

Productivity meets Performance: Julia on A64FX

^{1st} Mosè Giordano

*Advanced Research Computing
UCL*

London, United Kingdom
m.giordano@ucl.ac.uk

^{2nd} Milan Klöwer

*Atmospheric, Oceanic and Planetary Physics
University of Oxford*

Oxford, United Kingdom
milan.kloewer@physics.ox.ac.uk

^{3rd} Valentin Churavy

*CSAIL, EECS
Massachusetts Institute of Technology*
Cambridge, United States of America
vchuravy@mit.edu

Abstract—The Fujitsu A64FX ARM-based processor is used in supercomputers such as Fugaku in Japan and Isambard 2 in the UK and provides an interesting combination of hardware features such as Scalable Vector Extension (SVE), and native support for reduced-precision floating-point arithmetic. The goal of this paper is to explore performance of the Julia programming language on the A64FX processor, with a particular focus on reduced precision. Here, we present a performance study on `axpy` to verify the compilation pipeline, demonstrating that Julia can match the performance of tuned libraries. Additionally, we investigate Message Passing Interface (MPI) scalability and throughput analysis on Fugaku showing next to no significant overheads of Julia of its MPI interface. To explore the usability of Julia to target various floating-point precisions, we present results of `ShallowWaters.jl`, a shallow water model that can be executed at various levels of precision. Even for such complex applications, Julia’s type-flexible programming paradigm offers both, productivity and performance.

Index Terms—Julia, A64FX, BLAS, MPI, floating-point numbers, reduced precision

I. INTRODUCTION

The Julia programming language [1] is a dynamic programming language, with a focus on productivity and performance. It has found increased adoption in numerical and scientific computing, data processing and analytics, differentiable programming and scientific machine learning. Julia uses LLVM [2] as a compiler backend and supports multiple CPU architectures (x86, PPC, ARM) which includes Fujitsu’s A64FX that powers the Fugaku supercomputer. Based at the RIKEN Center for Computational Science (R-CCS) in Kobe, Japan, Fugaku is currently number 2 in the TOP500 ranking of the fastest supercomputers in the world. It has topped the list from its induction in June 2020 to June 2022 [3]. It is composed of 158 976 Fujitsu A64FX FX1000 CPUs, making it the first ARM supercomputer to claim the highest position in TOP500. Inter-node communication is powered by Tofu Interconnect D (TofuD), a proprietary technology developed by Fujitsu [4].

While 16-bit arithmetic are increasingly supported on modern hardware, 64-bit double-precision floating-point numbers (here called `Float64`, other formats likewise, referring respectively to the IEEE-754 standard formats [5]) are still widely used in scientific computing as they largely remove the necessity of a detailed numerical analysis of rounding errors in most applications. Smaller rounding errors and lower risk of under and overflows come at a cost of computational performance as low-precision arithmetic is executed significantly faster on

supporting hardware. Also `Float32` is widely supported on various CPU architectures, and many GPUs support different 16-bit formats (`Float16`, `BFloat16`), but A64FX is the first modern CPU for high-performance computing (HPC) that supports `Float16` arithmetic. Many mantissa bits in high-precision formats contain little to no information in applications with large uncertainties, such as deep learning [6], [7] or climates models [8]. In these examples, a higher performance from low-precision arithmetic could be exchanged for larger networks or higher resolution to increase the complexity of these models. In theory, A64FX promises $4\times$ performance increase of `Float16` over `Float64` in both memory and compute-bound applications through its 512-bit SVE vectorization unit [9].

This article is an experience report with Julia on Fujitsu A64FX, a processor architecture primarily used by supercomputers [10], [11]. We focus on the performance of low-precision float arithmetic and evaluate the overheads of using Julia for MPI programs on Fugaku. It follows a section on `ShallowWaters.jl`, a shallow water model, that makes use of Julia’s type-flexibility to be executable at various levels of precision.

II. SUPPORT FOR LOW-PRECISION FLOATING-POINT ARITHMETIC IN JULIA

Julia uses multiple dispatch which as a programming paradigm makes it easier to support new number formats. In the code snippet below we reproduce the type hierarchy of floating-point numbers, starting with the abstract type `Number` and ending at the primitive type `Float16`.

```
abstract type Number end
abstract type Real <: Number end
abstract type AbstractFloat <: Number end
primitive type Float64 <: AbstractFloat 64 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float16 <: AbstractFloat 16 end
```

This type-hierarchy comes into play when we look at the implementation of math routines, like the cube root function `cbrt`.

```
julia> methods(cbrt)
# 7 methods for generic function "cbrt":
[1] cbrt(x::Union{Float32, Float64}) in Base.Math at
special/cbrt.jl:142
```

```
[2] cbrt(a::Float16) in Base.Math at special/cbrt.jl:150
[3] cbrt(x::BigFloat) in Base.MPFR at mpfr.jl:626
[4] cbrt(x::AbstractFloat) in Base.Math at special/cbrt.jl:34
[5] cbrt(x::Real) in Base.Math at math.jl:1352
```

Julia provides for `cbrt` several implementations that range from the specialized to the generic. `Float32` and `Float64` share an implementation and `Float16` is separated. Julia then dynamically dispatches to the most specific method available for a given type at runtime. This allows Julia to both provide general implementations and fast implementations that take advantage of the structure of the types.

