JACC: Leveraging HPC Meta-Programming and Performance Portability with the Just-in-Time and LLVM-based Julia Language

Pedro Valero-Lara, William F. Godoy, Het Mankad, Keita Teranishi, and Jeffrey S. Vetter
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
{valerolarap},{godoywf},
{mankadhy},{teranishik},{vetter}@ornl.gov

Johannes Blaschke
Lawrence Berkeley National Laboratory
Berkeley, California, USA
jpblaschke@lbl.gov

Michel Schanen
Argonne National Laboratory
Lemont, Illinois, USA
mschanen@anl.gov

***Abstract*—We present JACC (Julia for ACCelerators), the first high-level, and performance-portable model for the just-in-time and LLVM-based Julia language. JACC provides a unified and lightweight front end across different back ends available in Julia, enabling the same Julia code to run efficiently on many HPC CPU and GPU targets. We evaluated the performance of JACC for common HPC kernels as well as for the most computationally demanding kernels used in applications, HPCCG, a supercomputing benchmark test for sparse domains, and HARVEY, a blood flow simulator to assist in the diagnosis and treatment of patients suffering from vascular diseases. We carried out the performance analysis on the most advanced US DOE supercomputers: Aurora, Frontier, and Perlmutter. Overall, we show that JACC has a negligible overhead versus vendor-specific solutions, reporting GPU speedups with no extra cost to programmability.**

***Index Terms*—Julia, Metaprogramming, Performance Portability, Programming Productivity, GPU Acceleration**

I. Introduction

Generic, high-level, metaprogramming-oriented techniques allow programmers to focus on the particular structure of an application while offloading target-specific code specializations to compiler back ends. This enables a single source code to be portable while enhancing development productivity for a variety of heterogeneous architectures. Representative parallel computing models that use this technique can be found in high-performance computing (HPC) languages like C++: RAJA [1], Kokkos [2], SYCL [3], OpenMP/OpenACC [4]. These models rely on highly optimized vendor back ends (e.g., CUDA, HIP, oneAPI), and their performance portability and overhead trade-offs have become an active area of research [5].

There is renewed interest in closing the gaps in high-level, just-in-time dynamic languages such as Python [6], Julia [7], and R [8] and their current performance levels, especially because these languages and their ecosystems have been at the forefront of AI/ML and data analysis workflows [9]. Additionally, the widespread adoption of LLVM [10] allows for unifying efforts in the compiler back-end layers for several languages and programming models. Julia and other efforts that target Python (e.g., Numba [11], codon [12]) reuse LLVM's modularity by generating intermediate representations (LLVM-IRs) to achieve improved performance from their high-level, just-in-time, and dynamic programming models.

In this work, we focus on both programming productivity and performance portability and evaluate how to bring that capacity into just-in-time solutions such as Julia. For that, we implemented the new JACC (Julia for ACCelerators) Julia package, the first and only high-level meta-programming and performance portable model in Julia. JACC's main contributions are:

1) Taking meta-programming and performance portable capabilities to the just-in-time and real-time interactive Julia ecosystem for the very first time, elevating the capabilities of programming productivity for the implementation of Julia scientific and HPC codes.
2) JACC sits atop existing package back ends leveraging different vendor solutions (e.g., CUDA for NVIDIA GPUs, AMDGPU for AMD GPUs, OneAPI for Intel GPUs, and Julia's Base.Threads based on pthreads for CPUs).
3) JACC provides a single and unified high-level and easy-to-use front end that abstracts the burden of low-level hardware and software details away from the programmers.
4) JACC is a portable and highly productive programming solution for Julia application developers who target a large variety of CPUs and GPUs, including AMD EPYC 7742 Rome CPUs, AMD MI100 GPUs, NVIDIA Ampere A100 GPUs, and Intel Data Center Max 1550 GPUs.
5) We evaluated the performance of JACC for common high-performance computing (HPC) kernels (e.g., DOT and AXPY BLAS Level-1 operations) as well as for the most computationally demanding kernels used in important applications, such as the Conjugate Gradient

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00245

algorithm used in MiniFE, a proxy application for unstructured implicit finite element codes, and in HPCCG, a supercomputing benchmark test for sparse domains, or the lattice-Boltzmann method used in HARVEY, a blood flow simulator to assist on the diagnosis and treatment of patients suffering from vascular diseases.

II. Julia in HPC

Julia was created to provide a unified programming language, community, and an integrated ecosystem (packaging, testing, software tools, AI) to enhance productivity while providing performance mechanisms that rely on LLVM advancements, addressing the main weaknesses of current HPC programming languages as outline in a recent community paper [13]. Julia uses the LLVM framework for JIT compilation, enabling the same runtime speed as other compiled languages such as *C*. Julia is also compatible with any external library implemented in *Python*, *Fortran*, and *C*. Similar to Python, Julia's syntax is simple and efficient and users interact either through passing source code files as arguments to the `julia` command, or optionally via its real-time interactive Read Eval Print Loop (REPL) command line to easily add commands, scripts, and packages.

Julia offers several advantages:

- Julia syntax is optimized for mathematics and scientific environments similar to the formulas used by domain-specific experts.
- JIT compilation on top of LLVM enables Julia to outperform other high-level languages (e.g. Python, R, Matlab) in terms of speed.
- Its native support for AI, makes Julia a real asset for HPC-AI integration.
- Community and integrated ecosystem invested and motivated by performance and productivity.

The support of Julia for HPC, although not as mature as in other languages, is already significant. The Julia ecosystem provides support for parallel computation on CPUs using Base.Threads, a Julia package implemented in *pthreads* on top of LLVM, which allows the distribution of the computation on different CPU cores by using decorators on top of loops (similar to *OpenMP* and *OpenACC*). Julia supports GPU[1] accelerator programming natively thanks to vendor's packages, such as CUDA.jl, AMDGPU.jl and OneAPI.jl. Other packages, such as Distributed.jl and MPI.jl [14], allow Julia codes to run on distributed memory environments.

