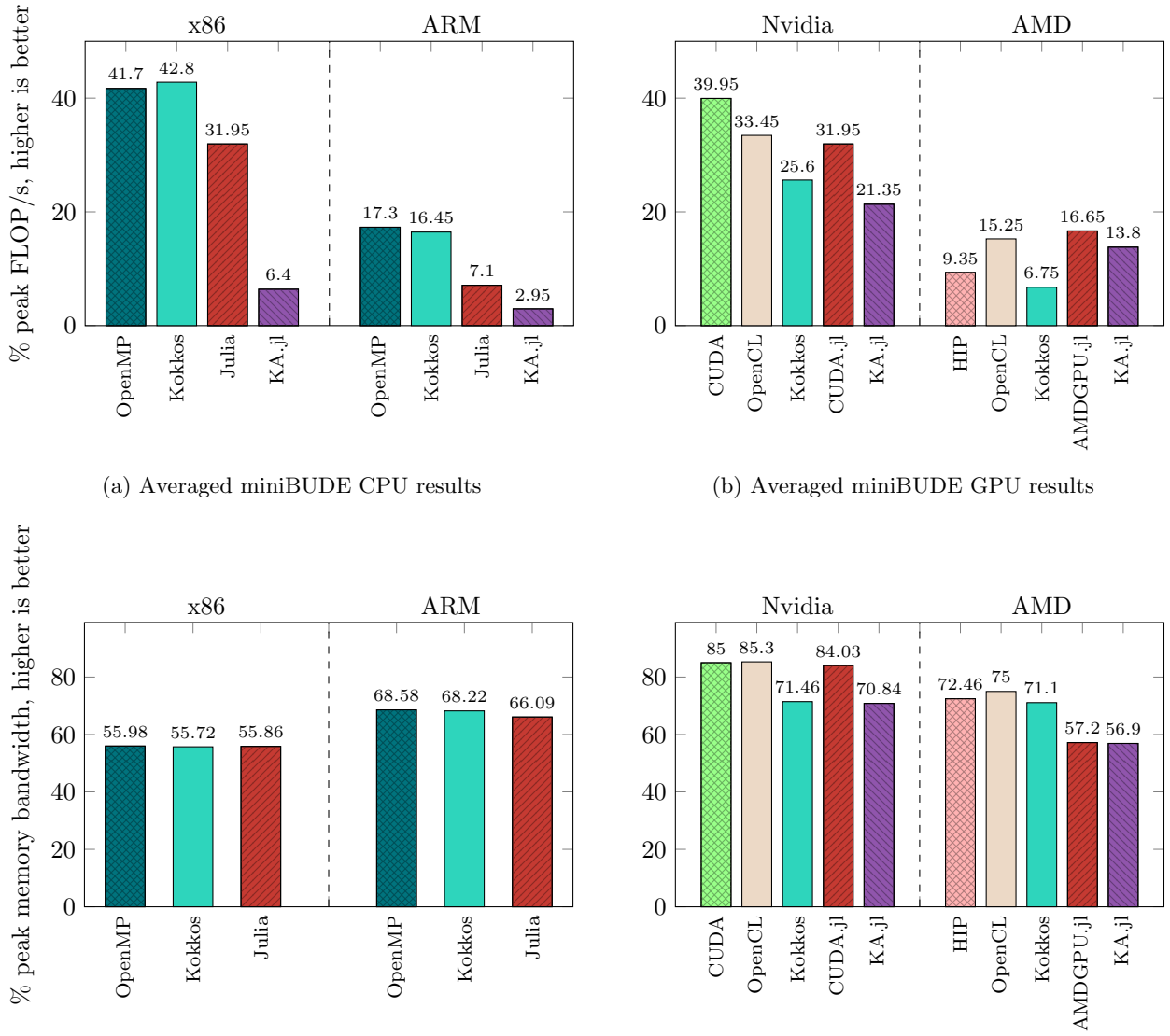
A. CPU: On the x86 architectures Julia was often able to produce vectorized code using the AVX2 instruction set, although it was not yet capable of emitting AVX512 instructions. This resulted in a performance drop compared to OpenMP and Kokkos on the AVX512 capable architectures and thus a drop in the average performance, as can be seen in Figure 2a. If AVX512 instructions were manually forbidden for OpenMP and Kokkos, Julia closed the gap, resulting in very similar performance. The exception to this was the AMD EPYC platform, where Julia performed 15% worse than both OpenMP and Kokkos. As there was almost no difference between the LLVM IR emitted by Julia in comparison to that emitted by OpenMP, this disparity could not be explained. [2]

On ARM, on the other hand, Julia performed notably worse than both OpenMP and Kokkos, losing out by about 59%. It is noteworthy that both OpenMP and Kokkos only reached about 7-25% of the peak FLOP/s possible on this architecture, which is far lower than the approximately 40% achieved on the x86 architectures. Julia attained about 2-12% on ARM, while reaching about 30% to 35% on x86. [2]

KA.jl performed 5 to 10 times worse than all the other approaches in all scenarios, which is to be expected, as the documentation states that execution on CPUs is currently not a priority [2].