Early versions of Julia supported `Float16` only as a storage format. Mathematical functions immediately promoted to `Float32` and no rounding was performed. This changed in Julia v0.6¹, and since then operations converted their output back to `Float16`. Yet this was fully done in software and no hardware support (even if available) was used. Since Julia v1.6² the compiler lowers the `Float16` type to LLVM’s half type.

The default behavior of LLVM on x86 chips was to extend the precision of half operations to float, which is unsuitable for numerical implementations that need to return consistent results on software and hardware implementations. GCC recognized this problem as well in version 12 [12]: “The default behavior for `FLT_EVAL_METHOD` is to keep the intermediate result of the operation as 32-bit precision. This may lead to inconsistent behavior between software emulation and AVX512-FP16 instructions”. In Julia we generally require that operations are numerically stable across different hardware platforms, and thus for software `Float16` we insert rounding operations. On hardware that support `Float16` natively we could use LLVM to directly lower to hardware instructions. There is ongoing work detailed in section IV-C to improve compiler support for detecting hardware that supports `Float16`. For the experiments in section III-B we explicitly turn on this support³.

III. TYPE-FLEXIBILITY AND PERFORMANCE

A. Performance and Scalability on Fugaku

We run benchmarks of Julia code on Fugaku to evaluate the efficiency of the code generated by LLVM for simple Julia functions, and the overhead of using the `MPI.jl` package for communications on a world-class supercomputer. The results of all these benchmarks are publicly available at <https://github.com/giordano/julia-on-fugaku>, including the job scripts used to run the benchmarks and the code to produce the plots reported in the present section, along with the Julia package environments used, for full reproducibility.

1) *Level 1 BLAS routine*: The Basic Linear Algebra Subprograms (BLAS) is a prescription of low-level routines for performing common linear algebra operations, such as matrix and vector operations [13]. BLAS routines are classified in

three levels: Level 1 includes vector and vector operations, Level 2 includes vector-matrix operations, Level 3 includes matrix-matrix operations. There are several high-performance implementations of BLAS routines, with hardware vendors often providing highly optimised BLAS implementations for their own systems. One of the most common Level 1 routines is `axpy` which represents the mathematical operations of multiplying a vector x by a scalar a , adding it to a vector y and storing the result back into y (“ a times x plus y ”):

$$y \leftarrow ax + y. \quad (1)$$

We implemented in Julia a generic single-threaded `axpy` function, which can take in input vectors of any arbitrary numerical Julia type:

```
function axpy!(a::T, x::Vector{T}, y::Vector{T}) where
    {T<:Number}
    @simd for i in eachindex(x, y)
        @inbounds y[i] = muladd(a, x[i], y[i])
    end
    return y
end
```

The `@simd` macro suggests the compiler to enable Single Instruction Multiple Data (SIMD) instructions, and the macro `@inbounds` informs the compiler it is safe to skip bounds checks, as we automatically iterate over the existing indices of the vectors x and y with the `eachindex` function. In section II we note that Julia is currently supposed to widen `Float16` numbers to `Float32` numbers, but due to a bug in the LLVM pass⁴ vectors of `Float16` are not widened to vectors of `Float32`, thus this implementation of `axpy` retains hardware performance for 16-bit floating-point numbers as desired.

We compare performance of the above generic Julia function, using Julia v1.7.2 (based on LLVM 12) having set the environment variable `JULIA_LLVM_ARGS` to `-aarch64-sve-vector-bits-min=512`, with the following binary libraries:

- vendor’s Fujitsu BLAS, from module `lang/tcsds-1.2.35` on Fugaku, library called `libfjlapackexsve_ilp64.so`, with ILP64 and support for SVE instructions,
- BLIS version 0.9.0,
- OpenBLAS version 0.3.20, built with the Spack package manager version 0.19, using the specification `openblas@0.3.20 %gcc@8.5.0 +ilp64 symbol_suffix=64_` (building OpenBLAS with the Fujitsu compiler resulted in multiple compilation errors⁵),
- ARM Performance Libraries (ARMPL) version 22.0.2 for RHEL 8 with GCC 8.2, library called `libarmpl_ilp64.so`, single-threaded, with ILP64.

¹<https://github.com/JuliaLang/julia/pull/17297>.

²<https://github.com/JuliaLang/julia/pull/37510>.

³<https://github.com/JuliaLang/julia/issues/40216>.

⁴<https://github.com/JuliaLang/julia/issues/45881>.

⁵See <https://github.com/xianyi/OpenBLAS/issues/3692> and <https://github.com/spack/spack/issues/31675>.

