

JACC v1.0: High-productivity Performance Portability in the LLVM-compiled Julia Language

Philip W Fackler

Oak Ridge National Laboratory

Pedro Valero-Lara

Oak Ridge National Laboratory

Narasinga Rao Miniskar

Oak Ridge National Laboratory

Steven Hahn

Oak Ridge National Laboratory

Het Mankad

Oak Ridge National Laboratory

Doris Chiang

Oak Ridge National Laboratory

Jose Gonzalez

Oak Ridge National Laboratory

Jhonny Gonzalez

Oak Ridge National Laboratory

Alexis Huante

Oak Ridge National Laboratory

Yuguo K Tang

Oak Ridge National Laboratory

Emily Yang

Oak Ridge National Laboratory

Claire A Winogradski

Oak Ridge National Laboratory

Ritesh Bhirud

University of Massachusetts Amherst

Julian Samaroo

Massachusetts Institute of Technology

Rabab Alomairy

Massachusetts Institute of Technology

Alan Edelman

Massachusetts Institute of Technology

Johannes Blaschke

Lawrence Berkeley National Laboratory

Michel Schanen

Argonne National Laboratory

Patrick Diehl

Los Alamos National Laboratory

Elijah Pelofske

Los Alamos National Laboratory

Jean-François Boussuge

Centre Européen de Recherche et de Formation Avancée en Calcul Scientifique (CERFACS)

Ludovic Räss

University of Lausanne

Simone Marras

New Jersey Institute of Technology

Yuki Nagai

University of Tokyo

Jeffrey S. Vetter

Oak Ridge National Laboratory

Keita Teranishi

Oak Ridge National Laboratory

William F. Godoy

godoywf@ornl.gov

Oak Ridge National Laboratory

Research Article

Keywords: Julia, High-performance computing, High productivity, CPU and GPU, performance portability

Posted Date: October 22nd, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10856841/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

JACC v1.0: High-productivity Performance Portability in the LLVM-compiled Julia Language

Philip Fackler^{1†}, Pedro Valero-Lara^{1†}, Narasinga Rao Miniskar¹,
Steven Hahn¹, Het Mankad¹, Doris Chiang¹, Jose Gonzalez¹,
Jhonny Gonzalez¹, Alexis Huante¹, Yuguo K Tang¹,
Emily Yang¹, Claire A Winogrodzki¹, Ritesh Bhirud^{1,10},
Julian Samaroo², Rabab Alomairy^{2,11}, Alan Edelman²,
Johannes Blaschke³, Michel Schanen⁴, Patrick Diehl⁵,
Elijah Pelofske⁵, Jean-François Boussuge⁶, Ludovic Räss⁷,
Simone Marras⁸, Yuki Nagai⁹, Jeffrey S. Vetter¹,
Keita Teranishi¹, William F. Godoy^{1*†}

^{1*}Oak Ridge National Laboratory, Oak Ridge, TN, USA.

²Massachusetts Institute of Technology, Cambridge, MA, USA.

³Lawrence Berkeley National Laboratory, Berkeley, CA, USA.

⁴Argonne National Laboratory, Lemont, IL, USA.

⁵Los Alamos National Laboratory, Los Alamos, NM, USA.

⁶Centre Européen de Recherche et de Formation Avancée en Calcul
Scientifique (CERFACS), Toulouse, France.

⁷University of Lausanne, Lausanne, Switzerland.

⁸New Jersey Institute of Technology, Newark, NJ, USA.

⁹University of Tokyo, Tokyo, Japan.

¹⁰University of Massachusetts Amherst, Amherst, MA, USA.

¹¹King Abdullah University of Science and Technology, Thuwal,
Western Province, Saudi Arabia.

*Corresponding author(s). E-mail(s): godoywf@ornl.gov;

†These authors contributed equally to this work.

Abstract

This paper presents Julia for ACCelerators (JACC) version 1.0, a framework that provides users of the LLVM-compiled Julia language with easy access

to CPU/GPU parallelization. JACC leverages Julia’s unique features and the JuliaGPU ecosystem, enabling users to write a single portable code using APIs for `array` allocation and for launching `parallel_for` and `parallel_reduce` kernels on CPUs and GPUs from major GPU vendors (NVIDIA, AMD, Intel, and Apple). Here, we provide an overview of JACC’s first stable release, its architecture and current stable functionality and support, interoperability with the Julia high-performance computing (HPC) ecosystem, advanced features (e.g., shared memory, multi-GPU programming, and asynchronous kernel execution), and notable science use cases in fluid dynamics, multiphysics and quantum solvers, and neutron science facilities. JACC closes gaps in the Julia ecosystem by providing a single unified framework that improves the productivity, portability, and performance of scientific application development. It thus advances the programmability and accessibility of HPC systems.

Keywords: Julia, High-performance computing, High productivity, CPU and GPU, performance portability

1 Motivation and Significance

Programming high-performance computing (HPC) systems is a complex endeavor, and it is expected to become even more so with the proliferation of heterogeneous hardware (e.g., a growing number of industry-driven AI accelerators) [1]. Despite the wide adoption of LLVM [2] by compiler vendors, a plethora of vendor-specific, standard, and third-party programming models are used, and they target mainly C++ , C, and Fortran as the *de facto* HPC languages. Every generation of computing has introduced new productive high-level programming paradigms to abstract out common low-level aspects so that programmers can focus more on domain-specific problems. Notable examples include Fortran [3], DARPA languages [4], and the wide adoption of dynamic, high-level memory-managed languages such as Python, R, and MATLAB for non-performance-critical code for their increased productivity and support.

The past decade saw large investments in accelerators toward exascale computing [5]. GPUs in particular have driven the evolution of performance-portable “*write once, run everywhere*” models, as in standard implementations (e.g., OpenMP [6], OpenACC [7], OpenCL [8], SYCL [9, 10]) and third-party implementations (e.g., Kokkos [11], RAJA [12], Alpaka [13])—implementations that are critical for exascale applications [14, 15]. These libraries ameliorate the large refactoring costs of porting to different architectures [16]. Though traditionally funded by government investment, vendor-neutral GPU portability has recently attracted interest from language developers in industry, including those working on Chapel [17], Julia [18], and the more recent multi-level intermediate representation (MLIR)–based Mojo [19] language.

In this paper, we present the first stable version of the Julia for ACCelerators (JACC) [20] library. JACC enables users of the Julia language to write portable kernels once, and these kernels can be deployed and run on different CPU and GPU architectures (NVIDIA, AMD, Intel, and Apple). JACC leverages Julia’s metaprogramming,

mathematical syntax, LLVM compilation, and ecosystem capabilities to provide simple `array`, `parallel_for`, and `parallel_reduce` constructs. This feature increases programmability by using a high-level language, as most alternatives target the C++ language (e.g., Kokkos, RAJA, SYCL, TBB). Given the growth in adoption of the Julia language in scientific computing, JACC helps close gaps between high productivity and high performance by providing accessible parallel computing for Julia users without specialized knowledge of accelerator programming. Moreover, JACC provides sufficient control for more advanced users to tune their kernel performance.

The principal contributions of this paper are as follows:

1. We present a **software** description of the APIs and architecture of JACC version 1.0, the first stable release, which leverages Julia’s ecosystem, including the JuliaGPU¹ backends.
2. We demonstrate the **impact** of Julia and JACC in scientific HPC applications.
3. We provide **new directions** for applications that can benefit from JACC, including quantum computing and novel runtimes, and the use of agentic AI.

The rest of the paper is organized as follows. Section 2 provides background information about Julia and the motivation for JACC. Section 3 describes JACC’s design principles, architecture and backend relationships, public APIs, and available functionality. Section 4 illustrates the impact of JACC for scientific users who have adopted the Julia language, and Section 5 discusses new science directions in which JACC can play a key role. Finally, Section 6 outlines our conclusions. JACC aims to provide a balance between high productivity and high performance by leveraging the LLVM-based Julia language and ecosystem. The stable JACC API gives Julia users immediate access to CPU and GPUs from major vendors (NVIDIA, AMD, Intel, and Apple) for their parallel computing needs. Thus, JACC v1.0 provides a stable development path that removes the need for vendor-specific code and flattens the steep learning curve on accelerator programming, and it provides enough low-level, hardware-specific mechanisms for performance tuning.

2 Background: Julia and HPC

In this section we discuss the recent adoption and use of the Julia language for scientific computing and HPC.

2.1 The Julia Programming Language

Julia offers a combination of unique features covering three aspects that support productive technical computing:

- Rich mathematical (e.g., native multi-dimensional arrays) and AI frameworks for scientific code and data analysis.
- Native performance via a type system, multiple dispatch, garbage collection, and just-in-time (JIT) compilation on top of LLVM for CPUs and GPUs (e.g., JuliaGPU [21]).

¹<https://juliagpu.org/>

- A unified ecosystem that supports packaging, interoperability via foreign function interfaces, interactive computing, and tools, all of which are maintained by a community of practice in industry and academia.

Similar to Fortran in past generations of programming languages, Julia satisfies modern reproducibility and low-cost requirements for scientific software beyond number-crunching. Julia combines low-level (e.g., compilation, GPU programming) and high-level (e.g., garbage collection, unified packaging) aspects as a lower-cost alternative to developing code with a high-level language, such as Python, in combination with an HPC language [22]. Julia has caught the attention of researchers in various scientific fields, such as high-energy physics [23–25], biology [26], quantum chemistry [27–29], and machine learning [30]. This adoption signals the need for HPC capabilities—in particular, easy parallel computing access on CPUs and GPUs. Although Julia’s ecosystem is mature, work is ongoing to bring Julia to the same level of adoption and integration as the dominant HPC languages. One notable example is the development of the recently available `juliac` static compiler [31].

Julia in HPC

Julia has been explored for use in HPC across several different areas. Churavy et al. [32] present a summary of the value proposition of Julia’s HPC ecosystem for programming models—Julia’s Threads, CUDA.jl [21], AMDGPU.jl [33], oneAPI.jl [34], the GPU-portable KernelAbstractions.jl [35, 36] library, and the MPI.jl [37] wrapper for communication using the Message Passing Interface (MPI) [38]. Regier et al. [39] showcased Julia’s viability for HPC on the CPU-based Cori supercomputer. Godoy et al. [40] studied the unifying value of Julia for simulation and data analysis on the Frontier exascale system. Giordano et al. [41] and Hunold et al. [42] reported negligible overheads in MPI communication benchmarks. Application frameworks and case studies have been reported by Ramadhan et al. [43] (global ocean simulations), Ranocha et al. [44] and Schlottke-Lakemper et al. [45] (Trixi.jl for partial differential equation solvers), Marras et al. [46] (Jexpresso.jl for general systems of conservation laws, the first application framework to embed JACC.jl), Shang et al. [47] (quantum chemistry on the Sunway TaihuLight supercomputer), Martorell et al. [48] (finite elements), and Wegert et al. [49] (level-set toolbox).

Performance portability in Julia has been studied by De la Calle and García [50], who introduced a tool to translate CUDA.jl to portable KernelAbstractions.jl code and demonstrated small overheads on science use cases such as the compute-bound miniBUDE (near-zero) and the memory-bound BabelStream (9%) miniapps. Briand et al. [51] compared Julia’s Polyester.jl for CPU and KernelAbstractions.jl for GPU with Kokkos for memory-bound Cartesian computational fluid dynamics (CFD) solvers. Their work showed small overheads for those memory-bound workloads, making Julia a good fit for the CFD applications. Godoy et al. [52] showed that Julia vendor-specific CPU and GPU models have higher performance portability than that of Python’s Numba and Kokkos for a simple matrix multiplication case. Lin et al. [53] tested CPU and GPU programming models in Julia and compared them with C++ implementations of the miniBUDE and BabelStream applications. They found that Julia’s non-portable programming models are on par in terms of performance, but

manual ports are still needed for CPU and GPU architectures, with KernelAbstractions.jl being the exception for GPUs. Schanen et al. [54] presented ExaPF.jl, one of the first performance-portable frameworks in Julia: a differentiable power flow solver built on KernelAbstractions.jl in which all core components, the nonlinear solver, linear solvers, automatic differentiation, and the power grid model, execute on all three major GPU architectures (NVIDIA, AMD, and Intel).

Therefore, the motivation for JACC v1.0 is to provide a robust framework that elevates CPU/GPU performance-portable programming to make it accessible as Julia adoption continues to grow for HPC users.

3 JACC APIs and Architecture

This section describes the JACC public APIs and internal architecture, which leverage Julia’s unique characteristics to combine scientific productivity and performance.