Julia scaled similarly to the OpenMP and Kokkos

[1]A description of the exact benchmark environment and setup can be found in the respective paper



(a) Averaged miniBUDE CPU results

(b) Averaged miniBUDE GPU results

(c) Averaged BabelStream CPU results

(d) Averaged BabelStream GPU results

Figure 2: Performance results, all plots adapted from [2]. All results are averages over all platform specific results. The BabelStream results are also averaged over the performance of the five different kernels seen in Algorithm 1.

implementations, the performance gap expanding linearly with the number of cores [2]. This highlights multithreading support for Julia.

B. GPU: On the Nvidia architectures (see Figure 2b), the native CUDA implementation of miniBUDE is ahead of the portable approaches by about 5% to 10% of the theoretical peak FLOP/s, 20% to 25% in relative terms. CUDA.jl performs slightly better than Kokkos and slightly worse than OpenCL. Depending on the architecture, CUDA.jl matches OpenCL in some scenarios, while staying behind by about 14% in others. KA.jl performed just behind Kokkos. [2]

AMDGPU.jl mostly outperformed OpenCL, HIP and Kokkos on the AMD architectures. OpenCL can still compete with AMDGPU.jl, only lagging behind slightly. HIP and Kokkos are about 50% to 70% slower than AMDGPU.jl. KA.jl also performs well, matching OpenCL and beating out HIP and Kokkos. [2]

3.1.2 Memory bound — BabelStream

A. CPU: Due to a bug in KA.jl, it was excluded from the comparison [2].

When using all available CPU cores, the performance of OpenMP, Kokkos and Julia was identical as can be seen in Figure 2c. Julia did however scale worse than the other two frameworks. The authors attribute this to Julia's inherent thread affinity [24] behavior: If OpenMP and Kokkos use the `OMP_PROC_BIND=close` pragma, they scale very simi-

larly to Julia, while `OMP_PROC_BIND=spread` yields the best scaling results. [2]

B. GPU: On the Nvidia architectures, CUDA.jl reaches the performance levels of the native CUDA implementation. OpenCL does so too, while both Kokkos and KA.jl lag behind slightly (see Figure 2d). [2]

In the case of the AMD architectures, KA.jl and AMDGPU.jl yield identical results, which suggests that the mapping from KA.jl to AMDGPU.jl is very close. Both packages perform about 20% worse than OpenCL, HIP and Kokkos, in contrast to the results of the compute bound benchmark. The Julia packages perform about 50% worse in the Dot kernel benchmark, which stands out as it is the only kernel in BabelStream performing a reduction operation. The Dot kernel results are not explicitly shown in Figure 2d, as they are integrated into the average results.

The authors attribute this overall suboptimal performance to the immaturity of the AMDGPU.jl package, which is still in the early stages of development. [2]

3.2 Inter-Node Performance

Julia version 1.4.0 (MPI.jl 0.14.3) showed nearly identical performance to Open MPI (C) in almost all scenarios. The only exception to this was the case of the `MPI_Allreduce` [13] method, which showed significant slowdown in MPI.jl for message sizes above 10kB. Performance was about 1.5 to 2 times slower than in the case of Open MPI, depending on the exact message size. Previous versions of Julia and the MPI.jl package performed worse, taking between 2 and 2.5 times longer than C. [21]

The `MPI_Allreduce` method is a non-rooted collective operation, in which all processes in all nodes of the cluster perform an operation on their local data and contribute their result to the global result [21].

Upon analyzing the distribution of the running times of MPI.jl and Open MPI over multiple runs of the `MPI_Allreduce` benchmark, the authors found highly variant running time distributions for MPI.jl for larger message sizes, whereas the direct use of Open MPI in C was fairly consistent in its running times across message sizes, as shown in Figure 3. The root cause of this inconsistency could not be explained. [21]

4 Discussion

4.1 Performance

In general, Julia's best packages and implementations do not lag far behind the other performance portable approaches in intra-node performance. In most memory bound applications there is no performance loss due to abstractions provided by Julia. Compute bound applications are still dominated by first party solutions such as CUDA, but Julia can generally compete with other performance portable solutions. It does lag behind slightly in CPU performance in these applications, due to the compiler not emitting AVX512 instructions.