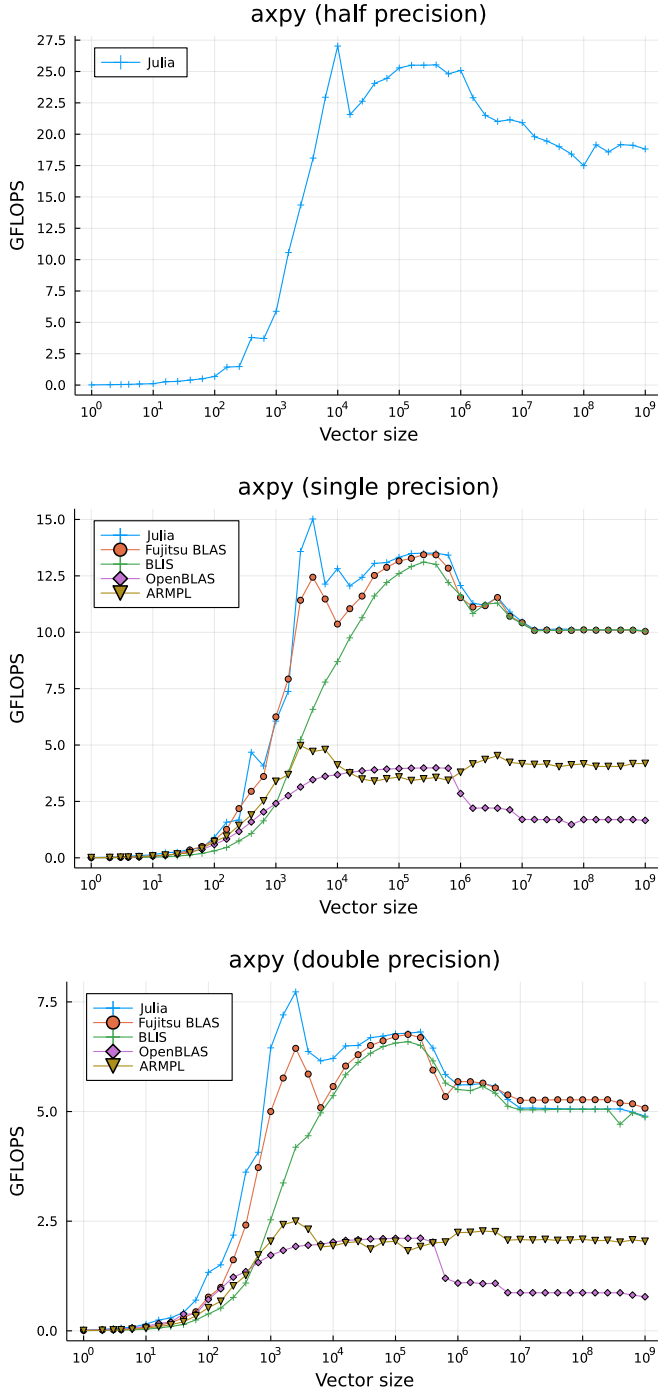


Fig. 1. Performance comparison of `axpy` implementations in Julia versus Fujitsu BLAS, BLIS, OpenBLAS, and ARMPL on Fugaku, using half precision (top panel, Float16), single precision (middle, Float32) and double precision (bottom, Float64). A single thread is used in all benchmarks. The vector size refers to the length of the vectors x and y in eq. (1). GFLOPS are gigaFLOPS, the number floating-point operations per second executed by each program. The panel for half precision shows performance only for Julia, because half-precision implementations of `axpy` are not available for the other binary libraries (Fujitsu BLAS, BLIS, OpenBLAS, ARMPL).

For these benchmarks we use `libblastrampoline`⁶, a library which uses Procedure Linkage Table (PLT) trampolines to forward BLAS calls to a chosen library (e.g., Fujitsu BLAS or BLIS) at runtime with near-zero overhead compared to the complexity of the routines invoked, without having to recompile an application to link to a different BLAS library. Figure 1 shows the results of these benchmarks. Both OpenBLAS and ARMPL show poor performance for this routine, likely because it is not taking full advantage of A64FX vectorization capabilities. We note that there are no implementations of `axpy` for half-precision floating-point numbers in Fujitsu BLAS, BLIS, OpenBLAS, and ARMPL, whereas Julia is able to generate code for the type-generic function `axpy!` with half-precision Float16 numbers. Also, Julia’s implementation consistently outperforms BLIS, OpenBLAS and ARMPL implementations of `axpy` in both single and double precision, it is competitive with Fujitsu BLAS across all sizes, and it achieves the best peak performance in all cases.

Thanks to contribution of ARM engineers, auto-vectorization in LLVM 14 is able to target SVE/SVE2 by default when available [14]. Preliminary benchmarks performed with the development version of Julia v1.9, based on LLVM 14.0.2, showed similar results to those reported in Figure 1 and the generated LLVM IR for the `axpy!` function uses `llvm.vscale` intrinsics, but without having to set the environment variable `JULIA_LLVM_ARGS` to `-aarch64-sve-vector-bits-min=512`, which improves the ability of writing high-performance code in Julia.

2) *Distributed computing with MPI.jl*: MPI is a communication protocol for distributed programs which run on multiple cores and is a staple in the HPC field: it is the de-facto standard for communication in highly parallel applications. `MPI.jl` is a Julia package to interface with this protocol [15]. Studies about MPI communications with `MPI.jl` on x86 architecture were conducted by [16], [17], who found relatively little overhead on AMD and Intel CPUs.