Julia is not different from other programming languages in facing performance portability challenges. Currently, Julia's programming models tend to follow closely vendor layers which could still be too low-level hindering programming productivity. JACC addresses this challenge for Julia programmers and applications, providing an HPC portable and highly productive model targeting current HPC CPU, GPU hardware, which could potentially be extended to other architectures (AI custom hardware, FPGAs), and configurations (distributed memory, multi-device, etc.).

III. JACC Description

The JACC model is divided into two main components: memory and compute (Figure 2). These components have different implementations, with one per back end supported (Figure 1). We implemented four back ends so far on top of Base.Threads, CUDA, AMDGPU, and OneAPI to target CPUs, NVIDIA GPUs, AMD GPUs, and Intel GPUs, respectively.

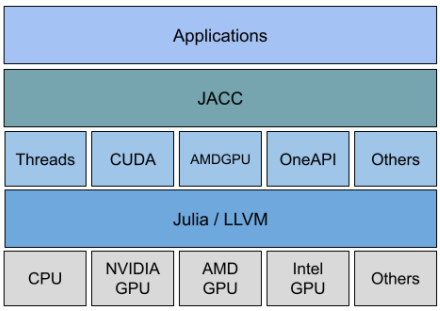

Fig. 1. JACC model illustrating its lightweight nature on top of LLVM for performance portable code.

Owing to the dynamic and just-in-time nature of the Julia language, JACC differs from other existing metaprogramming solutions in how the back end is chosen [1], [2]. We use Julia's Preferences[2] package, which generates the `LocalPreferences.toml` file before precompilation to store the preferences (back end) used for JACC. Additionally, JACC leverages the recently introduced package extensions in Julia v1.9 to allow for optional package dependencies or `weakdependencies`. Therefore, vendor-specific back-end implementations (e.g., CUDA, AMDGPU, OneAPI) inside JACC can coexist through function overloading and multiple dispatches without incurring additional costs when installing JACC. The default back end is Julia's Base.Threads implementation, which targets CPUs.

The memory management in JACC is transparent to the programmer, and we use a very similar syntax to that used in other Julia packages: `JACC.Array`. `JACC.Array` is mapped on the equivalent Julia function depending on the target back end. Notably, when using Base.Threads as the back end, using `JACC.Array` is not necessary.

As depicted in Figure 2, JACC has two primary constructs: `parallel_for` and `parallel_reduce`, and this is similar to metaprogramming solutions in other languages [1], [2]. We also included two different variants to be chosen based on the data layout used: unidimensional or multidimensional (up to three dimensions). These constructs are composed of three main components: (1) the number of iterations of the for-loop or reduction, which is typically equal to the size of the arrays; (2) the name of the function that defines the operations to be

[1]https://juliagpu.org/

[2]https://github.com/JuliaPackaging/Preferences.jl

```
# Unidimensional arrays
function axpy(i, alpha, x, y)
  x[i] += alpha * y[i]
end
function dot(i, x, y)
  return x[i] * y[i]
end
SIZE = 1_000_000
x = round.(rand(Float64, SIZE) * 100)
y = round.(rand(Float64, SIZE) * 100)
alpha = 2.5
dx = JACC.Array(x)
dy = JACC.Array(y)
JACC.parallel_for(SIZE, axpy, alpha, dx, dy)
res = JACC.parallel_reduce(SIZE, dot, dx, dy)
# Multidimensional arrays
function axpy(i, j, alpha, x, y)
  x[i,j] = x[i,j] + alpha * y[i,j]
end
function dot(i, j, x, y)
  return x[i,j] * y[i,j]
end
SIZE = 1_000
x = round.(rand(Float64, SIZE, SIZE) * 100)
y = round.(rand(Float64, SIZE, SIZE) * 100)
alpha = 2.5
dx = JACC.Array(x)
dy = JACC.Array(y)
JACC.parallel_for((SIZE,SIZE),axpy,alpha,dx,dy)
res = JACC.parallel_reduce((SIZE,SIZE),dot,dx,dy)
```

Fig. 2. JACC front end example.

computed in each iteration of the loop; and (3) the parameters used in the function. Another important difference between JACC and other existing meta-programming solutions for other programming languages, such as Kokkos, is that the function, which defines the operations to be computed in every iteration of the loop, has to be implemented separately and in advance of the call of `parallel_for` or `parallel_reduce`.

To better illustrate the differences between the model implemented in JACC versus other Julia packages, Figure 3 depict an implementation of the level-1 BLAS DOT operations using the CUDA Julia syntax. For clarity and lack of space, we had to reduce the CUDA code, which is much larger than the one illustrated.

As shown, JACC provides a very simple way to parallelize codes by providing a unified front end that can be deployed on top of other Julia packages and make use of different architectures. When using JACC, programmers do not have to burden themselves with low-level details at the hardware or software levels, and this abstraction provides a high-level and portable solution to make Julia a productive programming solution for HPC and scientific software.

A. *Differences with KernelAbstractions*

KernelsAbstractions (KA) (Figure 4) is a Julia package that enables writing GPU-like kernels to target different execution back ends. KA is intended to be a minimal, high-performance library that guides ways to write heterogeneous code. However, unlike KA, the proposed JACC programming model (1) avoids setting the coarse or fine granularity

```
function dot_cuda_kernel(SIZE, ret, x, y)
 shared_mem = @cuDynamicSharedMem(Float64, 512)
 i = ( blockIdx().x - 1) * blockDim().x + threadIdx
     ().x
 ti = threadIdx().x tmp::Float64 = 0.0
 shared_mem[threadIdx().x] = 0.0
 if i <= SIZE
  tmp = @inbounds x[i] * y[i]
  shared_mem[threadIdx().x] = tmp
 end
 sync_threads()
 if (ti <= 256)
  shared_mem[ti] += shared_mem[ti+256]
 end
 sync_threads()
 ...
 return nothing
end
function reduce_kernel(SIZE, red, ret)
 shared_mem = @cuDynamicSharedMem(Float64, 512)
 i = ( blockIdx().x - 1) * blockDim().x + threadIdx
     ().x
 ii = i
 tmp::Float64 = 0.0
 if SIZE > 512
  while ii <= SIZE
   tmp += @inbounds red[ii]
   ii += 512
  end
 ...
 return nothing
end
function dot_cuda(SIZE,x,y)
 threads = min(SIZE,512)
 blocks = ceil(Int,SIZE/threads)
 ret = CUDA.zeros(Float64,blocks)
 rret = CUDA.zeros(Float64,1)
 @cuda threads=threads blocks=blocks shmem=512*
     sizeof(Float64) dot_cuda_kernel(SIZE,ret,x,y)
 @cuda ... reduce_kernel(blocks,ret,rret)
 return rret
end
SIZE = 1_000_000
x = round.(rand(Float64, SIZE) * 100)
dx = CuArray(x)
y = round.(rand(Float64, SIZE) * 100)
dy = CuArray(y)
res = dot_cuda(SIZE, dx, dy)
```