3.1 JACC APIs

At the core of JACC is a simple but powerful API built around portable abstractions for the following:

- **Backend selection:** outside user code: `set_backend`, `remove_backend`; in portable user code: `@init_backend`
- **Array allocations:** `array`, `ones/zeros`, `fill`, `shared`
- **Kernel launching:** `parallel_for` and `parallel_reduce`

These abstractions allow users to express parallelism and data movement in a way that is portable across architectures and consistent with Julia’s mathematical syntax and its use in scientific computing. For example, the program in Fig. 1 shows that a meaningful kernel can be implemented in JACC in only 4 lines and run on any CPU or GPU from any major vendor, making it simpler than any other implementation in any language and programming system. Thus, users can exploit their laptops’ CPU or GPU (NVIDIA, AMD, Intel, Apple) for interactive development (e.g., using the Julia read-eval-print loop, REPL) and use the same code on the fastest GPU-powered (NVIDIA, AMD, Intel) and CPU-powered supercomputers in the world.

Figure 2 illustrates JACC’s backend selection mechanisms, which use Julia’s widely adopted Preferences.jl mechanisms to configure vendor-specific backends outside the portable user code, similar to the approach used by MPI.jl for detecting underlying MPI implementations. By default, all but the CPU Threads backend packages are optional through using Julia’s standard weak dependencies features; this minimizes potential costs and conflicts when a particular hardware is not supported (e.g., AMDGPU.jl on NVIDIA or Intel systems).

Kernel functions in JACC can be defined by any function in Julia that expresses independent workloads for each unique index (i,j,k) in a multi-dimensional range (Nx, Ny, Nz). An example of such a function is shown in Fig. 3a, in which indices are always defined first in the function signature, followed by regular function arguments (`args...`). Due to Julia’s flexibility, anonymous and “do-style” function variations are also supported by JACC (like the one defined in-place in Fig. 1).

```

[julia> using JACC

[julia> JACC.@init_backend
[ Info: Metal backend loaded

[julia> matrix = JACC.zeros((10,10));

[julia> JACC.@parallel_for range=(10,10) ((i,j,x)->x[i,j] = i+j-1)(matrix);

[julia> matrix
10×10 Metal.MtlMatrix{Float32, Metal.PrivateStorage}:
 1.0  2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0
 2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0 11.0
 3.0  4.0  5.0  6.0  7.0  8.0  9.0 10.0 11.0 12.0
 4.0  5.0  6.0  7.0  8.0  9.0 10.0 11.0 12.0 13.0
 5.0  6.0  7.0  8.0  9.0 10.0 11.0 12.0 13.0 14.0
 6.0  7.0  8.0  9.0 10.0 11.0 12.0 13.0 14.0 15.0
 7.0  8.0  9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0
 8.0  9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0 17.0
 9.0 10.0 11.0 12.0 13.0 14.0 15.0 16.0 17.0 18.0
10.0 11.0 12.0 13.0 14.0 15.0 16.0 17.0 18.0 19.0

```

Fig. 1: JACC vendor-agnostic code example on the interactive Julia REPL assigning matrix values on an Apple M3 GPU.

```

1 JACC.set_backend("AMDGPU")
2 # restart Julia session or
3 julia --project -e 'using JACC; JACC.set_backend("AMDGPU")'
4 # Options: Threads (default), CUDA, AMDGPU, oneAPI, Metal

1 ## LocalPreferences.toml
2 [JACC]
3 backends = ["amdgpu"]
4 default_backend = "amdgpu"
5
6 [JACC.placement]
7 amdgpu = "weakdeps"

1 # portable user code
2 import JACC
3 JACC.@init_backend

```

Fig. 2: JACC backend selection mechanisms reusing Preferences.jl and LocalPreferences.toml outside user code.

Basic kernel launching APIs

Shown in Fig. 3b are the different basic signatures and the reasoning behind them, which is to allow users to adapt to the style with which they are more familiar. This simple use of JACC allows users to call `parallel_for` and `parallel_reduce` signatures with the range and kernel functions as the only mandatory parameters, thus completely abstracting away hardware- and backend-specific details that are not necessarily portable (e.g., the GPU-only blocks and threads). In addition, the macro

signatures follow Julia’s style to provide equivalent macro and regular function signatures. It is important to note that all the signatures shown converge internally on the same kernel launching mechanisms, which preserves the JACC design requirement of adding minimal overhead on top of its backends, as reported by Valero-Lara et al. [20].

Advanced kernel launching APIs

As shown in Fig. 3c, the same basic APIs shown in Fig. 3b can be extended by adding more control parameters to the `parallel_for` and `parallel_reduce`. As seen, the added knobs are based on the internal and extensible `LaunchSpec` struct, which allows users to tune hardware-specific aspects of kernel launching at the cost of some portability. Current control knobs include those for setting the number of GPU `blocks` and `threads`, launching a kernel synchronously (`sync`, on by default) or asynchronously, shared memory size (`shmem_size`), and the `stream` used for kernel execution (e.g., CUDA and HIP streams or oneAPI queues). When a knob is not portable (e.g., setting GPU blocks on CPU), the option is simply ignored to favor productivity without reducing code portability. As new application needs arise, the set of launch specifications can be easily adapted and extended to new architectures. Thus, following Julia’s productivity goals, JACC effectively separates domain-specific functionality (or computational science) from hardware-specific optimizations (or computer science) in its APIs.

```

1 function f(i, j, k, ..., n, args...)
2 end

```

(a) JACC kernel structure: plain Julia functions beginning with index parameters (i,j,k,..., n) followed by user parameters.

```

1 # function first, Julia-like
2 parallel_for(f, range, x...)
3 parallel_reduce(f, range, x...; [type], [op], [init])
4
5 # range first, Kokkos-like
6 parallel_for(range, f, x...)
7 parallel_reduce(range, f, x...; [type], [op], [init])
8
9 # keyword arguments
10 parallel_for(; range=range, f=f, (args...))
11 parallel_reduce(; range, f, args, [type], [op], [init])
12
13 # macro version
14 @parallel_for range=range f(x...)
15 @parallel_reduce range=range [type=...] [op=...] [init=...] f(args...)
16
17 # where:
18 op = {+, *, min, max, <user-defined>}
19 # defaults:
20 init = { 0, 1, inf, -inf, <user-defined> }
21 type = JACC.default_float() either Float32 or Float64

```

(b) JACC basic APIs

```

1 spec = JACC.launch_spec(; blocks=blocks, threads=threads, sync=false)
2 # LaunchSpec parameters (all optional):
3 stream, threads, blocks, shmem_size, sync
4
5 # function first, Julia-like
6 parallel_for(f, spec, range, args...)
7 parallel_reduce(f, spec, range, args...; [type], [op], [init])
8
9 # range first, Kokkos-like
10 parallel_for(spec, range, f, args...)
11 parallel_reduce(spec, range, f, args...; [type], [op], [init])
12
13 # keyword arguments
14 parallel_for(; spec, range=range, f=f, (args...))
15 parallel_reduce(; spec, range=range, f=f, args..., [type], [op], [init])
16
17 # macro version
18 @parallel_for range=range blocks=blocks threads=threads shmem_size=shmem_size
19   sync=true stream=stream_handler f(args...)
19 @parallel_reduce range=range [type=...] [op=...] [init=...] blocks=blocks
   threads=threads shmem_size=shmem_size sync=true stream=stream_handler} f(
   args...)

```

(c) JACC advanced APIs via the LaunchSpec struct

Fig. 3: JACC APIs

Interoperability with Julia’s HPC ecosystem

Because of JACC’s metaprogramming capabilities, it is fully interoperable with vendor backends in JuliaGPU. As shown in Fig. 4, portable JACC code co-exists with (a) the portable kernel launching mechanisms of KernelAbstractions.jl and (b) vendor-specific CUDA.jl (or any other vendor) functionality, which ties JACC code path to a single vendor backend. This design decision enables writing JACC memory and kernel code in combination with KernelAbstractions.jl code that is entirely portable (no single line of code related to vendor-specific information), as well as piece-wise porting of vendor-specific kernel backend code (e.g., when CUDA.jl has been used for prototyping) to a fully portable JACC implementation. In addition, JACC’s integration with the JuliaGPU ecosystem extends the backends’ existing interoperability with tools (e.g., vendor profilers), libraries, and the Julia standard library without the need for any additional effort. JuliaGPU interoperability is critical in HPC—such as in CUDA-aware, ROCm-aware, and oneAPI-aware MPI.jl communications and other library ecosystems (e.g., parallel I/O, visualization)—because it allows developers to reuse existing HPC software.

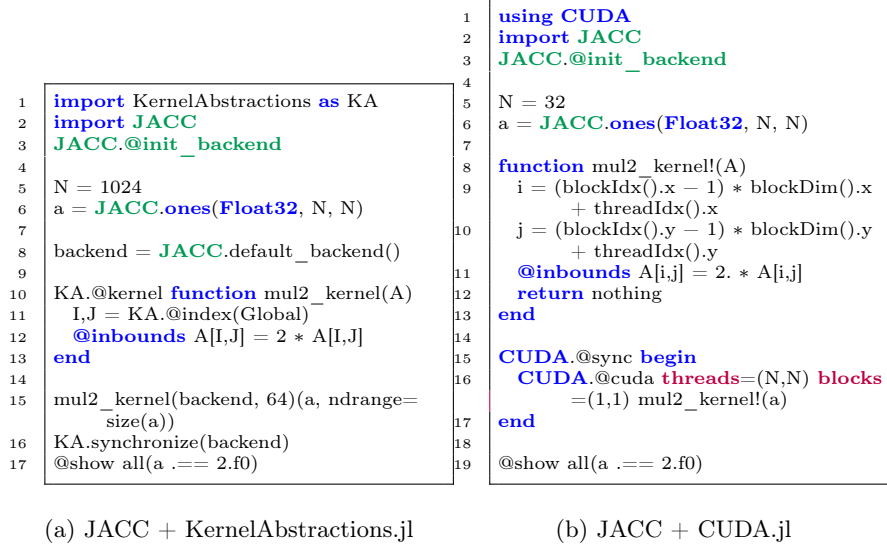


Fig. 4: Interoperability between JACC and JuliaGPU models.

Differences relative to KernelAbstractions.jl and Kokkos syntax

Figure 5 showcases syntax differences between (a) a minimal `parallel_for` kernel written in Kokkos, (b) KernelAbstractions.jl, and (c) JACC for the simple triad kernel. Even for a minimal `parallel_for` kernel, Kokkos incorporates several custom constructs (e.g., `View`, `KOKKOS_LAMBDA`) and pointers that must be added in a general language like C++ . KernelAbstractions.jl is a portable CUDA/HIP-like fine-granularity model in the JuliaGPU ecosystem exclusively for GPU kernel portability.

As seen from the code snippet and according to their latest documentation, users must track vendor-specific memory allocation and kernel launching. JACC can be seen as a higher-level programming model than KernelAbstractions.jl: JACC users can adapt between coarse (CPU) and fine (GPU) granularity using the widely adopted Julia Preferences.jl package mechanism (`LocalPreferences.toml`) so that user code does not contain any vendor-specific information. Thus, JACC is closer to the Kokkos, RAJA, or SYCL APIs for C++ , as they all aim to enable users to focus on writing one code and offload low-level aspects to the library, but JACC provides a more concise and targeted interface thanks to Julia’s metaprogramming and mathematical syntax. Nevertheless, Julia’s performance portability libraries still require a great deal of testing and evaluation to improve the language’s capabilities for high productivity and performance.

```

1 template <class T> void KokkosStream<T>::triad(){
2   Kokkos::View<T*> a(*d_a); ...
3   Kokkos::parallel_for(array_size, KOKKOS_LAMBDA (const long index){ a[index] = b[
     index] + alpha * c[index]; });
4   Kokkos::fence(); }

```

(a) C++ Kokkos `parallel_for` API

<pre> 1 import KernelAbstractions as KA 2 using CUDA: CuArray 3 #using AMDGPU: ROCArray 4 #using oneAPI: oneArray 5 #using Metal: MtlArray 6 7 N = 10_000 8 a = CuArray.zeros(Float32, N, N) 9 b = CuArray.ones(Float32, N, N) 10 c = CuArray.ones(Float32, N, N) 11 #a = ROCArray.zeros(Float32, N, N) 12 #b = ROCArray.ones(Float32, N, N) 13 #c = ROCArray.ones(Float32, N, N) 14 ... 15 alpha = Float32(1.0) 16 17 KA.@kernel function triad(a, b, alpha, c) 18 I, J = KA.@index(Global) 19 @inbounds a[I,J] = b[I,J] + alpha * c[I ,J] 20 end 21 22 backend = KA.get_backend(a) 23 workgroup_size = 64 24 triad(backend, workgroup_size)(a, b, alpha, c, ndrange=size(a)) 25 KA.synchronize(backend) 26 end </pre>	<pre> 1 # Set a backend with Preferences.jl 2 import JACC 3 # "CPU" or 4 # "CUDA", "AMDGPU", "oneAPI", "Metal" 5 JACC.set_backend("CUDA") 6 7 8 9 10 # User code 11 import JACC 12 JACC.@init_backend 13 14 function triad(i, j, a, b, alpha, c) 15 @inbounds a[i,j] = b[i,j] + alpha * c[i,j] 16 return 17 end 18 19 N = 10_000 20 a = JACC.zeros(Float32, (N,N)) 21 b = JACC.ones(Float32, (N,N)) 22 c = JACC.ones(Float32, (N,N)) 23 alpha = Float32(1.0) 24 25 JACC.@parallel_for range=(N,N) triad(a, b, alpha, c) 26 27 28 29 30 ### LocalPreferences.toml 31 [JACC] 32 backend = "CUDA" </pre>
---	--

(b) Julia KernelAbstractions.jl from the v0.9.41 documentation

(c) Julia JACC portable APIs

Fig. 5: BabelStream `triad` kernel code sample showing the syntax differences between portable programming models.