R-CCS presented at the *7th meeting for application code tuning on A64FX computer systems* the results of performance benchmarks of MPI communication on Fugaku [18]. These benchmarks were run with the *Intel MPI Benchmarks*⁷ (IMB) suite with the Fujitsu MPI library, based on Open MPI and optimised for TofuD. In order to measure performance of MPI communications on A64FX using `MPI.jl`, we developed `MPIBenchmarks.jl`⁸, a package to run Julia benchmarks comparable to some of those in the IMB suite.

Figure 2 shows the results of the benchmarks for a point-to-point operation (ping-pong), and Figure 3 refers to collective operations (Allreduce, Gather, Reduce). We ran the benchmarks with Julia v1.7.2, using `MPI.jl` v0.20, which was configured to use the Fujitsu MPI library available in the system. Point-to-point benchmarks were run with two MPI ranks on two nodes, collective benchmarks were run with 1536

⁶<https://github.com/JuliaLinearAlgebra/libblastrampoline>.

⁷<https://github.com/intel/mpi-benchmarks>.

⁸<https://github.com/JuliaParallel/MPIBenchmarks.jl>.

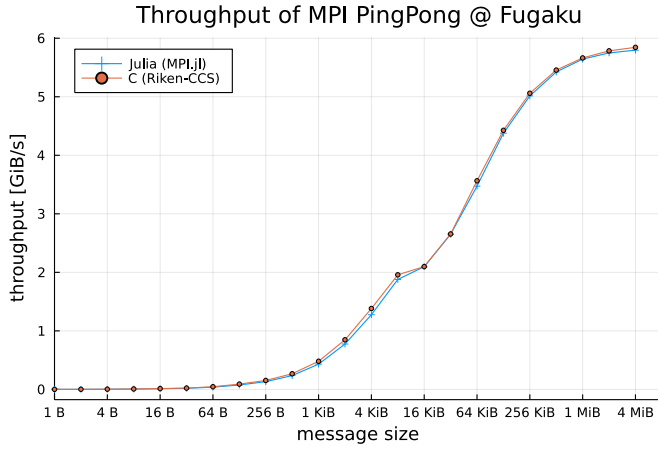
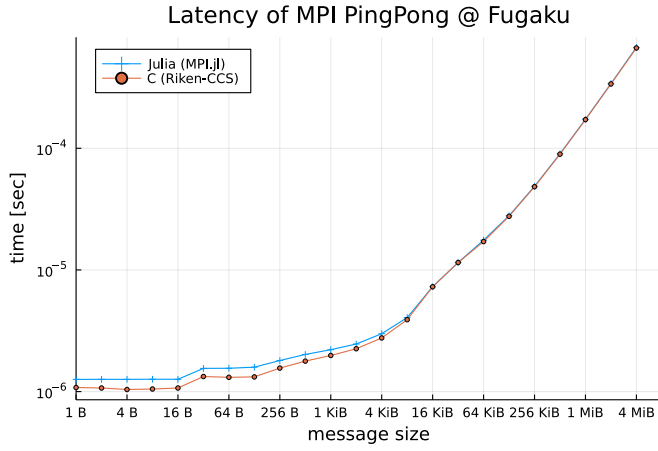


Fig. 2. Comparison of latency (top panel) and throughput (bottom panel) of inter-node point-to-point MPI communication between using `MPI.jl` in Julia and IMB benchmarks in C (results provided by R-CCS in [18]). Fugaku scheduler setup: `-L "node=2" -mpi "max-proc-per-node=1"`.

MPI ranks across 384 nodes using the torus layout, to match the scheduler configuration of the R-CCS benchmarks. `MPI.jl` typically showed very small overhead for messages larger than 1-2 KiB—peak throughput of ping-pong communication with `MPI.jl` is within 1% of that reported by R-CCS—but, slightly larger overhead for messages of smaller sizes. We note that, contrary to IMB, at the present time `MPIBenchmarks.jl` does not implement a cache-avoidance mechanism, which may explain why `MPI.jl` appears to show better latency than IMB for messages with size up to 64 KiB, which corresponds to the size of the L1 cache of the A64FX CPU. We also observe that, contrary to [16], we did not find a significant performance drop for the `Allreduce` operation for larger message sizes.

B. Type flexibility and reduced-precision with `Float16`

Developing complex applications using `Float16` is not easy. On A64FX, even the occasional occurrence of subnormals of `Float16` ($6 \cdot 10^{-8}$ to $6 \cdot 10^{-5}$) causes a heavy performance penalty but a compiler-flag is set to flush them to zero instead⁹. The available normal range of `Float16`, $6 \cdot 10^{-5}$ to 65,504,

⁹<https://github.com/JuliaLang/julia/issues/40151>.