Fig. 3. CUDA 1D DOT code.

based on the target back end (CPU or GPU); (2) unifies the non-portable, vendor-specific memory allocation functionality (e.g., `Array`, `CuArray`, `ROCArray`, `oneArray`) into a high-level portable `JACCArray`; and (3) provides a high-level descriptive solution that hides any low-level hardware/software detail.

KA provides portability at the cost of a more demanding and complex syntax. For example, KA requires setting the granularity based on the back end to be used. Unlike KA, JACC provides a high-level, transparent, portable, and simple-to-use syntax that abstracts all low-level details of hardware/-software specialization. These qualities make JACC a portable and highly productive programming solution for Julia codes.

```
@kernel function axpy_ka_kernel(alpha, x, y)
    i = @index(Global)
    x[i] += alpha * y[i]
end
function axpy_ka(SIZE, alpha, x, y)
  backend = get_backend(x)
  @assert get_backend(y) == backend
  groupsize = KernelAbstractions.isgpu(backend) ?
      256 : 1024
  kernel! = axpy_ka_kernel(backend, groupsize)
  kernel!(alpha, x, y, ndrange=SIZE)
  KernelAbstractions.synchronize(backend)
end
SIZE = 1_000_000
backend = CPU/GPU
x = round.(rand(allocate(backend, Float64, SIZE)) *
    100)
y = round.(rand(allocate(backend, Float64, SIZE)) *
    100)
alpha = 2.5
axpy_ka(SIZE, alpha, x, y)
```

Fig. 4. KernelsAbstraction AXPY code.

IV. JACC Implementation

As with other high-level solutions that run atop multiple back ends, a different implementation is required for each of these back ends to leverage the respective JACC front-end functions (Figure 1). However, the very nature of the high-level Julia ecosystem and programming model makes this implementation much simpler than other solutions that run on top of other languages [2], [15], [3], [16].

```
#JACC.Array and JACC.parallel_for on top of Threads
function __init__()
  const JACC.Array = Base.Array{T,N} where {T,N}
end
#Unidimensional
function parallel_for(N::I, f::F, x...) where {I<:
    Integer,F<:Function}
  Threads.@sync Threads.@threads for i in 1:N
    f(i, x...)
  end
end
#Multidimensional
function parallel_for((M, N)::Tuple{I,I}, f::F, x
    ...) where {I<:Integer,F<:Function}
  Threads.@sync Threads.@threads for j in 1:N
    for i in 1:M
      f(i, j, x...)
    end
  end
end
```

Fig. 5. `JACC.Array` and `JACC.parallel_for` implementations on top of Julia Threads.

To better describe the implementation of the JACC front-end functions, Figures 5 and 7 include pseudocode for the `JACC.Array` and `JACC.parallel_for` implementations on top of Base.Threads and CUDA, respectively. For simplicity, we focus on the CUDA and Base.Threads back ends and `JACC.parallel_for` implementations. However, the reader can access the implementation of `JACC.parallel_reduce` construct and other back ends

```
#JACC.Array and JACC.parallel_for on top of CUDA
function __init__()
  const JACC.Array = CUDA.CuArray{T,N} where {T,N}
end
#Unidimensional
function _parallel_for_cuda(f, x...)
  i = ( blockIdx().x - 1) * blockDim().x + threadIdx
      ().x
  f(i, x...)
  return nothing
end
function JACC.parallel_for(N::I, f::F, x...) where {
    I<:Integer,F<:Function}
  maxPossibleThreads = attribute(device(),CUDA.
      DEVICE_ATTRIBUTE_MAX_BLOCK_DIM_X)
  cuda_threads = min(N, maxPossibleThreads)
  cuda_blocks = ceil(Int, N/cuda_threads)
  CUDA.@sync @cuda threads=cuda_threads blocks=
      cuda_blocks _parallel_for_cuda(N, f, x...)
end
#Multidimensional
function _parallel_for_cuda_MN(f,x...)
  i = ( blockIdx().x - 1) * blockDim().x + threadIdx
      ().x
  j = ( blockIdx().y - 1) * blockDim().y + threadIdx
      ().y
  f(i, j, x...)
  return nothing
end
function JACC.parallel_for((M, N)::Tuple{I,I}, f::F,
     x...) where {I<:Integer,F<:Function}
  numThreads = 16
  Mthreads = min(M, numThreads)
  Nthreads = min(N, numThreads)
  Mblocks = ceil(Int, M/Mthreads)
  Nblocks = ceil(Int, N/Nthreads)
  CUDA.@sync @cuda threads=(Mthreads, Nthreads)
      blocks=(Mblocks, Nblocks)
      _parallel_for_cuda_MN(f, x...)
end
```

Fig. 6. `JACC.Array` and `JACC.parallel_for` implementations on top of CUDA.

(e.g., AMDGPU, OneAPI) via the public JACC repository. [3] More details can be found in Appendix A.