3.2 Advancing performance-portable programming

JACC’s capabilities have continued to evolve alongside advancements in HPC node architectures. The latest trend is the increasing use of multiple GPUs per node in pre-exascale and exascale systems coupled with a reduction in the total number of nodes in the overall system. Most current exascale applications targeting multiple GPUs on the same node were designed to follow the one-accelerator-per-MPI-process model [55], a pattern that continues to be unchanged despite the added partitioning and communication overhead and the increasing resources at the node level.

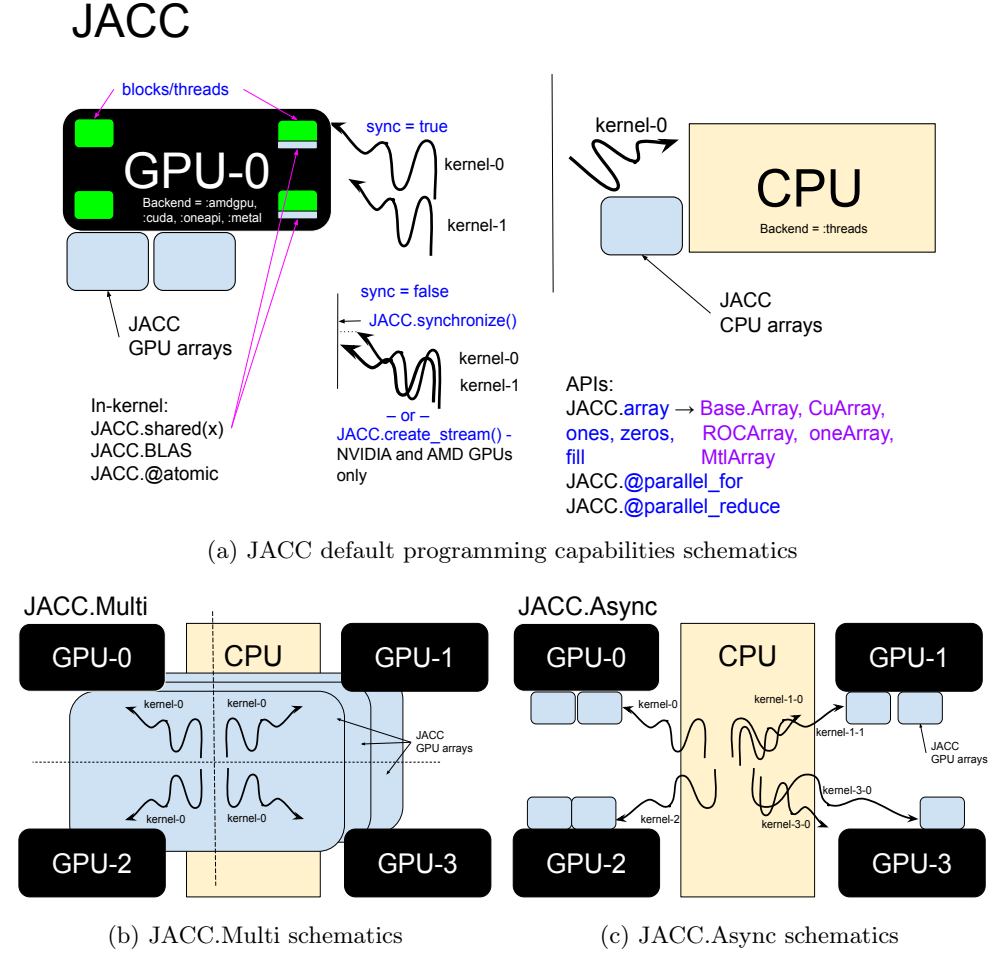


Fig. 6: JACC programming models for exascale nodes

As seen in Fig. 6, JACC v1.0 provides additional programming models to explore suitable paradigms for the exascale era. First, Fig. 6a presents the canonical single

CPU or GPU kernel launching mechanisms of the default APIs shown in Fig. 3. This model is suitable for development environments at small scales (laptops, desktop, single-GPU nodes, or cloud systems for testing and deployment). Due to its metaprogramming nature, JACC array types are compiled as aliases for the vendor-specific backend of choice (e.g., Julia CPU Array, CuArray, ROCArray). As shown in Fig. 7, this model allows for competitive performance portability for the common conjugate gradient (CG) algorithm on a single CPU or GPU, as reported by Valero-Lara et al. [20], relative to JuliaGPU vendor-specific implementations. We show that JACC offers minimal overhead with respect to its JuliaGPU backends, which is a promising starting point toward achieving full performance on more domain-specific workloads.

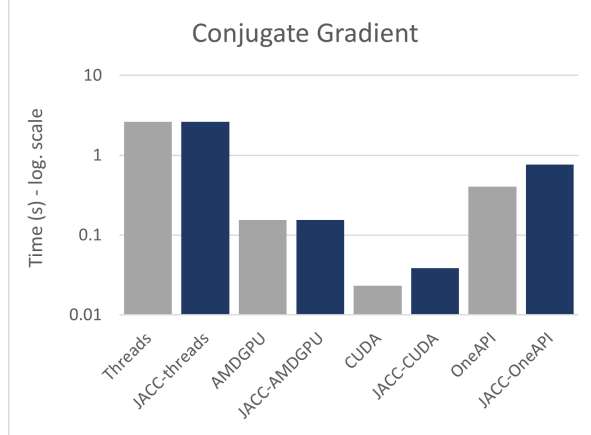


Fig. 7: JACC CG times on an AMD Rome CPU, AMD MI100 GPU, NVIDIA A100 GPU, and Intel MAX 1550 GPU compared with those of the JuliaGPU backends from Valero-Lara et al. [20].

Other advanced capabilities include the creation of several GPU compute streams on the same device. This allows for concurrent execution of independent kernels on a single GPU such that users can overlap separate parts of their calculations and make better use of available GPU compute resources. JACC also provides capabilities required in user kernel development (“in-kernel”): shared memory (JACC.shared), BLAS subroutines (JACC.BLAS), and atomic operations (JACC.@atomic). JACC.shared provides a way to allocate and exploit read-only memory caches (e.g., block-level memory) on GPUs. Figure 8 shows that a simple line of code, `JACC.shared(x)`, for reusing block-level “shared” GPU memory can accelerate convolution operations that involve a matrix $\times$ a filter (i.e., small read-only matrix), which are critical for certain classes of AI kernels. Depending on the array input size and convolution filter size, speedups of 2–3 $\times$ on an NVIDIA A100 GPU and 1–2 $\times$ on an AMD MI250X GPU can be achieved.

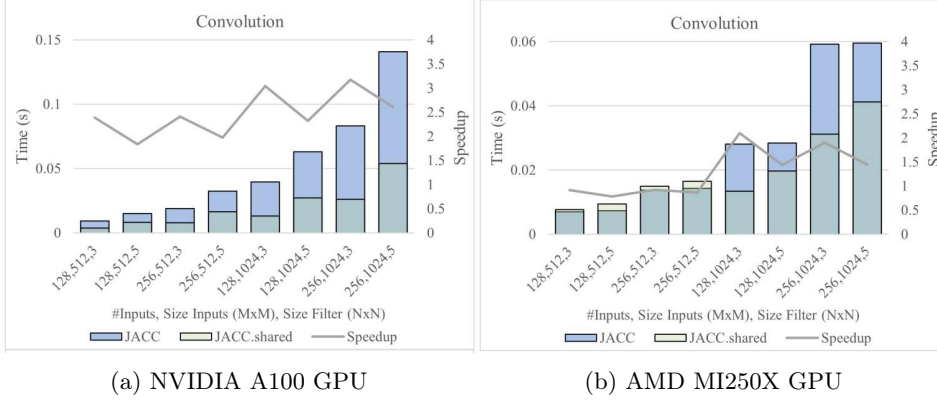


Fig. 8: JACC.shared execution time and speedup for a convolution operation on (a) a Perlmutter NVIDIA A100 and (b) a Frontier AMD MI250X GPU node from Valero-Lara et al. [56].

JACC also provides access to the widely used BLAS libraries in a portable way. The current version provides access to level 1 subroutines, and we plan to expand access to other levels as needed by applications and as available in the JuliaGPU ecosystem (e.g., wrapping vendor BLAS libraries). Atomic operations inside JACC kernels are available via the `Atomix.@atomic` macro (imported as `JACC.@atomic`), which is already integrated and supported by the current JuliaGPU backends packages. Thus, users can leverage out-of-the-box advanced functionality for writing their kernels.

Two newly integrated programming paradigms, JACC.Multi and JACC.Async, allow JACC users to explore multi-GPU execution on a single node for intra-node acceleration without the need for complex MPI APIs for partitioning and communication. The JACC.Multi programming paradigm is shown in Fig. 6b in which a single kernel (e.g., kernel-0) can be executed using a single-instruction, multiple-data model, partitioning arrays, and kernel workloads across the available GPUs on the node. As reported by Valero-Lara et al. [57] and shown in Fig. 9, JACC.Multi implementations of an Euler 5-point stencil solver (widely used in transport phenomena in fluid dynamics, radiation and thermal transfer, etc.) can achieve speedups of $1.7\times$ and $5\times$ on NVIDIA (Hudson, 2 H100 GPUs) and AMD (Frontier, 8 MI250X GPUs) multi-GPU powered nodes when compared to the canonical JACC implementation using the one-GPU-per-MPI-process model. Thus, JACC.Multi provides a path to performance as more GPUs are added per HPC node.

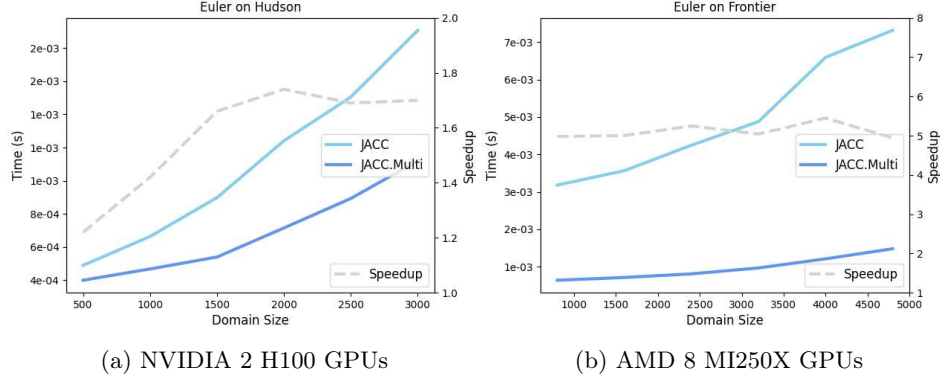


Fig. 9: JACC.Multi execution time and speedup for the Euler method on (a) a Hudson NVIDIA H100 2-GPU node and (b) a Frontier AMD MI250X 8-GPU node from Valero-Lara et al. [57].

Similarly, JACC.Async extends the concept of asynchronous task-based programming to multiple GPU accelerators. As shown in Fig. 6c, JACC.Async enables users to launch different concurrent kernels across the GPUs available within a compute node. The advantages and trade-offs of these models and their JACC implementations are still under exploration, and these paradigms are expected to be integrated with existing inter-node implementations of MPI.jl to support large-scale runs. Such approaches may be particularly useful for new critical applications—such as GPU workloads driven by AI training, convolution, or inference—that can benefit from minimizing intra-node overheads.