Except for the anomaly in the `MPI_Allreduce` case discussed in subsection 3.2 Julia's MPI performance was on par with that of the direct use of Open MPI, making it competitive in inter-node communication use cases.

4.2 Lines of code

In general one can assume to gain a significant reduction in the amount of lines of code (LOC) when using Julia compared to lower-level languages and frameworks. This is backed up by the work of Besard et al. [4] who implemented CUDANative.jl, the predecessor to the CUDA.jl package. In their benchmarks they compared native CUDA C code to equivalent kernel and host code written in Julia and found an overall reduction in lines of code of about 32%. Still using low level semantics, they achieved an average reduction of 8% in GPU device code. The lines of host code were reduced by about 38%, as it was possible to use more high level features there, without compromising on performance in device code, as described in subsection 2.1. The exact reductions in LOC can be examined in Figure 4.

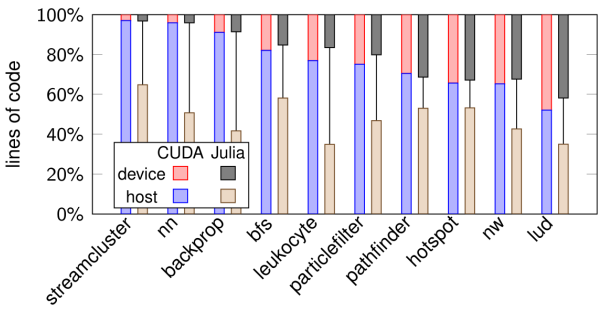
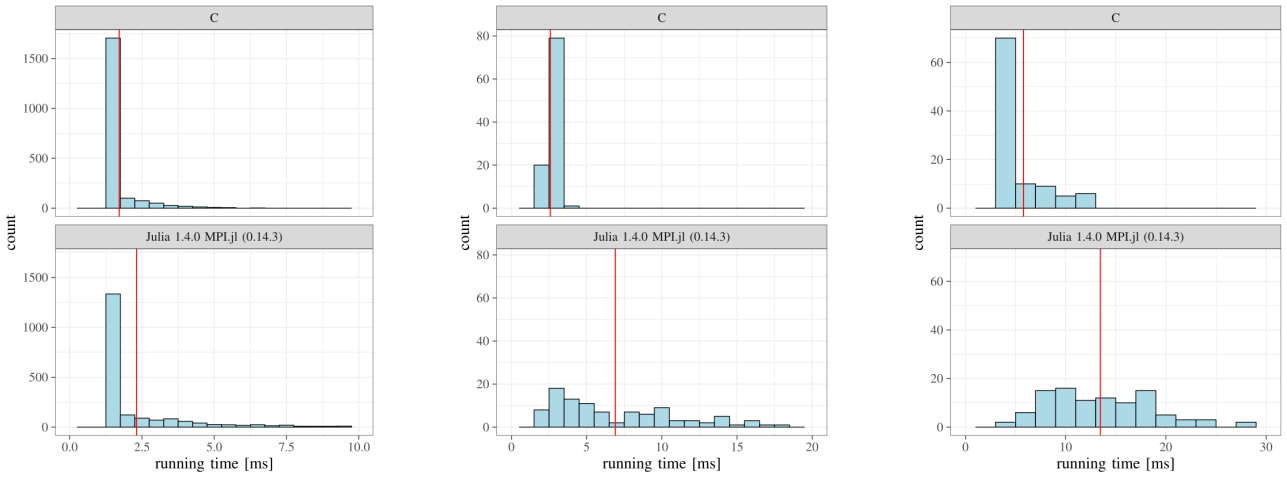


Figure 4: Lines of code reduction CUDA C vs CUDANative.jl Julia package in various benchmarks, taken from [4]

The authors also noted that programming device code in a low-level style of Julia was more comfortable compared to CUDA C, due to various Julia language features such as dynamic types and checked arithmetic [4].

4.3 Single source portability

Of the Julia packages that were benchmarked, only KA.jl can provide true single source portability. KA.jl is also the slowest package throughout and cannot compete in CPU or GPU applications, whether compute bound or memory bound. Porting implementations in GPU architecture specific packages such as



(a) 10 240 B (b) 102 400 B (c) 1 024 000 B

Figure 3: Histogram of running times over multiple runs of the `MPI_Allreduce` method, broken down by message size, mean values are marked in red, taken from [21]

CUDA.jl and AMDGPU.jl usually only requires API call substitutions using a table like Table I, because the Julia package ecosystem encourages a uniform approach to accelerator programming, as described in subsection 2.3. Thus, performance portable programming for HPC is possible in Julia, but still requires some manual effort. [2]

5 Conclusion

We compared Julia's performance to that of a number of other performance portability approaches, including OpenCL, Kokkos, OpenMP and HIP in a range of applications. We evaluated performance in the compute bound miniBUDE benchmark [6, 7] and the memory bound BabelStream benchmark [8], as well as communication performance using the Julia MPI.jl wrapper and the Open MPI C implementations of the MPI standard [12, 13]. Furthermore, we detailed how Julia's design and syntax, compilation process and package management make it appealing for writing mathematical and technical code, and enable convenience while maintaining comparable performance.