As shown, `JACC.Array` is a wrapper of the corresponding `Array` constructs implemented in the different back ends: `Base.Array` for Base.Threads and `CuArray` for CUDA. JACC adapts (in a transparent way) memory management and granularity to fit the characteristics of the hardware and the syntax of the models provided for that hardware. This enables us to exploit coarse granularity for CPU architectures while exploiting a fine-grain approach for GPU architectures. This also affects the way memory is accessed. Multidimensional arrays in Julia are stored in column-major order; therefore, the coarse-grain parallelization or decomposition for the Base.Threads back end must be carried out in column-wise order. For GPUs, we exploit a fine-grain approach in which every thread will access one single position of the array(s) involved in the computation. The objective is to force consecutive GPU threads to access consecutive memory locations (i.e., coalescing memory access). This enables us

[3] https://github.com/JuliaORNL/JACC.jl

```
#JACC.Array and JACC.parallel_for on top of OneAPI
function __init__()
    const JACC.Array = oneAPI.oneArray{T, N} where {
        T, N}
end
#Unidimensional
function _parallel_for_oneapi(f, x...)
  i = get_global_id()
  f(i, x...)
  return nothing
end
function JACC.parallel_for(N::I, f::F, x...) where {
    I <: Integer, F <: Function}
    maxPossibleItems = oneAPI.oneL0.
        compute_properties(device().
        maxTotalGroupSize)
    items = min(N, maxPossibleItems)
    groups = ceil(Int, N / items)
    oneAPI.@sync @oneapi items = items groups =
        groups _parallel_for_oneapi(f, x...)
end
#Multidimensional
function _parallel_for_oneapi_MN(f,x...)
  j = get_global_id(0)
  i = get_global_id(1)
  f(i, j, x...)
  return nothing
end
function JACC.parallel_for((M, N)::Tuple{I, I}, f::F
    , x...) where {I <: Integer, F <: Function}
    maxPossibleItems = 16
    Mitems = min(M, maxPossibleItems)
    Nitems = min(N, maxPossibleItems)
    Mgroups = ceil(Int, M / Mitems)
    Ngroups = ceil(Int, N / Nitems)
    oneAPI.@sync @oneapi items=(Mitems, Nitems)
        groups=(Mgroups, Ngroups)
        _parallel_for_oneapi_MN(f, x...)
end
```

Fig. 7. `JACC.Array` and `JACC.parallel_for` implementations on top of CUDA.

to alleviate the typically high latency of the high-bandwidth GPU memories to achieve the expected high performance that accelerators can provide.

Notably, JACC is a synchronous API, so it is guaranteed that the computation is finished after the invocation of any JACC construct.

V. Performance Analysis

In this section, we want to evaluate (i) the capability of JACC as a high-level, meta-programming and performance portable model for transparent GPU parallelization, and (ii) the potential overhead of JACC as a software layer on top of Julia vendors packages. Previously these authors evaluated the performance of Julia against other languages or models such as Kokkos, Python, OpenMP, HIP, or CUDA on multiple HPC configurations, concluding that Julia is competitive or even better in terms of performance than other vendor-specific or open-source models [17].

This section is composed of a set of well-known and widely-used test cases and applications for HPC that we used to evaluate the performance of JACC on four representative HPC architectures hosted in the most advanced US DOE

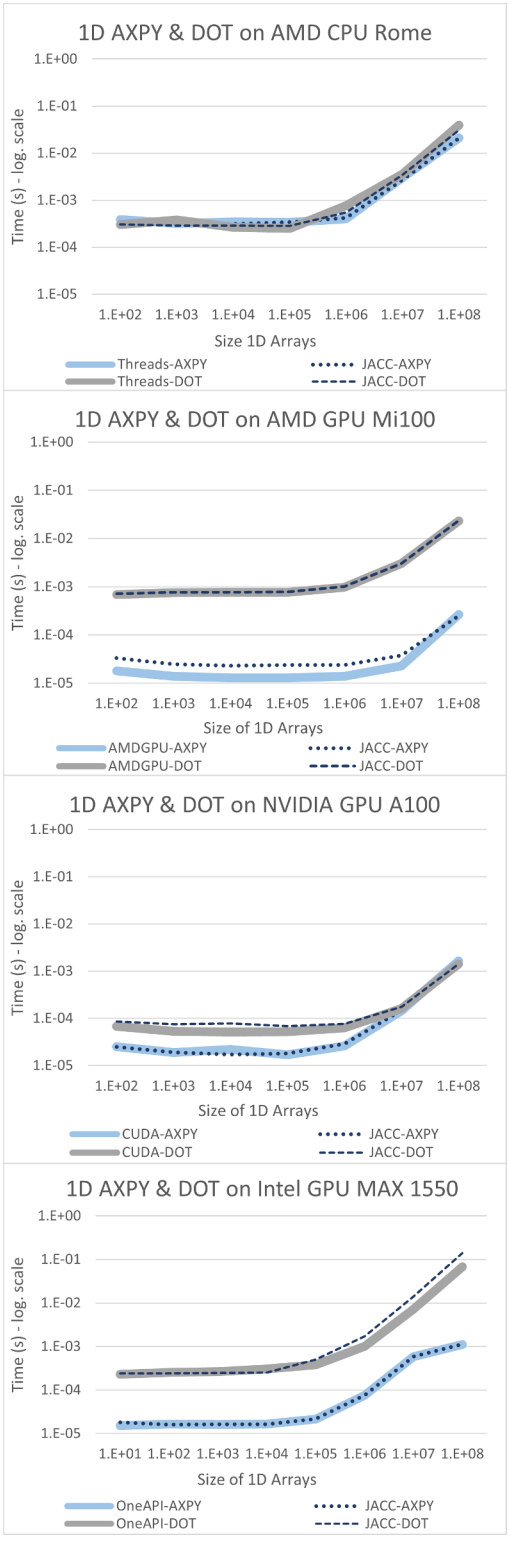

Fig. 8. The 1D AXPY and DOT time on one AMD Rome CPU, one AMD Mi100 GPU, one NVIDIA Ampere A100 GPU, and one Intel MAX 1550 GPU using device-specific and JACC models.

supercomputers, such as Frontier, Aurora, or Perlmutter: an AMD EPYC 7742 Rome CPU (64 cores), AMD Mi100 GPU, NVIDIA Ampere A100 GPU, and an Intel MAX 1550 GPU. The analysis focuses on evaluating the performance portability of JACC by comparing its performance against that of the device-specific models. In other words, we want to know if the performance reached by JACC on top of Base.Threads, CUDA, AMDGPU, and OneAPI is equivalent or not to codes that use the corresponding device-specific syntax, i.e., if there is an overhead when using JACC. Other details, including the comparison of the performance trends on the four different architectures and the GPU acceleration reached versus CPU times, are also provided. All computations are double-precision operations. For JACC code evaluation, we used the same JACC codes on all four architectures.