Programming for JACC.Multi and JACC.Async is written in JACC with a simple modular interface that allows users to port their codes to these new paradigms. As seen in Fig. 10 for simple axpy and dot product operations, porting from an initial canonical single-CPU/GPU JACC implementation (Fig. 10b) is straightforward for both models. Using the same kernel functions defined in Fig. 10a, the resulting JACC.Multi code in Fig. 10c requires only adding the submodule name “Multi”. Users can implement device-specific logic inside their function, but the expectation is that most workloads should be balanced across all multi-GPU devices. Similarly, and because of the already independent nature of each JACC array and kernel, porting to an explicit task-based model implementation via JACC.Async is a low-effort task. Figure 10d showcases the use of JACC.Async for independent axpy and dot product operations. The key element is the inclusion of a “task id” to identify each asynchronous execution context that is independent of the underlying hardware (e.g., number of GPUs, execution stream, etc.). JACC will then schedule the tasks using a round-robin strategy, abstracting all the implementation details from users who want to incorporate productive task-based parallelism in their algorithms.

```

1  # Multidimensional arrays
2  SIZE = 1_000
3  x = round.(rand(Float64,SIZE,SIZE)*100)
4  y = round.(rand(Float64,SIZE,SIZE)*100)
5  al = 2.5
6
7  function axpy(i, j, alpha, x, y)
8      @inbounds x[i,j] = x[i,j] + alpha * y[i,j]
9  end
10
11 function dot(i, j, x, y)
12     @inbounds return x[i,j] * y[i,j]
13 end

```

(a) CPU Arrays and Kernel Functions

```

1  ...
2
3  dx = JACC.array(x)
4  dy = JACC.array(y)
5
6  JACC.parallel_for((SIZE,SIZE),
7      axpy, al, dx, dy)
8
9  res = JACC.parallel_reduce(
10     (SIZE,SIZE), dot, dx, dy)

```

(b) JACC axpy/dot

```

1  ...
2
3  dx = JACC.Multi.array(x)
4  dy = JACC.Multi.array(y)
5
6  JACC.Multi.parallel_for((SIZE,SIZE),
7      axpy, al, dx, dy)
8
9  res = JACC.Multi.parallel_reduce(
10     (SIZE,SIZE), dot, dx, dy)

```

(c) JACC.Multi axpy/dot

```

1  ...
2
3  dx1 = JACC.Async.array(1, x)
4  dx2 = JACC.Async.array(2, x)
5  dy1 = JACC.Async.array(1, y)
6  dy2 = JACC.Async.array(2, y)
7
8  JACC.Async.parallel_for(1,
9      (SIZE,SIZE), axpy, alpha, dx1, dy1)
10
11 res = JACC.Async.parallel_reduce(2,
12     (SIZE,SIZE), dot, dx2, dy2)
13
14 JACC.Async.synchronize()
15 # optionally
16 # JACC.Async.synchronize(task_id)

```

(d) JACC.Async axpy/dot

Fig. 10: JACC single CPU/GPU, Multi-GPU and Async implementations of axpy

3.3 JACC’s Internal Architecture

JACC’s high-level capabilities are possible due to the pre-existing functionality in Julia and the JuliaGPU ecosystem. As shown in Fig. 11, JACC’s internal architecture unifies the existing Julia CPU multithreading (via Polyester.jl and the built-in Base.Threads) and GPU backends that build on top of the Julia, LLVM, and vendor-specific toolchains. To achieve this unique high-level “all-in-one” framework, JACC makes novel use of the Julia language’s distinctive features and how they impact each aspect of the development process.

- *JIT metaprogramming and LLVM compilation:* JACC lowers plain Julia functions to native CPU or GPU code at runtime. All JuliaGPU backends (CUDA.jl, AMDGPU.jl, oneAPI.jl, Metal.jl) share GPUCompiler.jl [58, 59] and LLVM.jl as a common kernel compilation layer that lowers Julia IR to the vendor targets (PTX, GCN, SPIR-V, AIR). JACC inherits this shared infrastructure, so supporting a

new vendor carries near-zero incremental compilation cost, and it enables interoperability with the broader Julia HPC ecosystem—JuliaGPU, vendor tools, MPI.jl, and KernelAbstractions.jl—through a single high-level codebase.

- *Precompilation and Multiple Dispatch*: Precompiling JACC with its selected backend reduces JIT times and multiple-dispatch helps maintain near-zero performance overhead compared to that of the native Julia backends.
- *Packaging*: Backends (except Polyester.jl) are maintained as optional dependencies to reduce download times, avoid conflict with unsupported hardware, and allow automatic dependency management through Pkg.jl.
- *Interactive programming*: JACC supports seamless exploratory programming on CPUs and GPUs through the interactive Julia REPL.
- *Dynamic type system*: ensures compatibility between supported data types and the underlying hardware, with some hardware-specific exceptions (e.g., unsupported FP64 on Apple and some Intel GPUs).

The result is an internal layered architecture that builds on each vendor-specific backend, with calls from scientific HPC applications dynamically dispatched using JACC. JACC leverages all these community investments, adopting a “backend-first,” extensible development model that shifts development costs away from the end user.

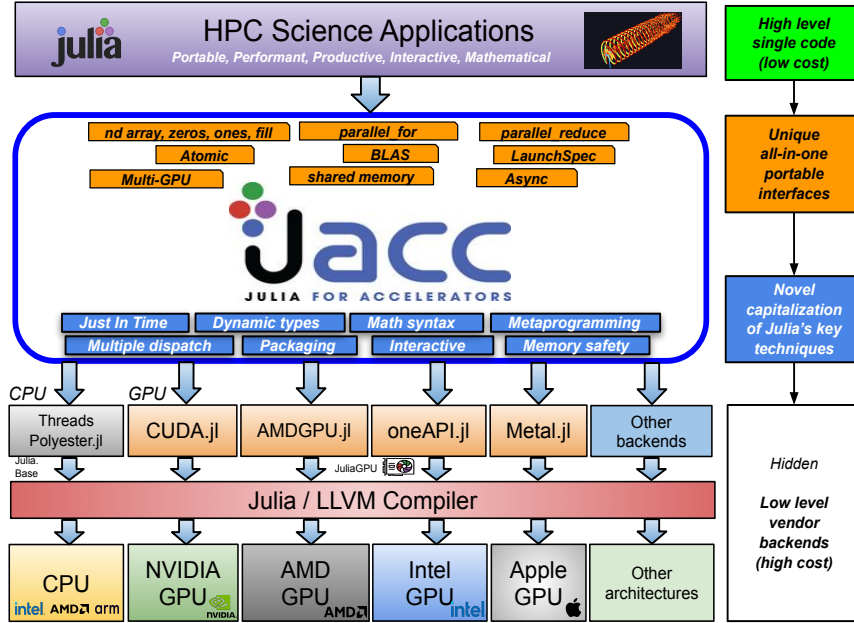


Fig. 11: JACC schematic architecture, which leverages Julia’s unique programming model capabilities and CPU and GPU backends.

Software engineering

JACC follows a canonical Julia package structure, as depicted in Fig. 12. This modular design allows each module and backend-specific extension to be defined separately, giving a clear scope for decentralized development. Although Julia lacks object-oriented programming capabilities such as those in Python or C++ , the current design is based on Julia’s modules and sub-modules, which enables a clear separation between backend-specific connectors in `src` for CPU Threads and those in `ext` for optional vendor-specific GPU backend packages. In addition, JACC uses the `ReTest.jl` package for unit testing and continuous integration (CI): GitHub Actions runners are used for CPUs, and Oak Ridge National Laboratory’s (ORNL’s) Experimental Computing Laboratory (ExCL)² is used for GPU tests. The JACC repository follows best practices and is recognized with an OpenSSF badge, which enables coordinated contributions and advancement of the software capabilities. JACC leverages recent general features in the Julia language (version 1.12 as of mid-2026)—in particular, the inclusion of optional weak package dependencies in Julia v1.9 and more aggressive precompilation capabilities since the Julia v1.10 release.

²<https://www.excl.ornl.gov/>

JACC.jl



Fig. 12: JACC extensible internal code structure, which reuses Julia’s package structure and includes optional package dependencies (**ext**) for GPU backends.

4 Impact on Science Users

This section describes some of the reported use cases that have seen early adoption of JACC and examines their impact on scientific application development beyond the benefits of performance portability. This section also illustrates the multi-disciplinary appeal of Julia and JACC for actual HPC users.

Computational Fluid Dynamics

JACC is currently being used at the European Centre for Research and Advanced Training in Scientific Computation (CERFACS) for the development of next-generation CFD codes. CERFACS was exploring how to write CFD codes with four key requirements—performance, portability, ease of use, and a low barrier to entry—so that scientists, not just software experts, can read, understand, and contribute to the

code. The Julia language was one of the most promising paths to develop lattice Boltzmann solvers for a real CFD application. JACC is a key enabler in Julia’s adoption, as the same codebase ran on NVIDIA, AMD, and Intel GPUs as well as multi-threaded CPUs. As seen in Fig. 13, the Julia+JACC implementation (BLAST-JACC) ran with competitive portable performance when compared to the more complex C++ OpenCL state-of-the-art FluidX3D code [60], and it did so without the need to maintain separate vendor-specific backend kernels. When Metal support was later added to JACC, BLAST ran on Apple GPUs with zero changes, which enabled rapid development of new large-scale HPC solver code on Apple laptops.

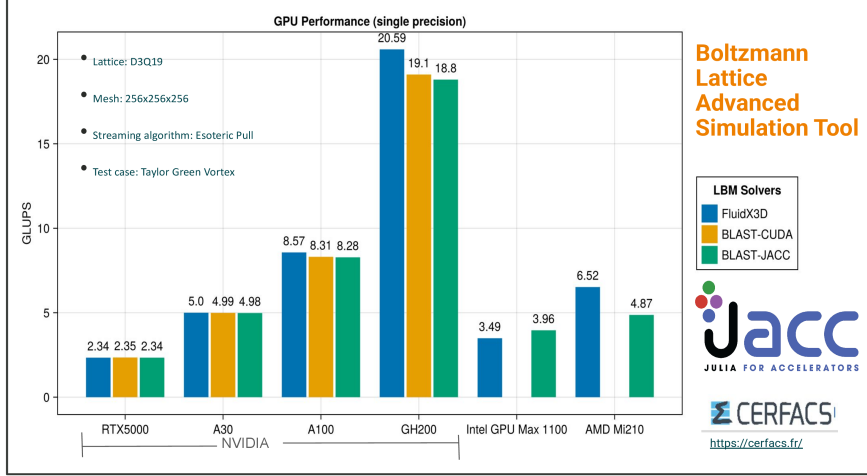


Fig. 13: JACC performance-portable results on a real CFD code lattice Boltzmann method solver. Credit: CERFACS.

Multiphysics Solvers

Similarly, JACC has been adopted by scientists at the New Jersey Institute of Technology (NJIT) for their Jexpresso [61] framework for multiphysics applications such as fluid dynamics, radiative transfer, and magnetohydrodynamics. JACC enables Jexpresso to run performance-portable CPU/GPU simulations with performance on par with NJIT’s legacy Fortran codes, as reported by Marras et al. [46]. JACC enables seamless interoperability with MPI.jl for a combined intra-/inter-node parallelization of the solver and allows multi-threading, which proved to be extremely advantageous for applications such as Jexpresso in leveraging large nodes with up to 128 cores.

Science Facilities

As reported by Hahn et al. [62], JACC has been used to study the integration between experimental neutron scattering at the Spallation Neutron Source (SNS) and computational facilities at the Oak Ridge Leadership Computing Facility (OLCF), at

ORNL. JACC enabled scientists to conduct design exploration to speed up their state-of-the-practice CPU-only reduction workflows for neutron scattering cross section calculations. The study produced proxy applications that showed JACC and Julia’s ability to support performant CPU and GPU code for next-generation experimental science software.

Scientists at RIKEN, Japan’s leading national comprehensive research institute, are exploring performance-portable CPU/GPU programming systems for the next-generation FugakuNEXT supercomputer, equipped with Fujitsu CPUs and NVIDIA GPUs. This development is a departure from Fugaku, one of the two CPU-only HPC systems in the top 10 fastest supercomputers in the world, and the transition requires porting several CPU-only legacy codes (e.g., Fortran + OpenMP or OpenACC) to a CPU/GPU performance-portable programming environment. As presented in their basic technical report for Fugaku [63], JACC is one of the technologies being explored to enable this transition, as the Julia language has gained traction in recent years among users.

Quantum Chromodynamics

Recent work by Nagai et al. [64] demonstrates a new approach to developing hybrid Monte Carlo (HMC) simulations for lattice gauge theories, including lattice quantum chromodynamics (QCD). By combining JACC with the automatic differentiation (AD) package Enzyme.jl [65], the lattice action can be implemented once as performance-portable Julia code, while the corresponding HMC force is automatically generated through AD at the LLVM-IR level. Importantly, JACC allows both the original action calculation and the automatically differentiated force calculation to remain performance portable across CPU and GPU architectures. This approach eliminates the need to manually derive, implement, and maintain architecture-specific force kernels, which can be particularly labor-intensive and error-prone for complicated lattice actions. GPU implementations using JACC achieved reported speedups of approximately 40–50 $\times$ over the corresponding single-threaded CPU implementations while retaining a single source implementation. JACC has also been integrated into the JuliaQCD framework [66], in which it provides a performance-portable foundation for lattice QCD simulations on heterogeneous computing architectures.