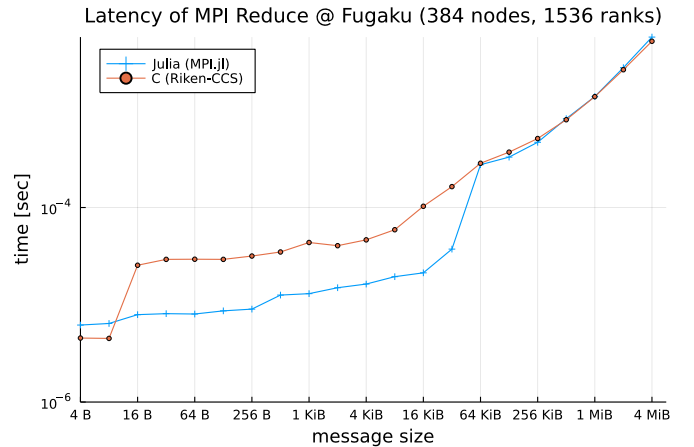
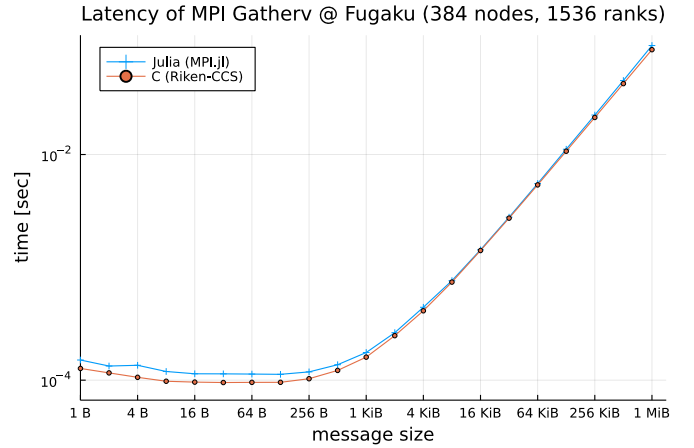
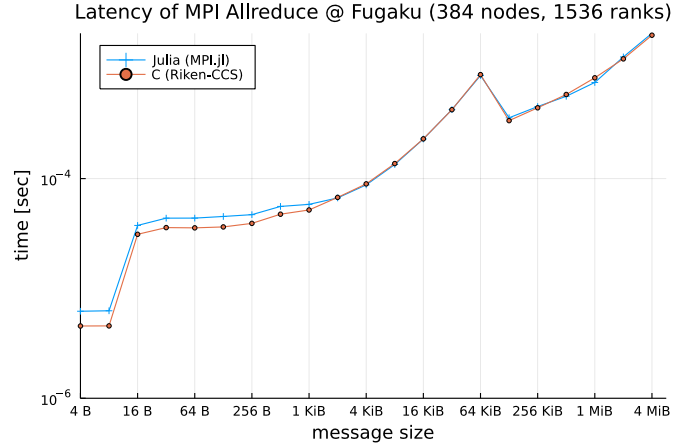


Fig. 3. Comparison of latency of collective MPI operations between using `MPI.jl` in Julia and IMB benchmarks in C (results provided by R-CCS in [18]): MPI `Allreduce` (top panel), MPI `Gatherv` (middle panel), MPI `Reduce` (bottom panel). Fugaku scheduler setup: `-L "node=4x6x16:torus:strict-io" -L "rscgrp=small-torus" -mpi proc=1536`.

is less than 10 orders of magnitude and scaling is often required to guarantee no under or overflow. While developing an application that is resilient to the limitations of `Float16` it is therefore beneficial to retain compatibility to higher-precision formats. In practice, many complex applications will have parts that are performance-critical, other parts that are precision or range-critical. An approach is therefore needed that combines a flexibility with the number format for development and performance when ported to various hardware without sacrificing productivity. Many existing libraries hardcode the number format and deliver performance by essentially duplicating conceptually identical code, which harms productivity and portability to number formats available on modern hardware. On the other hand, Julia’s multiple-dispatch allows the development of fully type-flexible applications, such that the number format, or combinations of different formats, can be chosen at compile time as described in the previous section. In the following we present one application that prototypes this concept for weather and climate models and runs in `Float16` on A64FX or in `Float64` on x86 without changes.

`ShallowWaters.jl`, a fluid circulation model that solves the shallow water equations for idealized weather and climate simulations, has been developed with a focus on number format-flexibility [19]. It runs in `Float64`, `Float32`, or `Float16` alike and in general supports any (custom) number format as long as a standard set of arithmetic operations are implemented. Functions are written for element types `T<:AbstractFloat` and Julia dynamically dispatches an arithmetic operations like addition to `+(x::T, y::T)`, i.e. the respective method defined for the number format `T`. While this method can be defined in Julia’s Base library (as is the case for floats) any custom number format can be defined by implementing a standard set of arithmetic operations. What this set of operations needs to contain depends on the application. `ShallowWaters.jl`, for example, only uses transcendental functions like `log`, `exp`, `sin`, `cos` etc. for precalculating constants and not in the performance-critical main loop. To optimize the range of numbers occurring `ShallowWaters.jl`, we developed the analysis-number format `Sherlogs.jl`, which records a histogram of numbers during the simulation that allowed us to monitor, for example, how a multiplicative scaling s of the equations avoids `Float16` subnormals. For development purposes we therefore run `ShallowWaters.jl` with `T=Sherlog32` (`Sherlogs.jl`’s equivalent of `Float32`), and, after choosing s , we execute the same code with `T=Float16, s=s` for performance. For more details see [19]. Type-flexibility therefore is not just important for performance code, but can also greatly assist in the development of low precision-resilient code and retains compatibility with higher precision formats.