A. AXPY and DOT BLAS Operations

We use AXPY and DOT operations for our first test case. The first operation performs a scalar-vector product and adds the result to a vector. The second operation performs a dot product between two vectors. These are well-known level-1 BLAS operations used in multiple applications and benchmarks [18], [19]. For the JACC implementations, we use the `parallel_for` construct to implement the AXPY operation and the `parallel_reduce` construct for the DOT implementation (see Figure 2).

1) 1D Arrays: We begin by analyzing the performance of 1D arrays. Figure 8 shows the performance on both device-specific codes (Base.Threads, AMDGPU, CUDA, OneAPI) and JACC codes. We see very similar trends in performance for both AXPY and DOT test cases on the AMD CPU. We do not see significant differences between the performance achieved by the device-specific codes that use the Base.Threads Julia package for this case versus the JACC codes running on top of the same package (Figure 5).

On the AMD GPU, we see a clear difference between AXPY and DOT performance. This is mainly because of the two separate kernels for DOT operations on GPUs (Figure 3) adding extra overhead in terms of latency and CPU-GPU communication [20], [16]. Once again, we do not see a significant difference between the performance of device-specific codes and JACC codes. The exception is the AXPY operation, for which JACC codes are slower than the device-specific codes on small- to medium-sized arrays. However, the JACC codes provide similar performance for the AXPY operation on computations that use large arrays.

On the NVIDIA GPU, we see a trend that is similar to the results for the AMD GPU. However, the gap between AXPY and DOT performance is smaller than for the AMD GPU, and the gap is minimal when computing large vectors. This is because of a faster CPU-GPU connection, which helps mitigates latency. Once again, we see similar performance for CUDA and JACC codes. Unlike with the AMD GPU, we do not see any overhead for JACC codes when running AXPY test cases on small- and medium-sized arrays. However, we see a small overhead when running DOT test cases on small- and medium-sized arrays.

Finally, on the Intel GPU, we again see a difference between the execution times of computing AXPY and DOT. Although very similar to the AMD GPU's results, the gap in time between the two operations is slightly smaller on the Intel GPU than on the AMD GPU. In this case, we do not see any overhead of JACC codes when running AXPY and achieve very similar performance to the codes implemented using the OneAPI model. However, we see some overhead for JACC codes when running the DOT operation, especially on larger vector sizes, and this overhead is about 35%.

When comparing CPU and GPU performance (e.g., AMD CPU versus AMD GPU), we note that GPUs provide better performance for AXPY operations, but this is not the case for DOT computations. For DOT, the CPU provides better performance than GPUs for small- and medium-sized arrays because of the particular requirements to compute this operation on GPUs (Figures 3). Our findings are consistent with another study that used the C++ metaprogramming Kokkos model [20].

In terms of portability provided by JACC, we note that the same JACC code (AXPY) running on the AMD GPU can reach a speedup of about $70\times$ versus the same code running on the AMD CPU. However, for DOT products on small arrays (which favor CPUs), the same JACC code can reach a speedup of $2\times$ on the AMD CPU versus the AMD GPU.

2) 2D Arrays: Next, we extend this analysis to 2D arrays. For that, we use JACC's multidimensional API (Figure 2). As with the analysis of 1D arrays, we see a very similar performance trend for AXPY and DOT computations for device-specific codes and JACC codes on the AMD CPU.

In general, we see that the gap in performance between AXPY and DOT computations is reduced in all GPUs. Also, the overhead observed in the previous analysis of 1D arrays for JACC codes is mostly absent here, except for the NVIDIA GPU, which exhibits some overhead for JACC codes when computing AXPY because of more allocations computed by the JACC code. Although in most cases the number of allocations is similar for both code types (i.e., device-specific and JACC), there are slightly more allocations in the JACC code due to the metaprogramming nature of this approach (functions are managed as one more parameter). Note that the JACC-AXPY line overlaps with the JACC-DOT line in the NVIDIA A100 GPU graph (Figure 9). Although similar overlapping is shown for the Intel GPU, this happens on both JACC and CUDA codes. Also, on the Intel GPU, the time consumed by DOT operations is about $2\times$ longer than the time consumed by AXPY operations (note that we are using log. scale for time). This is not the case for DOT computations, in which we see very similar performance for both codes (JACC and device-specific codes) and architectures (NVIDIA GPU and Intel GPU). Further analysis is required to identify the source of the performance overhead in AXPY operations on 2D arrays.