Quantum Magnetism

Sunny.jl [67] is a Julia package for simulating spin dynamics in magnetic materials and computing the inelastic neutron scattering spectra measured at facilities such as ORNL’s SNS. Modeling chemical disorder requires running spin wave theory on supercells, which can take an hour or more to execute. Multimodal AI for 2D Quantum Magnets (MAIQMag), a DOE Genesis Mission AI Pathfinder effort applying machine learning to quantum magnetism, conducted exploratory work to accelerate this calculation and compared Julia’s portability options on two kernels that dominate its runtime. The JACC implementation of the reduction achieved performance within about twelve percent of a hand-written CUDA kernel on an NVIDIA H100, and the complete intensity calculation stayed within roughly ten percent of both CUDA and KernelAbstractions.jl on NVIDIA and AMD hardware. The same kernels ran from one

source across AMD, Apple, Intel, and NVIDIA GPUs. This accelerated path brings a large disordered supercell calculation from about an hour of serial computation down to seconds on a single GPU. This experiment shows that JACC’s higher-level abstraction costs little for kernels important to Sunny.

5 New Science Directions

This section discusses emerging scientific directions in which JACC is being explored for new user communities.

5.1 Quantum Computing

Quantum computing is a novel computing technology that uses quantum mechanics to carry out information processing tasks. The interest in quantum computation is due in part to the idea that there are, theoretically, algorithms that can perform certain computations much faster than can conventional computing approaches [68–71]. One of the central challenges of quantum computing currently is in constructing quantum processors such that they are relatively error free but can also execute large quantum programs. Doing so requires systematic error correction to be implemented in the processors in order to remove the effects of decoherence and various other sources of error. These technical challenges mean that quantum computing is currently in the “NISQ-era” [72], meaning noise largely dominates relatively short-depth circuit computations, and the total number of qubits that can be used in a single coherent computation is on the order of hundreds to, in some rare cases, thousands of qubits. On this path to large-scale quantum computers, there is a variety of HPC and parallel classical compute techniques that must be utilized in order to engineer the quantum algorithms and software that will eventually be needed. Here, we outline three classes (which are not comprehensive) of algorithms and techniques used in quantum computation that can be greatly accelerated by CPU or GPU parallelization in Julia via JACC.

The first is *quantum compilers*. Digital quantum algorithms are expressed as a quantum circuit. Like any other circuit abstraction, there must exist software infrastructure to adapt gate instructions to equivalent sets of instructions, route and compile a circuit to a particular quantum computer architecture, and, most importantly, optimize a given circuit to reduce the overall number of instructions that are required to express that operation. All of this compute is performed on a classical co-processor. There do exist current software libraries for expressing and optimizing quantum circuits; however, they are not currently built to be efficient (in memory or compute time) for very large-scale quantum circuits [73, 74]. Moreover, there exists a type of quantum compiling known as unitary synthesis, in which the goal is to compile an arbitrary quantum unitary U into a set of hardware-native universal instructions (such as CX , RZ , and H) [75]. This task is highly computationally intensive in general, and even with parallelized HPC techniques, the current state of unitary compilation produces circuits that are very large, making them effectively impossible to implement on near-term hardware. Parallelized acceleration of any aspect of quantum compilers would be useful in quantum computing because it would allow larger circuits to be

defined and optimized [76], and it would also allow smaller circuits to be expressed using fewer total quantum instructions.

The second is *quantum algorithmic validation*. Because no large-scale error-corrected quantum computers currently exist, it is challenging to implement quantum algorithms and test granular properties of these algorithms (e.g., hyperparameter tuning, exact resource–accuracy scaling, or the effects of approximation techniques). Therefore, for quantum algorithms the circuit sizes of which exceed the capabilities of current NISQ computers, the only current way to validate the performance of a quantum algorithm in detail is to use classical simulations of the quantum algorithm (such as using tensor network methods) [77–83]. In part, this aspect of the field includes the state-of-the-art competition between the largest quantum advantage demonstrations [81, 84–86] and clever classical algorithms paired with HPC compute [87–92]. These types of computations include NISQ computer benchmarking protocols that require classical compute, such as quantum volume [93–96]. These simulations inherently have a strongly exponential classical compute cost because such algorithms cannot be executed or implemented efficiently at scale classically. Therefore, they must be implemented using parallelization, HPC, GPU acceleration, and a variety of algorithmic approximations in order to run in any reasonable amount of time in the intermediate system size regime. Accelerator tools such as JACC could accelerate these numerical computations, such as tensor network methods or even full exact statevector calculations, which would then aid in improved understanding of quantum algorithmic properties for algorithms run on future quantum computers.

The third is *efficient quantum state and time evolution representation*. Certain types of quantum circuits exist that must be used as sub-routines in other quantum algorithms that are costly to implement because of their large quantum gate counts. A good example of this is the approximation of the time dynamics e^{iHt} , where H is a quantum Hamiltonian. Another good example is ground-state preparation of a quantum Hamiltonian. Quantum state preparation that has substantial overlap with a ground-state of a quantum Hamiltonian of interest is a critical sub-routine in the quantum phase estimation algorithm, and preparing such a state for large-scale quantum Hamiltonians is generally difficult. In both of these cases, there also exist certain algorithms, typically tensor network–based classical algorithms, that can prepare efficient circuit representations of these subroutines for arbitrarily large systems—provided that the quantum Hamiltonian satisfies certain connectivity restrictions. In particular, they typically need to be one-dimensional in order for these methods to achieve high accuracy [97–101]. These techniques can effectively be viewed as a type of machine learning as well as a type of quantum compilation. The limitation of these methods, however, is the upfront classical compute cost to define these states, which can be accelerated using CPU/GPU parallelism. The other limitation of these methods is that they are not necessarily generally applicable, at least for the current form of the algorithms. Once the recipe is devised to construct these states, they can then be implemented on a quantum computer and be used as a sub-routine. We aim to do the exploration taking advantage of the high-productivity path provided by JACC and the Julia language.

5.2 Intelligent Runtimes

JACC by itself provides a clean, vendor-agnostic interface for writing data-parallel kernels that can be compiled to either CPU or GPU backends. To fully exploit a truly heterogeneous multi-device environment, however, an intelligent runtime such as IRIS [102–104] is required. IRIS handles the orchestration of task kernels across heterogeneous devices, automatically mapping each kernel to the most suitable accelerator, managing data movement [105] between host and device memories, and launching multiple kernels in a task-parallel fashion. Its scheduler dynamically assigns streams on devices that support asynchronous execution, maximizes concurrency within a single GPU, and resolves data flow dependencies between kernel tasks so that execution order is preserved without manual synchronization.

The integration of JACC into this runtime environment decouples kernel implementation from hardware orchestration. While IRIS manages the high-level task graph and data movement, JACC provides the underlying hardware-agnostic capabilities required for execution. In this way, developers would not need to provide multiple vendor-specific implementations of the same kernel. Instead, a single JACC kernel can be dispatched by IRIS to any discovered accelerator. And the Julius framework that leverages this combination would enable true end-to-end portability such that both the task scheduling and the kernel execution are independent of the underlying hardware. It would thus do so without introducing the performance overhead typically associated with high-level abstractions.

```
1 using Julius
2
3 function axpy_jacc(i, Z, A, X, Y) # JACC kernel
4     @inbounds Z[i] = A*X[i] + Y[i]
5 end
6
7 X1, X2 = (rand(Float32, N), rand(Float32, N))
8 Y1, Y2 = (rand(Float32, N), rand(Float32, N))
9 (Z1, Z2) = (Array{Float32}(undef, N), Array{Float32}(undef, N))
10
11 Z = Array{Float32}(undef, N)
12 A = 10.0f0
13
14 task0 = Julius.@jparallel_for wait:false axpy(Z1, A, X1, Y1)
15 task1 = Julius.@jparallel_for wait:false axpy(Z2, A, X2, Y2)
16 task2 = Julius.@jparallel_for wait:false axpy(Z, A, Z1, Z2)
17 Julius.wait(task2)
18 println("Z:", Z)
```

Fig. 14: JACC AXPY kernel with Julius runtime tasks.

The Julius example shown above demonstrates how a single, type-invariant JACC kernel (AXPY) written in pure Julia can be compiled by JACC in a device-agnostic manner and executed across heterogeneous hardware. By submitting three `@jparallel_for` tasks—computing $Z_1 = A X_1 + Y_1$, $Z_2 = A X_2 + Y_2$, and finally $Z = A Z_1 + Z_2$ —Julius’ intelligent runtime automatically discovers all available CPUs or GPUs, orchestrates data movement, and schedules the kernels in parallel, leveraging multiple streams on

asynchronous devices while resolving data dependencies without explicit synchronization. Consequently, the same source code runs identically on CPU-only, single-GPU, or multi-GPU nodes and thus offers high productivity (through type-invariant, concise syntax), high performance (via automatic parallelization and dynamic runtime scheduling), and reduced development overhead.

5.3 Agentic AI-Assisted Development with JACC

The advent of LLMs and agentic AI has transformed the field of software engineering, and HPC code development is no exception [106, 107]. Recent research has shown that LLMs can reason about HPC parallel code [108–110], including more targeted languages like Julia [111–113]. Elevating the abstraction of HPC code via JACC would reduce the need for large context windows in LLMs such as those required for more general-purpose languages like Python or C++ . This is an area that still needs to be explored because HPC code is mostly written in compiled languages like C++ and Fortran. Julia and JACC could provide the crucial balance between agentic AI generation and human understanding as frontier models continue to advance, opening several opportunities for increased AI advantage in science.

6 Conclusions

In this paper, we present the first stable version of JACC, v1.0, a performance-portable library in the high-productivity LLVM-compiled Julia language. JACC provides a unified vendor-neutral API that enables users to write a single Julia code that can be deployed on CPUs and GPUs, leveraging the JuliaGPU ecosystem of vendor-specific backends. In this paper, we discuss the stable JACC v1.0 public APIs, the internal architecture that enables vendor-neutral computing while leveraging Julia’s unique capabilities, the impact on current scientific users, and future directions for the library, including science exploration in quantum computing, intelligent runtimes, and agentic AI. JACC closes a gap in the Julia ecosystem by providing a package that is well-suited for HPC prototyping: developers can write and test kernels on a laptop CPU or GPU and then deploy them to multi-GPU supercomputer nodes with minimal or no changes to the application code. Users of the default APIs do not need prior CPU/GPU programming knowledge to parallelize their codes because JACC provides low-level performance APIs (e.g., blocks, threads, async, shared memory, stream, multi-GPU, etc.) for hardware-specific optimizations. JACC also advances the programmability of the latest exascale computing nodes equipped with multiple GPUs via the novel single-instruction, multiple-data JACC.Multi and the task-based JACC.Async capabilities as a natural extension of the current JACC portable single-device APIs. In addition, JACC’s design enables interoperability with JuliaGPU’s KernelAbstractions and vendor-specific backends (CUDA.jl, AMDGPU.jl, oneAPI.jl and Metal.jl), MPI.jl for inter-node parallelization including GPU-aware communication, and performance analysis tools. This interoperability enables users of existing components to transition incrementally toward fully portable HPC code. This paper introduces the JACC open-source framework to a broader audience seeking to explore

new paradigms in their scientific computing application areas, and it contributes to the adoption of the Julia language and ecosystem for science.

Acknowledgements

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan <https://www.energy.gov/doe-public-access-plan>. This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research's Computer Science Competitive Portfolios program, MAGMA/Fairbanks project; and the Next Generation of Scientific Software Technologies program S4PST project. This research used resources of the Oak Ridge Leadership Computing Facility and the Experimental Computing Laboratory at the Oak Ridge National Laboratory, which are supported by the Office of Science of the US Department of Energy under Contract No. DE-AC05-00OR22725. This work was supported by the U.S. Department of Energy through the Los Alamos National Laboratory. Los Alamos National Laboratory is operated by Triad National Security, LLC, for the National Nuclear Security Administration of the U.S. Department of Energy (Contract No. 89233218CNA000001). E.P. was supported by the Laboratory Directed Research and Development program of Los Alamos National Laboratory under project number 20240032DR. Approved as LA-UR-26-26725. The authors would like to thank Mosè Giordano, Anton Smirnov, and the JuliaGPU community of developers for the helpful support and feedback.