The `ShallowWaters.jl` simulations with `Float16` are qualitatively indistinguishable from simulations with `Float64` (Figure 4) and rounding errors remain smaller than model or discretization errors. The precision-critical part is the time integration for which we include a compensated summation that compensates for the rounding error of the previous time

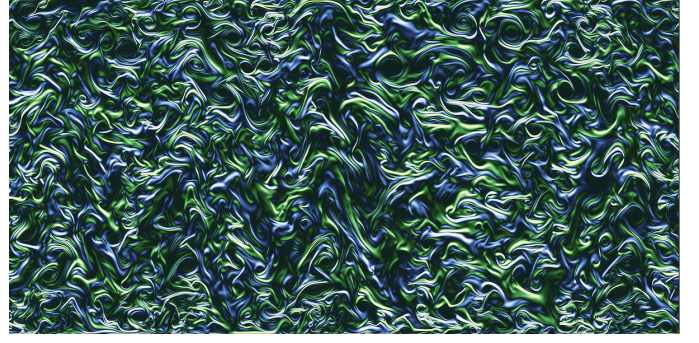


Fig. 4. Geophysical turbulence simulated by `ShallowWaters.jl` using `Float16` arithmetic on A64FX and a spatial resolution of 3000×1500 grid points. `ShallowWaters.jl` uses an identical code base that is dynamically dispatched to any number format. To reduce rounding errors in the precision-critical time integration, `Float16` simulations includes a compensated summation, which is not required for higher-precision formats. The equivalent `Float64` simulation is qualitatively indistinguishable but ran $3.6\times$ slower [19].

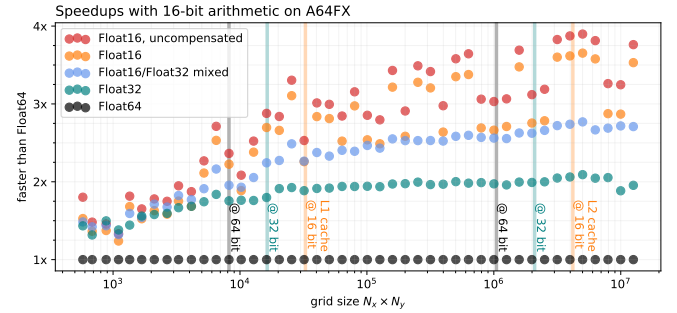


Fig. 5. Speedups of low-precision simulations on A64FX with `ShallowWaters.jl` over `Float64` with varying problem sizes. `Float16` has by default a compensated time integration in `ShallowWaters.jl`, to reduce rounding errors, which causes an about 5% overhead in runtime. `Float16/32` is a mixed-precision simulation that uses `Float32` precision for the time integration. Reproduced from [19].

step by adding a correction to the next time step. This introduces a 5% overhead in runtime and therefore clearly outperforms a mixed-precision approach whereby the precision-critical time integration is computed using `Float32` (Figure 5). As `ShallowWaters.jl` is a memory-bound application it benefits from `Float16` on A64FX even without vectorization and approaches $4\times$ speedups over `Float64` for large problems (3000×1500 grid points, corresponding approximately to array sizes). `Float32` simulations are $2\times$ faster than `Float64` over a much wider range of problem sizes.

IV. OPPORTUNITIES FOR IMPROVEMENTS

A. General performance

An evaluation of performance portability of Julia code across multiple architectures, including A64FX, was carried out by [20], who showed that Julia could achieve on this platform performance close to that of equivalent code written in C/C++. The authors of this study noted that the performance improved sensibly when moving from Julia v1.6, which is based on

LLVM 11, to Julia v1.7, based on LLVM 12. Analogously, we found that the performance of a simple Julia implementation of the BLAS `axpy` routine can be competitive with that of a vendor implementation of BLAS (Fujitsu BLAS), or another highly optimised library (BLIS). Furthermore we observed that thanks to recent advancements in LLVM, Julia v1.9 will be able to more easily vectorize the code, without requiring users to manually set LLVM flags. Thus, owing to the fact it relies on a compiler infrastructure which receives active support from hardware vendors and HPC engineers, Julia can enjoy improvements versions after versions also on very specialised CPUs, despite the fact that to date Julia itself has not received directly any specific optimisation for A64FX.

A64FX is a non-general-purpose CPU, with strong focus on vectorization. This results in poor performance in some tasks, such as compilation of software. On Fugaku this issue can be limited by cross-compiling static software optimised for A64FX on the Intel login nodes. However, Julia is Just-In-Time-compiled (JIT), thus paying the cost of longer compile times in every session whenever a new method needs to be compiled. Julia currently does not support cross-compilation of code for a different architecture, but there are tools to enable basic ahead-of-time compilation, to generate a system image to reduce the need to compile methods at runtime¹⁰. Improvements in this area of the Julia ecosystem can enhance the ability to run large-scale applications on A64FX and other similar non-general-purpose CPUs.