In this case (2D computations), GPUs provide better per-

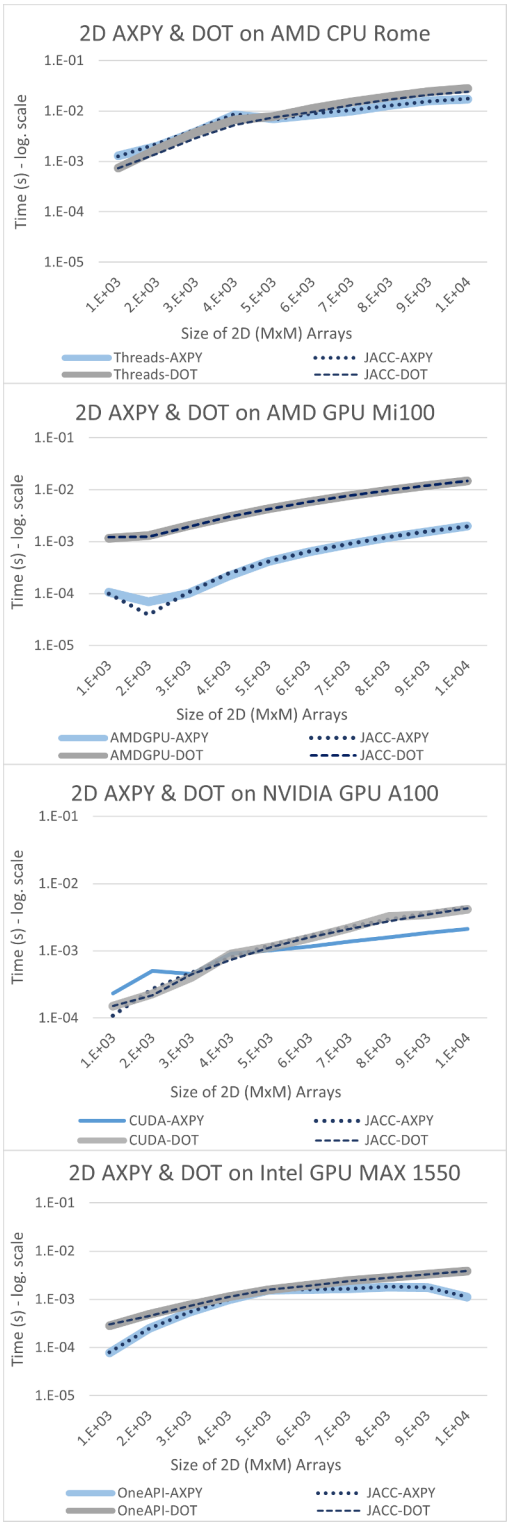

Fig. 9. The 2D AXPY and DOT time on one AMD Rome CPU, one AMD Mi100 GPU, one NVIDIA Ampere A100 GPU, and one Intel MAX 1550 GPU using device-specific and JACC models.

formance than CPUs in most cases for DOT computations. In general, by increasing the computational demand and using 2D arrays instead of 1D arrays, JACC codes can achieve performance comparable to device-specific codes in most of the test cases.

B. HARVEY: Lattice-Boltzmann Method

The lattice-Boltzmann method (LBM) is a widely used method for Computational Fluid Dynamics (CFD) simulations [21]. In particular, it is very popular for low Reynolds number or laminar flow simulations, such as the ones used by the HARVEY simulator [22]. LBM is a key component of HARVEY, a massively parallel CFD code to study the mechanisms driving disease development to inform treatment planning and improve clinical care on the diagnosis and treatment of patients suffering from vascular disease. LBM [23], [24] is an explicit Navier-Stokes solver for weakly compressible flows with lattice-symmetry characteristics and is used for the conservation of the macroscopic moments [25], [26].

LBM does this by modeling the fluid as a distribution function of microscopic particles. These dual microscopic and macroscopic aspects are key features of the mesoscopic method and have significant implications for LBM regularization schemes. The evolution of particle distribution f is formulated as equilibrium and non-equilibrium terms by using the lattice-Boltzmann equation,

$$f_i(\mathbf{x}+\mathbf{c}_i, t+1) = f_i^{eq}(\mathbf{x},t) + \left(1-\frac{1}{\tau}\right) f_i^{neq}(\mathbf{x},t), \quad (1)$$

for relaxation time τ, lattice site $\mathbf{x}$, and time step t. The lattice Boltzmann equation is best understood as the combination of two steps: collision and streaming. In the collision step, local inter-particle interaction leads to a relaxation toward the equilibrium distribution, which is computed as $\mathrm{f}_i^*(\mathbf{x},t) = f_i^{eq}(\mathbf{x},t) + \left(1-\frac{1}{\tau}\right) f_i^{neq}(\mathbf{x},t)$ for post-collision distribution f^*. During streaming, conversely, post-collision distribution components advance along the lattice according to their discrete velocities: $\mathrm{f}_i(\mathbf{x}+\mathbf{c}_i, t+1) = f_i^*(\mathbf{x},t)$.

For this study, as we can see in Figure 10, we implemented the 2-lattice D2Q9 pull algorithm [27], [28], which is used by HARVEY. This approach is considered state-of-the-art and is the fastest algorithm for LBM on both CPUs and GPUs [29], [30]. To implement the LBM code using JACC, we used a single multidimensional `parallel_for` construct (see Figure 10).

As shown in Figure 11, JACC provides performance comparable to that of the device-specific models on the four architectures used in this study. The speedup provided by the GPU over the CPU is equivalent to the one provided by other metaprogramming solutions [20]. Unlike the AXPY test case on 2D arrays (multidimensional `parallel_for` JACC construct), where we see a relatively small performance overhead for JACC codes on the NVIDIA GPU, we do not see that here. Indeed, the performance results for device-specific codes and JACC codes are very similar. This is because of the higher computational demand of the LBM

```
# Implementation of the LBM 2DQ9 algorithm
function lbm( x, y, f, f1, f2, t, w, cx, cy, SIZE)
 u = 0.0
 v = 0.0
 p = 0.0
 x_stream = 0
 y_stream = 0
 if x > 1 && x < SIZE && y > 1 && y < SIZE
  for k in 1:9
   x_stream = x - cx[k]
   y_stream = y - cy[k]
   ind =  ( k - 1 ) * SIZE * SIZE + x * SIZE + y
  iind = ( k - 1 ) * SIZE * SIZE + x_stream * SIZE +
       y_stream
   f[trunc(Int,ind)] = f1[trunc(Int,iind)]
  end
  for k in 1:9
   ind =  ( k - 1 ) * SIZE * SIZE + x * SIZE + y
   p += f[ind]
   u += f[ind] * cx[k]
   v += f[ind] * cy[k]
  end
  u /= p
  v /= p
  for k in 1:9
   cu = cx[k] * u + cy[k] * v
   feq = w[k] * p * ( 1.0 + 3.0 * cu + cu * cu -
         1.5 * ( ( u * u ) + ( v * v ) ) )
   ind = ( k - 1 ) * SIZE * SIZE + x * SIZE + y
   f2[trunc(Int,ind)] = f[trunc(Int,ind)] * (1.0 -
       1.0 / t) + feq * 1 / t
  end
 end
end
# Initialization
SIZE = 1_000
#We assume that the variables are already
    initialized
df = JACC.Array(f)
df1 = JACC.Array(f1)
df2 = JACC.Array(f2)
dcx = JACC.Array(cx)
dcy = JACC.Array(cy)
dw = JACC.Array(w)
# Invocation of the Lattice-Boltzmann Method
parallel_for((SIZE, SIZE), lbm, df, df1, df2, t, dw,
     dcx, dcy, SIZE)
```

Fig. 10. JACC LBM code.

implementation. This is in agreement with our findings of the previous subsection; the higher the computational cost the lower the potential overhead of using JACC. When comparing the same JACC code on the AMD CPU versus the GPUs, we see that JACC can reach a speedup of about 14× on the AMD GPU, 20× on the NVIDIA GPU, and 6.5× on the Intel GPU.