References

- [1] Vetter, J.S., et al.: Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity (2018) <https://doi.org/10.2172/1473756>
- [2] Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004., pp. 75–86 (2004). IEEE
- [3] Backus, J.: The history of Fortran I, II, and III, pp. 25–74. Association for Computing Machinery, New York, NY, USA (1978). <https://doi.org/10.1145/800025.1198345>
- [4] Dongarra, J., Graybill, R., Harrod, W., Lucas, R., Lusk, E., Luszczek, P., McMahon, J., Snavely, A., Vetter, J., Yelick, K., Alam, S., Campbell, R., Carrington, L., Chen, T.-Y., Khalili, O., Meredith, J., Tikir, M.: DARPA's

- HPCS Program: History, Models, Tools, Languages. In: Advances in COMPUTERS. Advances in Computers, vol. 72, pp. 1–100. Elsevier, ??? (2008). [https://doi.org/10.1016/S0065-2458\(08\)00001-6](https://doi.org/10.1016/S0065-2458(08)00001-6)
- [5] Kothe, D., Lee, S., Qualters, I.: Exascale Computing in the United States. Computing in Science & Engineering **21**(1), 17–29 (2019) <https://doi.org/10.1109/MCSE.2018.2875366>
 - [6] Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. IEEE computational science and engineering **5**(1), 46–55 (1998)
 - [7] Wienke, S., *et al.*: Openacc—first experiences with real-world applications. In: Euro-Par 2012 Parallel Processing: 18th International Conference, pp. 859–870 (2012). Springer
 - [8] Stone, J.E., Gohara, D., Shi, G.: OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. Computing in Science & Engineering **12**(3), 66–73 (2010) <https://doi.org/10.1109/MCSE.2010.69>
 - [9] Deakin, T., McIntosh-Smith, S.: Evaluating the performance of hpc-style sycl applications. In: International Workshop on OpenCL, pp. 1–11 (2020)
 - [10] Johnston, B., *et al.*: Evaluating the performance and portability of contemporary SYCL implementations. In: IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 45–56 (2020). IEEE
 - [11] Trott, C.R., Lebrun-Grandié, D., Arndt, D., Ciesko, J., Dang, V., Ellingwood, N., Gayatri, R., Harvey, E., Hollman, D.S., Ibanez, D., Liber, N., Madsen, J., Miles, J., Poliakoff, D., Powell, A., Rajamanickam, S., Simberg, M., Sunderland, D., Turcksin, B., Wilke, J.: Kokkos 3: Programming model extensions for the exascale era. IEEE Transactions on Parallel and Distributed Systems **33**(4), 805–817 (2022) <https://doi.org/10.1109/TPDS.2021.3097283>
 - [12] Beckingsale, D.A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A.J., Pearce, O., Robinson, P., Ryujin, B.S., Scogland, T.R.: Raja: Portable performance for large-scale scientific applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 71–81 (2019). <https://doi.org/10.1109/P3HPC49587.2019.00012>
 - [13] Zenker, E., Worpitz, B., Widera, R., Huebl, A., Juckeland, G., Knüpfer, A., Nagel, W.E., Bussmann, M.: Alpaka – an abstraction library for parallel kernel acceleration. In: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 631–640 (2016). <https://doi.org/10.1109/IPDPSW.2016.50>
 - [14] Alexander, F., Almgren, A., Bell, J., Bhattacharjee, A., Chen, J., Colella, P.,

- Daniel, D., DeSlippe, J., Diachin, L., Draeger, E., Dubey, A., Dunning, T., Evans, T., Foster, I., Francois, M., Germann, T., Gordon, M., Habib, S., Halappanavar, M., Hamilton, S.P., Hart, W., Huang, Z., Hungerford, A., Kasen, D., Kent, P.R.C., Koley, T., Kothe, D.B., Kronfeld, A., Luo, Y., Mackenzie, P., McCallen, D., Messer, B., Mniszewski, S., Oehmen, C., Perazzo, A., Perez, D., Richards, D., Rider, W.J., Rieben, R., Roche, K., Siegel, A., Sprague, M., Steefel, C., Stevens, R., Syamlal, M., Taylor, M., Turner, J., Vay, J.-L., Voter, A.F., Windus, T.L., Yelick, K.: Exascale applications: skin in the game. *Philosophical Transactions of the Royal Society. A, Mathematical, Physical and Engineering Sciences* **378**(2166) (2020) <https://doi.org/10.1098/rsta.2019.0056>
- [15] Evans, T.M., Siegel, A., Draeger, E.W., Deslippe, J., Francois, M.M., Germann, T.C., Hart, W.E., Martin, D.F.: A survey of software implementations used by application codes in the Exascale Computing Project. *The International Journal of High Performance Computing Applications* **36**(1), 5–12 (2022) <https://doi.org/10.1177/10943420211028940>
- [16] Sorensen, B., Norton, A., Joseph, E., Conway, S.: Special report for NASA: Exploring options for a bespoke supercomputer targeted for weather and climate workloads. Hyperion Research, Saint Paul, MN, USA, Tech. Rep. HR4 (2019)
- [17] Kayraklioglu, E., Stone, A.: Productive, vendor-neutral gpu programming using chapel. In: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1914–1922 (2024). IEEE
- [18] Bezanson, J., *et al.*: Julia: A fresh approach to numerical computing. *SIAM Review* **59**(1), 65–98 (2017) <https://doi.org/10.1137/141000671>
- [19] Godoy, W.F., Melnichenko, T., Valero-Lara, P., Elwasif, W., Fackler, P., Da Silva, R.F., Teranishi, K., Vetter, J.S.: Mojo: Mlir-based performance-portable hpc science kernels on gpus for the python ecosystem (2025) <https://doi.org/10.1145/3731599.3767573>
- [20] Valero-Lara, P., Godoy, W.F., Mankad, H., Teranishi, K., Vetter, J.S., Blaschke, J., Schanen, M.: JACC: Leveraging HPC Meta-Programming and Performance Portability with the Just-in-Time and LLVM-based Julia Language. In: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1955–1966 (2024). <https://doi.org/10.1109/SCW63240.2024.00245>
- [21] Besard, T., Foket, C., De Sutter, B.: Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* **30**(4), 827–841 (2019) <https://doi.org/10.1109/TPDS.2018.2872064>
- [22] Straßel, D., Reusch, P., Keuper, J.: Python Workflows on HPC Systems. In: 2020 IEEE/ACM 9th Workshop on Python for High-Performance and Scientific

- Computing (PyHPC), pp. 32–40 (2020). <https://doi.org/10.1109/PyHPC51966.2020.00009>
- [23] Eschle, J., Gál, T., Giordano, M., Gras, P., Hegner, B., Heinrich, L., Hernandez Acosta, U., Kluth, S., Ling, J., Mato, P., et al.: Potential of the julia programming language for high energy physics computing. *Computing and Software for Big Science* **7**(1) (2023) <https://doi.org/10.1007/s41781-023-00104-x>
 - [24] Sagan, D., Hoffstaetter, G., Signorelli, M., Coxe, A.: Bmad-Julia: a Julia environment for accelerator simulations including machine learning. In: *Proceedings of the 15th International Particle Accelerator Conference (IPAC’24)*, Nashville, USA, pp. 2562–2565 (2024)
 - [25] Meneghini, O., Slendebroek, T., Lyons, B.C., McLaughlin, K., McClenaghan, J., Stagner, L., Harvey, J., Neiser, T.F., Ghiozzi, A., Dose, G., Guterl, J., Zalzal, A., Cote, T., Shi, N., Weisberg, D., Smith, S.P., Grierson, B.A., Candy, J.: FUSE (Fusion Synthesis Engine): A Next Generation Framework for Integrated Design of Fusion Pilot Plants (2024)
 - [26] Roesch, E., Greener, J.G., MacLean, A.L., Nassar, H., Rackauckas, C., Holy, T.E., Stumpf, M.P.: Julia for biologists. *Nature methods* **20**(5), 655–664 (2023)
 - [27] Aroeira, G.J.R., Davis, M.M., Turney, J.M., Schaefer, H.F.I.: Fermi.jl: A Modern Design for Quantum Chemistry. *Journal of Chemical Theory and Computation* **18**(2), 677–686 (2022) <https://doi.org/10.1021/acs.jctc.1c00719>
 - [28] Poole, D., Galvez Vallejo, J.L., Gordon, M.S.: A New Kid on the Block: Application of Julia to Hartree–Fock Calculations. *Journal of Chemical Theory and Computation* **16**(8), 5006–5013 (2020) <https://doi.org/10.1021/acs.jctc.0c00337>
 - [29] Gardner, J., Douglas-Gallardo, O.A., Stark, W.G., Westermayr, J., Janke, S.M., Habershon, S., Maurer, R.J.: Nqcdynamics. jl: A julia package for nonadiabatic quantum classical molecular dynamics in the condensed phase. *The Journal of Chemical Physics* **156**(17) (2022)
 - [30] Gao, K., Mei, G., Piccialli, F., Cuomo, S., Tu, J., Huo, Z.: Julia language in machine learning: Algorithms, applications, and open issues. *Computer Science Review* **37**, 100254 (2020)
 - [31] Carlson, F.B., Tapscott, C., Baraldi, G., Rackauckas, C.: C codegen considered unnecessary: go directly to binary, do not pass C. *Compilation of Julia code for deployment in model-based engineering* (2025)
 - [32] Churavy, V., Godoy, W.F., Bauer, C., Ranocha, H., Schlottke-Lakemper, M., Räss, L., Blaschke, J., Giordano, M., Schnetter, E., Omlin, S., Vetter, J.S., Edelman, A.: Bridging HPC Communities through the Julia Programming Language

(2022)

- [33] Samaroo, J., Smirnov, A., Churavy, V., Räss, L., Hodgson, T., Montoisson, A., Phillips, W., Ramadhan, A., Barmpareos, J., Besard, T., TagBot, J., Schanen, M., Bauer, C., Giordano, M., Arakaki, T., Antholzer, S., Alessandro, Elrod, C., Baraldi, G., Ranocha, H., Kunz, M., Raayai, M., Gymnich, T., Hu, T.: JuliaGPU/AMDGPU.jl: V0.7.3. <https://doi.org/10.5281/zenodo.10040461>
- [34] Besard, T.: oneAPI.jl. <https://doi.org/10.5281/zenodo.14931098> . <https://doi.org/10.5281/zenodo.14931098>
- [35] Churavy, V., Aluthge, D., Wilcox, L.C., Schloss, J., Byrne, S., Waruszewski, M., Samaroo, J., Ramadhan, A., Meredith, Schaub, S., Bolewski, J., Smirnov, A., Kawczynski, C., Hill, C., Liu, J., Schulz, O., Oscar, Haraldsson, P., Arakaki, T., Besard, T.: JuliaGPU/KernelAbstractions.jl: V0.8.3. <https://doi.org/10.5281/zenodo.6742177> . <https://doi.org/10.5281/zenodo.6742177>
- [36] Utkarsh, U., Churavy, V., Ma, Y., Besard, T., Srisuma, P., Gymnich, T., Gerlach, A.R., Edelman, A., Barbastathis, G., Braatz, R.D., Rackauckas, C.: Automated translation and accelerated solving of differential equations on multiple gpu platforms. *Computer Methods in Applied Mechanics and Engineering* **419**, 116591 (2024) <https://doi.org/10.1016/j.cma.2023.116591>
- [37] Byrne, S., Wilcox, L.C., Churavy, V.: Mpi.jl: Julia bindings for the message passing interface. *Proceedings of the JuliaCon Conferences* **1**(1), 68 (2021) <https://doi.org/10.21105/jcon.00068>
- [38] Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-passing Interface vol. 1. MIT press, ??? (1999)
- [39] Regier, J., Fischer, K., Pamnany, K., Noack, A., Revels, J., Lam, M., Howard, S., Giordano, R., Schlegel, D., McAuliffe, J., Thomas, R., Prabhat: Cataloging the visible universe through bayesian inference in julia at petascale. *Journal of Parallel and Distributed Computing* **127**, 89–104 (2019) <https://doi.org/10.1016/j.jpdc.2018.12.008>
- [40] Godoy, W.F., Valero-Lara, P., Anderson, C., Lee, K.W., Gainaru, A., Ferreira Da Silva, R., Vetter, J.S.: Julia as a unifying end-to-end workflow language on the frontier exascale system. In: *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W '23*, pp. 1989–1999. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3624062.3624278>
- [41] Giordano, M., Klöwer, M., Churavy, V.: Productivity meets Performance: Julia on A64FX. In: *IEEE International Conference on Cluster Computing*, pp. 549–555 (2022). <https://doi.org/10.1109/CLUSTER51413.2022.00072>