We conclude that Julia can generally keep up with the performance of other performance portability frameworks. Its portability is still confined to the specific GPU package used, but porting Julia kernel code to different architectures is simplified by the API uniformity of the different GPU packages. As these integrate tightly with the Julia compilation process and LLVM, it can be presumed that portability to future architectures will be possible in the same way it is now.

Julia thus offers a convenient, high-level language, designed for performance and being able to deliver on that promise. High performance computing in Julia is possible with good performance across architectures and nodes, but its portability can still be improved.

References

[1] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman. Julia: a fast dynamic language for technical computing. *arXiv preprint*, 2012. DOI: 10.48550/ARXIV.1209.5145.

[2] W.-C. Lin and S. McIntosh-Smith. Comparing Julia to Performance Portable Parallel Programming Models for HPC. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 94–105, 2021. DOI: 10.1109/PMBS54543.2021.00016.

[3] J. Aycock. A Brief History of Just-in-Time. *ACM Comput. Surv.*, 35(2):97–113, June 2003. ISSN: 0360-0300. DOI: 10.1145/857076.857077.

[4] T. Besard, C. Foket, and B. De Sutter. Effective extensible programming: unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2018. ISSN: 1045-9219. DOI: 10.1109/TPDS.2018.2872064. arXiv: 1712.03112 [cs.PL].

[5] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* Pages 75–86, 2004. DOI: 10.1109/CGO.2004.1281665.

[6] A. Poenaru, W.-C. Lin, and S. McIntosh-Smith. A Performance Analysis of Modern Parallel Programming Models Using a Compute-Bound Application. In *36th International Conference, ISC High Performance 2021*, Frankfurt, Germany, 2021.

[7] S. McIntosh-Smith, J. Price, R. B. Sessions, and A. A. Ibarra. High performance in silico virtual drug screening on many-core processors. *The International Journal of High Performance Computing Applications*, 29(2):119–134, 2015. DOI: 10.1177/1094342014528252. PMID: 25972727.

[8] 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*, 17(3):247–262, Special issue, 2018. DOI: 10.1504/IJCSE.2018.095847.

[9] 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*, 2(19-25), 1995.

[10] MPI.jl. URL: https://juliaparallel.org/MPI.jl/stable/.

[11] Open MPI. URL: https://www.open-mpi.org/.

[12] The MPI Forum. URL: https://www.mpi-forum.org/ (visited on June 5, 2022).

[13] M. Snir, W. Gropp, S. Otto, S. Huss-Lederman, J. Dongarra, and D. Walker. *MPI – the Complete Reference: the MPI core*, volume 1. MIT press, 1998, pages 178–180.

[14] *Julia Developer Documentation on Eval of Julia Code*. URL: https://docs.julialang.org/en/v1/devdocs/eval/ (visited on May 20, 2022).

[15] *Julia Manual on Types*. URL: https://docs.julialang.org/en/v1/manual/types/#man-types (visited on May 18, 2022).

[16] *Julia Developer Documentation on Bounds checking*. URL: https://docs.julialang.org/en/v1/devdocs/boundscheck/ (visited on May 26, 2022).

[17] *Julia Manual on Unicode input*. URL: https://docs.julialang.org/en/v1/manual/unicode-input/ (visited on May 26, 2022).

[18] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: a fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[19] Julia GPU Group. URL: https://juliagpu.org/ (visited on May 26, 2022).

[20] User guide for the LLVM NVPTX Back-end. URL: https://www.llvm.org/docs/NVPTXUsage.html (visited on June 4, 2022).

[21] S. Hunold and S. Steiner. Benchmarking Julia's Communication Performance: Is Julia HPC ready or Full HPC? In *2020 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 20–25, 2020. DOI: 10.1109/PMBS51919.2020.00008.

[22] ReproMPI Benchmark Suite. URL: https://github.com/hunsa/reprompi (visited on May 26, 2022).

[23] ReproMPI Benchmark Suite, Julia port. URL: https://github.com/sebastian-steiner/reproMPI.jl (visited on May 26, 2022).

[24] Wikipedia contributors. Processor affinity — Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Processor_affinity&oldid=1027552885 (visited on June 4, 2022).