C. MiniFE and HPCCG: Conjugate Gradient

Conjugate gradient (CG) is a well-known and widely used iterative method for solving sparse systems of linear equations. This algorithm is a key component in a large variety of HPC applications, such as MiniFE [31], a proxy application for unstructured implicit finite element codes. These systems appear in finite difference and finite element methods, PDEs, structural analysis, circuit analysis, and many more linear algebra–related problems [32]. Given the ubiquity and impor-

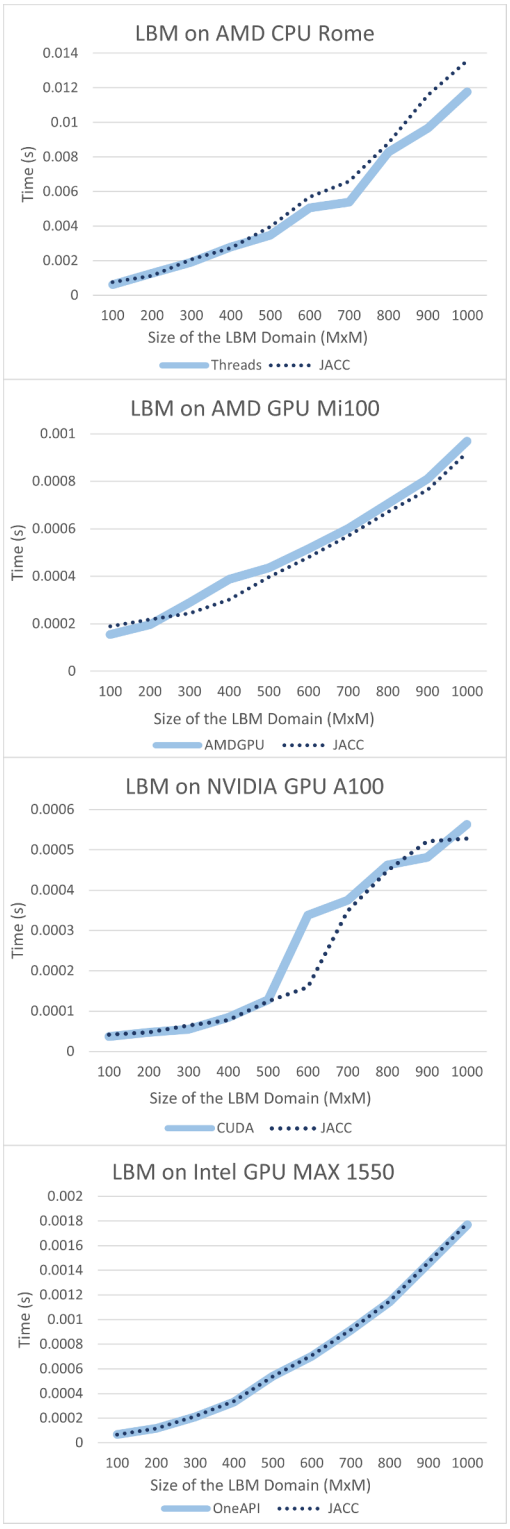

Fig. 11. LBM time on one AMD Rome CPU, one AMD Mi100 GPU, one NVIDIA Ampere A100 GPU, and one Intel MAX 1550 GPU using device-specific and JACC models.

tance of this operation in HPC applications, it is no surprise that it is also used to benchmark supercomputer performance (HPCCG)[4].

```
# Tridiagonal Matrix-Vector Multiplication
function matvecmul(i, a1, a2, a3, x, y, SIZE)
  if i == 1
    y[i] = a2[i] * x[i] + a1[i] * x[i+1]
  elseif i == SIZE
    y[i] = a3[i] * x[i-1] + a2[i] * x[i]
  else
    y[i] = a3[i] * x[i-1] + a1[i] * + x[i] + a1[i] *
        + x[i+1]
  end
end
# Conjugate Gradient Algorithm
function cg(SIZE, a0, a1, a2, r, p, s, x, r_old,
    r_aux )
  cond = 1.0
  while cond <= 1e-12
    r_old = copy(r)
    JACC.parallel_for(SIZE, matvecmul, a0, a1, a2, p
        , s, SIZE)
    alpha0 = JACC.parallel_reduce(SIZE, dot, r, r)
    alpha1 = JACC.parallel_reduce(SIZE, dot, p, s)
    alpha = alpha0 / alpha1
    negative_alpha = alpha * (-1.0)
    JACC.parallel_for(SIZE, axpy, negative_alpha, r,
        s)
    JACC.parallel_for(SIZE, axpy, alpha, x, p)
    beta0 = JACC.parallel_reduce(SIZE, dot, r, r)
    beta1 = JACC.parallel_reduce(SIZE, dot, r_old,
        r_old)
    beta = beta0 / beta1
    r_aux = copy(r)
    JACC.parallel_for(SIZE, axpy, beta, r_aux, p)
    cond = JACC.parallel_reduce(SIZE, dot, r, r)
    p = copy(r_aux)
  end
end
# Initialization
SIZE = 100_000_000
a0 = ones(SIZE) a1 = a1 * 4.0 da0 = JACC.Array(a0)
a1 = ones(SIZE) da1 = JACC.Array(a1)
a2 = ones(SIZE) da2 = JACC.Array(a2)
r = ones(SIZE) r = r * 0.5 dr = JACC.Array(r)
p = ones(SIZE) p = p * 0.5 dp = JACC.Array(p)
s = zeros(SIZE) ds = JACC.Array(s)
x = zeros(SIZE) dx = JACC.Array(x)
r_old = zeros(SIZE) dr_old = JACC.Array(r_old)
r_aux = zeros(SIZE) dr_aux = JACC.Array(r_aux)
# Invocation of the Conjugate Gradient Algorithm
cg(SIZE, a0, a1, a2, r, p, s, x, r_old, r_aux)
```

Fig. 12. JACC CG code.