B. Custom reduction operators in `MPI.jl`

An issue that is limiting the ability to run some MPI applications on ARM CPUs is the impossibility to use custom MPI reduction operations on non-Intel architectures due to how they are implemented in `MPI.jl`¹¹. When this bug will be resolved, it will be possible to run a larger class of Julia MPI programs on ARM systems, including Fugaku.

C. Improved compiler support

As noted in section II compilers need be careful in how they handle `Float16` on platforms that have only software support for it. If they allow for extending precision intermediately, this can cause issues with code as simple as:

```
muladd(x, y, z) = x*y+z
```

which Julia lowers to the following LLVM Intermediate Representation (IR):

```
define half @julia_muladd(half %0, half %1, half %2) {
top:
  %3 = fmul half %0, %1
  %4 = fadd half %3, %2
  ret half %4
}
```

¹⁰<https://github.com/JuliaLang/PackageCompiler.jl>.

¹¹<https://github.com/JuliaParallel/MPI.jl/issues/404>.

In order to ensure the consistency between hardware and software, Julia inserts `fpext` and `fptrunc` operations explicitly:

```
define half @julia_muladd(half %0, half %1, half %2){
top:
  %3 = fpext half %0 to float
  %4 = fpext half %1 to float
  %5 = fmul float %3, %4
  %6 = fptrunc float %5 to half
  %7 = fpext half %6 to float
  %8 = fpext half %2 to float
  %9 = fadd float %7, %8
  %10 = fptrunc float %9 to half
  ret half %10
}
```

On systems with full hardware support this is clearly sub-optimal and there is ongoing work in LLVM and Julia to address this issue. In particular we will need to extend Julia's multi-versioning support to detect full `Float16` hardware support and then selecting a copy of the cache code that was compiled without inserting these extra conversion operations.

V. CONCLUSIONS

In this paper we evaluated the use of the Julia programming language on A64FX, in particular with regards to the ability to easily generate efficient code on a non-general purpose CPU; the possibility of writing type-generic code which can take advantage of hardware acceleration; and the overhead of running distributed applications with `MPI.jl` on a large supercomputer.

With the example of a simple Level 1 BLAS routine, we found that, by leveraging the work done in the LLVM compiler, Julia allows users to write high-level generic code which is compiled down to native code for A64FX with performance competitive with that of specialised libraries. Recent improvements in LLVM will further enhance the ability to automatically vectorize the code and fully use SVE instructions, particularly important with the introduction of more ARM CPUs supporting this extension, such as Neoverse V1 and N2.

The `MPI.jl` package provides a natural interface to the MPI protocol, which allows calling directly MPI libraries optimised for the network of the current system. Communication benchmarks of `MPI.jl` showed relatively little overhead compared to the performance analysis carried out by R-CCS, especially for sufficiently large messages, in line with the findings of [16], [17]: peak throughput of point-to-point communications was nearly identical to that measured using the IMB suite in C.

We then presented the case of the `ShallowWaters.jl` package, a fully type-flexible fluid circulation solver for the shallow water equations which, thanks to Julia's multiple dispatch and code generation, can run on different architectures (e.g., x86 and ARM) and with different types (e.g., `Float16` and `Float64`) by using different data types as input to the program, without changing the code base for the different situations. This showed that Julia is particularly well suited for developing generic numerical code which can effortlessly use

different numerical data types without sacrificing performance, boosting scientific productivity. We were also able to verify that running simulations with `Float16` on A64FX delivers a nearly 4x speedup over `Float64`.

However, some challenges in the use of Julia on A64FX remain. Prior to Julia v1.9, the now in-development version which will be based on LLVM 14, SVE instructions are enabled only when manually setting the LLVM flag `-aarch64-sve-vector-bits-min=512`, which however also causes crashes in many situations¹². `Float16` numbers are promoted to `Float32` numbers even when hardware support for 16-bit numbers is available, although it is possible to compile Julia with this mechanism disabled (as done here in section III-B). Compilation latency on A64FX can sometimes hinder runtime performance of Julia, more than on general-purpose CPUs, in particular in short-running tasks. `MPI.jl` does not currently support custom MPI reduction operators on ARM CPUs, which poses a limitation on running certain distributed application in Julia on these systems. While more work is needed to fully unlock Julia’s potential on A64FX-based clusters—and a large part of it is on-going both in LLVM and Julia itself and its ecosystem, not limited to this CPU—we found that this is already an effective language for writing generic and high-performance code for A64FX.