- [42] Hunold, S., Steiner, S.: 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), pp. 20–25 (2020). <https://doi.org/10.1109/PMBS51919.2020.00008>
- [43] Ramadhan, A., Wagner, G.L., Hill, C., Campin, J.-M., Churavy, V., Besard, T., Souza, A., Edelman, A., Ferrari, R., Marshall, J.: Oceananigans.jl: Fast and friendly geophysical fluid dynamics on gpus. *Journal of Open Source Software* **5**(53), 2018 (2020) <https://doi.org/10.21105/joss.02018>
- [44] Ranocha, H., Schlottke-Lakemper, M., Winters, A.R., Faulhaber, E., Chan, J., Gassner, G.: Adaptive numerical simulations with Trixi.jl: A case study of Julia for scientific computing. *Proceedings of the JuliaCon Conferences* **1**(1), 77 (2022) <https://doi.org/10.21105/jcon.00077> 2108.06476
- [45] Schlottke-Lakemper, M., Winters, A.R., Ranocha, H., Gassner, G.J.: A purely hyperbolic discontinuous Galerkin approach for self-gravitating gas dynamics. *Journal of Computational Physics* **442**, 110467 (2021) <https://doi.org/10.1016/j.jcp.2021.110467> 2008.10593
- [46] Marras, S., Tissaoui, Y., Wang, H., Stechmann, S.N.: Jexpresso v0.1.0: A Julia-language, user-friendly, multi-physics parallel solver for the solution of conservation laws on CPUs and GPUs. In: *The Proceedings of the 36th Parallel CFD International Conference 2025 on Parallel Computational Fluid Dynamics* (2025)
- [47] Shang, H., Shen, L., Fan, Y., Xu, Z., Guo, C., Liu, J., Zhou, W., Ma, H., Lin, R., Yang, Y., Li, F., Wang, Z., Zhang, Y., Li, Z.: Large-scale simulation of quantum computational chemistry on a new sunway supercomputer. In: *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14 (2022). <https://doi.org/10.1109/SC41404.2022.00019>
- [48] Martorell, P.A., Badia, S.: Stlcutters.jl: A scalable geometrical framework library for unfitted finite element discretisations. *Computer Physics Communications* **309**, 109479 (2025) <https://doi.org/10.1016/j.cpc.2024.109479>
- [49] Wegert, Z.J., Manyer, J., Mallon, C.N., Badia, S., Challis, V.J.: Gridaptopopt.jl: a scalable julia toolbox for level set-based topology optimisation. *Structural and Multidisciplinary Optimization* **68**(1), 22 (2025)
- [50] de la Calle, E., García, C.: Evaluation of Juliana Tool: A translator for Julia’s CUDA.jl code into KernelAbstraction.jl. *Future Generation Computer Systems*, 107813 (2025) <https://doi.org/10.1016/j.future.2025.107813>
- [51] Briand, L., Jourden, H., Pérache, M.: Julia versus c++ kokkos for performance portable cartesian cfd solvers on heterogeneous architectures. *The International Journal of High Performance Computing Applications* **39**(4), 481–501 (2025)

<https://doi.org/10.1177/10943420251341179>

- [52] Godoy, W.F., Valero-Lara, P., Dettling, T.E., Trefftz, C., Jorquera, I., Sheehy, T., Miller, R.G., Gonzalez-Tallada, M., Vetter, J.S., Churavy, V.: Evaluating performance and portability of high-level programming models: Julia, python/numba, and kokkos on exascale nodes. In: 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 373–382 (2023). <https://doi.org/10.1109/IPDPSW59300.2023.00068>
- [53] Lin, W.-C., McIntosh-Smith, S.: Comparing Julia to Performance Portable Parallel Programming Models for HPC. In: International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pp. 94–105 (2021). <https://doi.org/10.1109/PMBS54543.2021.00016>
- [54] Schanen, M., Maldonado, D.A., Pacaud, F., Montoisson, A., Anitescu, M., Kim, K., Kim, Y., Rao, V., Subramanyam, A.: Julia as a portable high-level language for numerical solvers of power flow equations on gpu architectures. *Les Cahiers du GERAD* ISSN **711**, 2440 (2020)
- [55] Bernholdt, D.E., Boehm, S., Bosilca, G., Gorentla Venkata, M., Grant, R.E., Naughton, T., Pritchard, H.P., Schulz, M., Vallee, G.R.: A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience* **32**(3), 4851 (2020) <https://doi.org/10.1002/cpe.4851> <https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4851>. e4851 cpe.4851
- [56] Valero-Lara, P., Godoy, W.F., Teranishi, K., Vetter, J.S.: JACC.shared: Leveraging HPC Metaprogramming and Performance Portability for Computations That Use Shared Memory GPUs. In: 2024 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2024). <https://doi.org/10.1109/HPEC62836.2024.10938453>
- [57] Valero-Lara, P., Godoy, W.F., Fackler, P.W., Teranishi, K., Vetter, J.S.: Enabling Scientific Applications with Performance-Portability and High-Productivity for Multi-GPU Programming with JACC.Multi. In: 2025 IEEE International Conference on eScience (eScience), pp. 240–249 (2025). <https://doi.org/10.1109/eScience65000.2025.00036>
- [58] Besard, T., De Sutter, B., Informatiesystemen., U.F.I.: Abstracties voor het programmeren van grafische processoren in hoogniveau-programmeertalen. Proefschrift ingediend tot het behalen van de graad van Doctor in de Ingenieurswetenschappen: Computerwetenschappen (2019)
- [59] Besard, T., Churavy, V., Samaroo, J., Moses, W., Baraldi, G., Guinard, C., Kadowaki, S., Schaub, S.D., Gymnich, T., Aluthge, D., Carlsson, K., Wilcox, L.C., Giordano, M., Fischer, K., Sukera, Byrne, S., Smirnov, A., Wennberg, D., Briand, L., Schanen, M., pchintalapudi, collinwarner, Zentrik, Wang, W.,

- Arakaki, T., Kelly, S., Surace, S.C.: JuliaGPU/GPUCompiler.jl:v2.2.1. <https://doi.org/10.5281/zenodo.21983960> . <https://doi.org/10.5281/zenodo.21983960>
- [60] Lehmann, M.: FluidX3D. <https://github.com/ProjectPhysX/FluidX3D>
- [61] Tissaoui, Y., Kelly, J.F., Marras, S.: Efficient spectral element method for the euler equations on unbounded domains. *Applied Mathematics and Computation* **487**, 129080 (2025) <https://doi.org/10.1016/j.amc.2024.129080>
- [62] Hahn, S.E., Fackler, P.W., Godoy, W.F., Maheshwari, K., Morgan, Z., Savici, A.T., Hoffmann, C.M., Valero-Lara, P., Vetter, J.S., Silva, R.: Integrating ORNL’s HPC and Neutron Facilities with a Performance-Portable CPU/GPU Ecosystem. In: *Proceedings of the SC ’24 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis. SC-W ’24*, pp. 2107–2117 (2025). <https://doi.org/10.1109/SCW63240.2024.00264>
- [63] RIKEN: Basic Design Technical Report for the New Flagship System “FugakuNEXT”. Technical report, RIKEN Center for Computational Science (2026). https://www.r-ccs.riken.jp/fugaku-next/fugakunext_basicdesign-technicalreport_en.pdf
- [64] Nagai, Y., Tomiya, A., Ohno, H.: Lattice Gauge Theory via LLVM-Level Automatic Differentiation (2026). <https://arxiv.org/abs/2602.20516>
- [65] Moses, W., Churavy, V.: Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H. (eds.) *Advances in Neural Information Processing Systems*, vol. 33, pp. 12472–12485 (2020)
- [66] Nagai, Y., Tomiya, A.: JuliaQCD: Portable lattice QCD package in Julia language (2024) [arXiv:2409.03030](https://arxiv.org/abs/2409.03030) [hep-lat]
- [67] Dahlbom, D., Zhang, H., Miles, C., Quinn, S., Niraula, A., Thiye, B., Wilson, M., Matin, S., Mankad, H., Hahn, S., Pajerowski, D., Johnston, S., Wang, Z., Lane, H., Li, Y.W., Bai, X., Mourigal, M., Batista, C.D., Barros, K.: Sunny.jl: A julia package for spin dynamics. *Journal of Open Source Software* **10**(116), 8138 (2025) <https://doi.org/10.21105/joss.08138>
- [68] Shor, P.W.: Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing* **26**(5), 1484–1509 (1997) <https://doi.org/10.1137/s0097539795293172>
- [69] Grover, L.K.: A fast quantum mechanical algorithm for database search (1996). <https://arxiv.org/abs/quant-ph/9605043>
- [70] Harrow, A.W., Hassidim, A., Lloyd, S.: Quantum algorithm for linear systems of equations. *Physical Review Letters* **103**(15) (2009) <https://doi.org/10.1103/>

- [71] Gidney, C.: How to factor 2048 bit RSA integers with less than a million noisy qubits (2025). <https://arxiv.org/abs/2505.15917>
- [72] Preskill, J.: Quantum Computing in the NISQ era and beyond. *Quantum* **2**, 79 (2018) <https://doi.org/10.22331/q-2018-08-06-79>
- [73] Javadi-Abhari, A., Treinish, M., Krsulich, K., Wood, C.J., Lishman, J., Gacon, J., Martiel, S., Nation, P.D., Bishop, L.S., Cross, A.W., Johnson, B.R., Gambetta, J.M.: Quantum computing with Qiskit (2024). <https://arxiv.org/abs/2405.08810>
- [74] Sivarajah, S., Dilkes, S., Cowtan, A., Simmons, W., Edgington, A., Duncan, R.: $t|ket\rangle$: a retargetable compiler for NISQ devices. *Quantum Science and Technology* **6**(1), 014003 (2020) <https://doi.org/10.1088/2058-9565/ab8e92>
- [75] Akibue, S., Kato, G., Tani, S.: Probabilistic unitary synthesis with optimal accuracy. *ACM Transactions on Quantum Computing* **5**(3) (2024) <https://doi.org/10.1145/3663576>
- [76] Hietala, K., Rand, R., Hung, S.-H., Wu, X., Hicks, M.: A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages* **5**(POPL), 1–29 (2021) <https://doi.org/10.1145/3434318>
- [77] King, A.D., Suzuki, S., Raymond, J., Zucca, A., Lanting, T., Altomare, F., Berkley, A.J., Ejtemaee, S., Hoskinson, E., Huang, S., Ladizinsky, E., MacDonald, A.J.R., Marsden, G., Oh, T., Poulin-Lamarre, G., Reis, M., Rich, C., Sato, Y., Whittaker, J.D., Yao, J., Harris, R., Lidar, D.A., Nishimori, H., Amin, M.H.: Coherent quantum annealing in a programmable 2,000 qubit Ising chain. *Nature Physics* **18**(11), 1324–1328 (2022) <https://doi.org/10.1038/s41567-022-01741-6>
- [78] Shaydulin, R., Li, C., Chakrabarti, S., DeCross, M., Herman, D., Kumar, N., Larson, J., Lykov, D., Minssen, P., Sun, Y., Alexeev, Y., Dreiling, J.M., Gaebler, J.P., Gatterman, T.M., Gerber, J.A., Gilmore, K., Gresh, D., Hewitt, N., Horst, C.V., Hu, S., Johansen, J., Matheny, M., Mengle, T., Mills, M., Moses, S.A., Neyenhuis, B., Siegfried, P., Yalovetzky, R., Pistoia, M.: Evidence of scaling advantage for the quantum approximate optimization algorithm on a classically intractable problem. *Science Advances* **10**(22) (2024) <https://doi.org/10.1126/sciadv.adm6761>
- [79] Pelofske, E., Rams, M.M., Bärtschi, A., Czarnik, P., Braccia, P., Cincio, L., Eidenbenz, S.: Evaluating the limits of quantum approximate optimization algorithm parameter transfer at high rounds on sparse ising models with geometrically local cubic terms. *Physical Review Research* **8**(2) (2026) <https://doi.org/10.1103/p2lg-z4kn>

- [80] Pelofske, E., Bärttschi, A., Cincio, L., Golden, J., Eidenbenz, S.: Scaling whole-chip QAOA for higher-order ising spin glass models on heavy-hex graphs. *npj Quantum Information* **10**(1) (2024) <https://doi.org/10.1038/s41534-024-00906-w>
- [81] King, A.D., Nocera, A., Rams, M.M., Dziarmaga, J., Wiersema, R., Bernoudy, W., Raymond, J., Kaushal, N., Heinsdorf, N., Harris, R., Boothby, K., Altomare, F., Asad, M., Berkley, A.J., Boschnak, M., Chern, K., Christiani, H., Cibere, S., Connor, J., Dehn, M.H., Deshpande, R., Ejtemaee, S., Farre, P., Hamer, K., Hoskinson, E., Huang, S., Johnson, M.W., Kortas, S., Ladizinsky, E., Lanting, T., Lai, T., Li, R., MacDonald, A.J.R., Marsden, G., McGeoch, C.C., Molavi, R., Oh, T., Neufeld, R., Norouzpour, M., Pasvolsky, J., Poitras, P., Poulin-Lamarre, G., Prescott, T., Reis, M., Rich, C., Samani, M., Sheldan, B., Smirnov, A., Sterpka, E., Trullas Clavera, B., Tsai, N., Volkmann, M., Whiticar, A.M., Whittaker, J.D., Wilkinson, W., Yao, J., Yi, T.J., Sandvik, A.W., Alvarez, G., Melko, R.G., Carrasquilla, J., Franz, M., Amin, M.H.: Beyond-classical computation in quantum simulation. *Science* **388**(6743), 199–204 (2025) <https://doi.org/10.1126/science.ad6285>
- [82] Deshpande, R., Kheirhah, M., Rich, C., Harris, R., Raymond, J., Hoskinson, E., Sathe, P., Berkley, A.J., Paul, S., Barch, B., Lidar, D.A., Müller, M., Aeppli, G., King, A.D., Amin, M.H.: Analog-Digital Quantum Computing with Quantum Annealing Processors (2026). <https://arxiv.org/abs/2603.15534>
- [83] Lee, Y.-T., Kumaran, K., Pokharel, B., Scheie, A., Sarkis, C.L., Tennant, D.A., Humble, T., Schleife, A., Kandala, A., Banerjee, A.: Benchmarking quantum simulation with neutron-scattering experiments (2026). <https://arxiv.org/abs/2603.15608>
- [84] DeCross, M., Haghshenas, R., Liu, M., Rinaldi, E., Gray, J., Alexeev, Y., Baldwin, C.H., Bartolotta, J.P., Bohn, M., Chertkov, E., Cline, J., Colina, J., DelVento, D., Dreiling, J.M., Foltz, C., Gaebler, J.P., Gatterman, T.M., Gilbreth, C.N., Giles, J., Gresh, D., Hall, A., Hankin, A., Hansen, A., Hewitt, N., Hoffman, I., Holliman, C., Hutson, R.B., Jacobs, T., Johansen, J., Lee, P.J., Lehman, E., Lucchetti, D., Lykov, D., Madjarov, I.S., Mathewson, B., Mayer, K., Mills, M., Niroula, P., Pino, J.M., Roman, C., Schechter, M., Siegfried, P.E., Tiemann, B.G., Volin, C., Walker, J., Shaydulin, R., Pistoia, M., Moses, S.A., Hayes, D., Neyenhuis, B., Stutz, R.P., Foss-Feig, M.: Computational power of random quantum circuits in arbitrary geometries. *Phys. Rev. X* **15**, 021052 (2025) <https://doi.org/10.1103/PhysRevX.15.021052>
- [85] Martiel, S., Chung, J.-U., Seif, A., Ghosh, S., Hincks, I., Deshpande, A., Feferman, B., Gambetta, J.M., Javadi-Abhari, A.: Sampling hard circuits with verifiably high fidelity (2026). <https://arxiv.org/abs/2607.25941>
- [86] Kim, Y., Eddins, A., Anand, S., Wei, K.X., Van Den Berg, E., Rosenblatt, S.,