ACKNOWLEDGMENT

The authors thank Tim Besard, Jameson Nash and Simon Byrne for their inputs and work on both Julia and `MPI.jl`. This work used computational resources of the supercomputer Fugaku provided by RIKEN through the HPCI System Research Project (Project ID: ra000019). The ShallowWaters.jl simulations were run on the Isambard UK National Tier-2 HPC Service operated by GW4 and the UK Met Office, and funded by the Engineering and Physical Sciences Research Council EPSRC (EP/P020224/1). MK gratefully acknowledges funding from the European Research Council under the European Union’s Horizon 2020 research an innovation programme (grant no. 741112). VC gratefully acknowledges funding from NSF (grants OAC-1835443, OAC-2103804, AGS-1835860, and AGS-1835881), DARPA under agreement number HR0011-20-9-0016 (PaPPa). This research was made possible by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program, by the Paul G. Allen Family Foundation, Charles Trimble, Audi Environmental Foundation. This material is based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003965. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copy-right notation herein.

¹²See for example <https://github.com/JuliaLang/julia/issues/43069>, <https://github.com/JuliaLang/julia/issues/44263>, <https://github.com/JuliaLang/julia/issues/44401>.

REFERENCES

- [1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, “Julia: A fresh approach to numerical computing,” *SIAM Review*, vol. 59, no. 1, pp. 65–98, Jan. 2017.
- [2] 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.* IEEE, 2004.
- [3] J. Dongarra, “Report on the Fujitsu Fugaku system,” Department of Electrical Engineering and Computer Science, Innovative Computing Laboratory, University of Tennessee, Tech. Rep. ICL-UT-20-06, Jun. 2020. [Online]. Available: <https://www.top500.org/news/report-fujitsu-fugaku-system-jack-dongarra/>
- [4] Y. Ajima, T. Kawashima, T. Okamoto, N. Shida, K. Hirai, T. Shimizu, S. Hiramoto, Y. Ikeda, T. Yoshikawa, K. Uchida, and T. Inoue, “The tofu interconnect d,” in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, 2018, pp. 646–654.
- [5] “IEEE standard for Floating-Point arithmetic,” IEEE, Piscataway, NJ, USA, Tech. Rep., 2019.
- [6] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, “Training Deep Neural Networks with 8-bit Floating Point Numbers,” *arXiv:1812.08011 [cs, stat]*, Dec. 2018.
- [7] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, “A Study of BFLOAT16 for Deep Learning Training,” *arXiv:1905.12322 [cs, stat]*, Jun. 2019.
- [8] M. Klöwer, M. Razingier, J. J. Dominguez, P. D. Düben, and T. N. Palmer, “Compressing Atmospheric Data into Its Real Information Content,” *Nature Computational Science*, vol. 1, no. 11, pp. 713–724, Nov. 2021.
- [9] FUJITSU, “FUJITSU Processor A64FX Datasheet,” 2020. [Online]. Available: https://www.fujitsu.com/downloads/SUPER/a64fx/a64fx_datasheet.pdf
- [10] T. Odajima, Y. Kodama, M. Tsuji, M. Matsuda, Y. Maruyama, and M. Sato, “Preliminary Performance Evaluation of the Fujitsu A64FX Using HPC Applications,” in *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2020, pp. 523–530.
- [11] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu, “Co-Design for A64FX Manycore Processor and “Fugaku”,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov. 2020, pp. 1–15.
- [12] “Half-Precision (using the GNU compiler collection (GCC)),” <https://gcc.gnu.org/onlinedocs/gcc/Half-Precision.html>, accessed: 2022-6-30.
- [13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, “Basic Linear Algebra Subprograms for Fortran Usage,” *ACM Trans. Math. Softw.*, vol. 5, no. 3, p. 308–323, Sep. 1979. [Online]. Available: <https://doi.org/10.1145/355841.355847>
- [14] W. Lovett, (2022) LLVM 14 - what’s new and improved for Arm. [Online]. Available: <https://community.arm.com/arm-community-blogs/b/tools-software-ides-blog/posts/llvm-14>
- [15] S. Byrne, L. C. Wilcox, and V. Churavy, “MPI.jl: Julia bindings for the Message Passing Interface,” *Proceedings of the JuliaCon Conferences*, vol. 1, no. 1, p. 68, 2021. [Online]. Available: <https://doi.org/10.21105/jcon.00068>
- [16] 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)*, 2020, pp. 20–25.
- [17] A. Rizvi and K. C. Hale, “A Look at Communication-Intensive Performance in Julia,” *arXiv e-prints*, p. arXiv:2109.14072, Sep. 2021.
- [18] Y. Nakamura, “Basic Performance of Fujitsu MPI on Fugaku,” Jan. 2022. [Online]. Available: https://www.hpci-office.jp/invoke2/documents2/meeting_A64FX_220127/A64FX-Tuning_2022-0127_Nakamura.pdf
- [19] M. Klöwer, S. Hatfield, M. Croci, P. D. Düben, and T. N. Palmer, “Fluid Simulations Accelerated With 16 Bits: Approaching 4x Speedup on A64FX by Squeezing ShallowWaters.jl Into Float16,” *Journal of Advances in Modeling Earth Systems*, vol. 14, no. 2, p. e2021MS002684, 2022.
- [20] W. Lin and S. McIntosh-Smith, “Comparing Julia to Performance Portable Parallel Programming Models for HPC,” in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2021, pp. 94–105.