- Nayfeh, H., Wu, Y., Zaletel, M., Temme, K., *et al.*: Evidence for the utility of quantum computing before fault tolerance. *Nature* **618**(7965), 500–505 (2023)
- [87] Tindall, J., Mello, A.F., Fishman, M., Stoudenmire, E.M., Sels, D.: Dynamics of disordered quantum systems with two- and three-dimensional tensor networks. *Science* **392**(6800), 868–872 (2026) <https://doi.org/10.1126/science.adx2728>
- [88] Begušić, T., Gray, J., Chan, G.K.-L.: Fast and converged classical simulations of evidence for the utility of quantum computing before fault tolerance. *Science Advances* **10**(3) (2024) <https://doi.org/10.1126/sciadv.adk4321>
- [89] Maunon, L., Carleo, G.: Challenging the Quantum Advantage Frontier with Large-Scale Classical Simulations of Annealing Dynamics (2025). <https://arxiv.org/abs/2503.08247>
- [90] Oh, C., Jiang, L., Fefferman, B.: Spoofing cross-entropy measure in boson sampling. *Phys. Rev. Lett.* **131**, 010401 (2023) <https://doi.org/10.1103/PhysRevLett.131.010401>
- [91] Manabe, H., Gu, H., Pan, F.: Classical Simulation and Design Frontiers for IBM’s Doped Clifford Sampling Experiment (2026). <https://arxiv.org/abs/2608.13110>
- [92] Tindall, J., Fishman, M., Stoudenmire, E.M., Sels, D.: Efficient Tensor Network Simulation of IBM’s Eagle Kicked Ising Experiment. *PRX Quantum* **5**, 010308 (2024) <https://doi.org/10.1103/PRXQuantum.5.010308>
- [93] Cross, A.W., Bishop, L.S., Sheldon, S., Nation, P.D., Gambetta, J.M.: Validating quantum computers using randomized model circuits. *Physical Review A* **100**(3) (2019) <https://doi.org/10.1103/physreva.100.032328>
- [94] Baldwin, C.H., Mayer, K., Brown, N.C., Ryan-Anderson, C., Hayes, D.: Re-examining the quantum volume test: Ideal distributions, compiler optimizations, confidence intervals, and scalable resource estimations. *Quantum* **6**, 707 (2022) <https://doi.org/10.22331/q-2022-05-09-707>
- [95] Pelofske, E., Bäertschi, A., Eidenbenz, S.: Quantum Volume in Practice: What Users Can Expect From NISQ Devices. *IEEE Transactions on Quantum Engineering* **3**, 1–19 (2022) <https://doi.org/10.1109/tqe.2022.3184764>
- [96] Moses, S.A., Baldwin, C.H., Allman, M.S., Ancona, R., Ascarrunz, L., Barnes, C., Bartolotta, J., Bjork, B., Blanchard, P., Bohn, M., Bohnet, J.G., Brown, N.C., Burdick, N.Q., Burton, W.C., Campbell, S.L., Campora, J.P., Carron, C., Chambers, J., Chan, J.W., Chen, Y.H., Chernoguzov, A., Chertkov, E., Colina, J., Curtis, J.P., Daniel, R., DeCross, M., Deen, D., Delaney, C., Dreiling, J.M., Ertsgaard, C.T., Esposito, J., Estey, B., Fabrikant, M., Figgatt, C., Foltz, C., Foss-Feig, M., Francois, D., Gaebler, J.P., Gatterman, T.M., Gilbreth, C.N.,

- Giles, J., Glynn, E., Hall, A., Hankin, A.M., Hansen, A., Hayes, D., Higashi, B., Hoffman, I.M., Horning, B., Hout, J.J., Jacobs, R., Johansen, J., Jones, L., Karcz, J., Klein, T., Lauria, P., Lee, P., Liefer, D., Lu, S.T., Lucchetti, D., Lytle, C., Malm, A., Matheny, M., Mathewson, B., Mayer, K., Miller, D.B., Mills, M., Neyenhuis, B., Nugent, L., Olson, S., Parks, J., Price, G.N., Price, Z., Pugh, M., Ransford, A., Reed, A.P., Roman, C., Rowe, M., Ryan-Anderson, C., Sanders, S., Sedlacek, J., Shevchuk, P., Siegfried, P., Skripka, T., Spaun, B., Sprenkle, R.T., Stutz, R.P., Swallows, M., Tobey, R.I., Tran, A., Tran, T., Vogt, E., Volin, C., Walker, J., Zolot, A.M., Pino, J.M.: A race-track trapped-ion quantum processor. *Phys. Rev. X* **13**, 041052 (2023) <https://doi.org/10.1103/PhysRevX.13.041052>
- [97] Berry, D.W., Tong, Y., Khattar, T., White, A., Kim, T.I., Low, G.H., Boixo, S., Ding, Z., Lin, L., Lee, S., Chan, G.K.-L., Babbush, R., Rubin, N.C.: Rapid Initial-State Preparation for the Quantum Simulation of Strongly Correlated Molecules. *PRX Quantum* **6**, 020327 (2025) <https://doi.org/10.1103/PRXQuantum.6.020327>
- [98] Foss-Feig, M., Hayes, D., Dreiling, J.M., Figgatt, C., Gaebler, J.P., Moses, S.A., Pino, J.M., Potter, A.C.: Holographic quantum algorithms for simulating correlated spin systems. *Physical Review Research* **3**(3) (2021) <https://doi.org/10.1103/physrevresearch.3.033002>
- [99] Gibbs, J., Cincio, L.: Deep Circuit Compression for Quantum Dynamics via Tensor Networks. *Quantum* **9**, 1789 (2025) <https://doi.org/10.22331/q-2025-07-09-1789>
- [100] Gibbs, J., Cincio, L., Sarma, C., Holmes, Z., Stevenson, P.: Low T -count preparation of nuclear eigenstates with tensor networks (2026). <https://arxiv.org/abs/2603.11156>
- [101] Gibbs, J., Cincio, L.: Learning Circuits with Infinite Tensor Networks (2025). <https://arxiv.org/abs/2506.02105>
- [102] Kim, J., Lee, S., Johnston, B., Vetter, J.S.: Iris: A performance-portable framework for cross-platform heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems* (2024)
- [103] Johnston, B., Miniskar, N.R., Young, A., Monil, M.A.H., Lee, S., Vetter, J.S.: Iris: Exploring performance scaling of the intelligent runtime system and its dynamic scheduling policies. In: 2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 58–67 (2024). IEEE
- [104] Miniskar, N.R., Lee, S., Beau, J., Young, A., Monil, M.A.H., Valero-Lara, P., Vetter, J.S.: Iris reimaged: Advancements in intelligent runtime system for task-based programming. In: Workshop on Asynchronous Many-Task Systems

and Applications, pp. 46–58 (2024). Springer

- [105] Miniskar, N.R., Monil, M.A.H., Valero-Lara, P., Liu, F.Y., Vetter, J.S.: Irisdmem: Efficient memory management for heterogeneous computing. In: 2023 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7 (2023). IEEE
- [106] Chen, L., Ahmed, N.K., Dutta, A., Bhattacharjee, A., Yu, S., Mahmud, Q.I., Abebe, W., Phan, H., Sarkar, A., Butler, B., Hasabnis, N., Oren, G., Vo, V.A., Munoz, J.P., Willke, T.L., Mattson, T., Jannesari, A.: The Landscape and Challenges of HPC Research and LLMs (2024). <https://arxiv.org/abs/2402.02018>
- [107] Teranishi, K., Menon, H., Godoy, W.F., Balaprakash, P., Bau, D., Ben-Nun, T., Bhatele, A., Franchetti, F., Franusich, M., Gamblin, T., Georgakoudis, G., Goldstein, T., Guha, A., Hahn, S.E., Iancu, C., Jin, Z., Jones, T., Low, T.-M., Mankad, H., Miniskar, N.R., Monil, M.A.H., Nichols, D., Parasyris, K., Pophale, S., Valero-Lara, P., Vetter, J.S., Williams, S., Young, A.: Leveraging ai for productive and trustworthy hpc software: Challenges and research directions. In: Neuwirth, S., Paul, A.K., Weinzierl, T., Carson, E.C. (eds.) High Performance Computing, pp. 615–625. Springer, Cham (2026)
- [108] Nichols, D., Davis, J.H., Xie, Z., Rajaram, A., Bhatele, A.: Can Large Language Models Write Parallel Code? Association for Computing Machinery, New York, NY, USA (2024)
- [109] Cui, B., Ramesh, T., Hernandez, O., Zhou, K.: Comprehensive evaluation of llms in hpc code performance optimization. In: Workshop Proceedings of the 54th International Conference on Parallel Processing. ICPP Workshops ’25, pp. 1–8. Association for Computing Machinery, New York, NY, USA (2025). <https://doi.org/10.1145/3750720.3757280> . <https://doi.org/10.1145/3750720.3757280>
- [110] Valero-Lara, P., Godoy, W.F., Teranishi, K., Balaprakash, P., Vetter, J.S.: ChatBLAS: The First AI-Generated and Portable BLAS Library. In: SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 19–24 (2024). <https://doi.org/10.1109/SCW63240.2024.00010>
- [111] Godoy, W., Valero-Lara, P., Teranishi, K., Balaprakash, P., Vetter, J.: Evaluation of OpenAI Codex for HPC Parallel Programming Models Kernel Generation. In: Proceedings of the 52nd International Conference on Parallel Processing Workshops. ICPP Workshops ’23, pp. 136–144. Association for Computing Machinery, New York, NY, USA (2023). <https://doi.org/10.1145/3605731.3605886> . <https://doi.org/10.1145/3605731.3605886>
- [112] Valero-Lara, P., Huante, A., Al Lail, M., Godoy, W.F., Teranishi, K., Balaprakash, P., Vetter, J.S.: Comparing Llama-2 and GPT-3 LLMs for HPC

Kernels Generation. In: Languages and Compilers for Parallel Computing: 36th International Workshop, LCPC 2023, Lexington, KY, USA, October 11–13, 2023, Revised Selected Papers, pp. 20–32. Springer, Berlin, Heidelberg (2023). https://doi.org/10.1007/978-3-032-02436-7_2

- [113] Godoy, W.F., Valero-Lara, P., Teranishi, K., Balaprakash, P., Vetter, J.S.: Large language model evaluation for high-performance computing software development. *Concurrency and Computation: Practice and Experience* **36**(26), 8269 (2024) <https://doi.org/10.1002/cpe.8269> <https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.8269>

Appendix A JACC Software Artifacts

Repository	https://github.com/JuliaGPU/JACC.jl
Current version (08/2026)	v1.3.1
License	MIT license
Language	Julia (v1.11 minimum)
Package Stats	https://juliapkgstats.com/pkg/JACC
User documentation	https://juliagpu.github.io/JACC.jl/stable
OpenSSF best practices badge	https://www.bestpractices.dev/en/projects/12117/passing
Examples and Tutorials	https://github.com/JuliaORNL/JACC-applications