For this study, we implemented the plain CG algorithm [18], [19] without a precondition (see Figure 12), as the one used in MiniFE or other libraries such as the HPCCG benchmark. This simplifies the study of the optimizations thanks to the elimination of the preconditioning step. To this end, we generate a diagonal-dominant tridiagonal sparse matrix (see `matvelmul` in Figure 12), which is commonly used in these contexts (e.g., in the MiniFE application and the HPCCG benchmark). The JACC implementation of CG is composed of multiple unidimensional `parallel_for` and `parallel_reduce` invocations to compute the different steps of the algorithm, which consists of a series of DOT and AXPY computations, among other kind of operations.

[4]http://www.hpcg-benchmark.org/

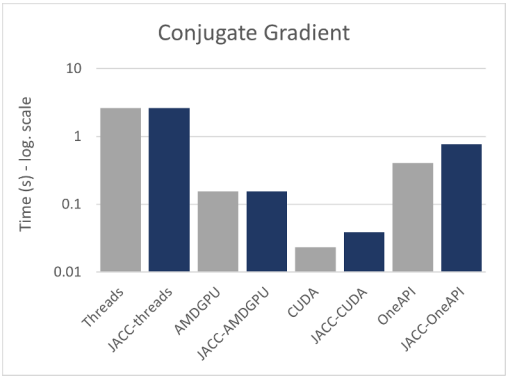

Fig. 13. CG time on one AMD Rome CPU, one AMD Mi100 GPU, one NVIDIA Ampere A100 GPU, and one Intel GPU using both device-specific and JACC models.

Figure 13 shows the execution time of computing one iteration of the CG algorithm on a tridiagonal matrix size equal to 100M. As shown, we see similar performance on both device-specific codes and JACC codes. Only in the Intel GPU results do we see some overhead for running JACC, and this finding agrees with the previous results for the 1D AXPY and DOT test cases. We observe significant speedup using JACC over the AMD CPU when comparing the execution times for the AMD GPU ($17\times$), the NVIDIA GPU ($68\times$), and the Intel GPU ($4\times$).

VI. Related Work

Julia's HPC ecosystem has been an active area of exploration and community engagement in recent years [13]. For example, Ranocha et al. [33] assessed their Trixi hyperbolic partial differential equation (PDE) solver at scale providing similar performance to traditional HPC languages. More recent work by Godoy et al. [9] evaluated Julia as a unified end-to-end language for HPC workflows on Frontier (#1 supercomputer on the Top500 list and they found good scalability on up to 4K GPUs (512 nodes). Shang et al. [34] conducted quantum computational chemistry simulations achieving up to 91% efficiency when measuring weak scaling on up to 21M cores of the Sunway supercomputer. Lin and McIntosh-Smith [35] used memory and compute-bound mini apps to show that Julia's performance is on par or slightly behind traditional compiled languages across several CPU/GPU HPC hardware configurations. Godoy et al. [36] compared the performance of Julia codes against other high-level solutions such as those based on pragmas (OpenMP and OpenACC) or metaprogramming (Kokkos), among others vendor-specific solutions (CUDA and HIP) concluding that Julia is very competitive or even faster than other solution for HPC targets. Faingnaert et al. [37] provided optimized GEMM kernels in Julia that are competitive with cuBLAS and CUTLASS implementations. Giordano et al. [38] found competitive system performance

for Julia's Message Passing Interface (MPI) [39], MPI.jl, on the Fujitsu A64FX Arm-based Fugaku system. Overall, Julia is promising due to its compiled nature leveraging advances in LLVM.

VII. Conclusions and Future Directions

We present JACC, the first and the only high-level and metaprogramming programming model for performance portable codes in the just-in-time and LLVM-based Julia language. JACC provides a unified front end with a near-zero overhead when compared to CPU and GPU device-specific models available in the Julia ecosystem. The descriptive nature of JACC allows the compiler to perform advanced optimizations for varying computing patterns and device types. Results show that JACC not only eliminates the burden of device specialization for Julia applications, thereby providing a real performance-portable and highly productive programming solution, but it also achieves significant acceleration for certain hardware and application characteristics.

In future work, we plan to extend this effort to other platforms. We will also explore novel features for a higher level of transparency and more efficient exploitation of available resources, including heterogeneous memory architectures and heterogeneous multi-device nodes.

Acknowledgment

This research used resources of the Oak Ridge Leadership Computing Facility and the Experimental Computing Laboratory (ExCL) at the Oak Ridge National Laboratory, which is supported by the Office of Science of the US Department of Energy under Contract No. DE-AC05-00OR22725. This work is funded, in part, by Bluestone, an X-Stack project in the DOE Advanced Scientific Computing Office with program manager Hal Finkel. This research was funded in part by the ASCR Stewardship for Programming Systems and Tools (S4PST) project, part of the Next Generation of Scientific Software Technologies (NGSST). This manuscript has been authored by UT-Battelle LLC under contract DE-AC05-00OR22725 with 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 (http://energy.gov/downloads/doe-public-access-plan).

References

[1] D. Beckingsale, R. D. Hornung, T. Scogland, and A. Vargas, "Performance portable C++ programming with RAJA," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 455–456. [Online]. Available: https://doi.org/10.1145/3293883.3302577

[2] C. Trott, L. Berger-Vergiat, D. Poliakoff, S. Rajamanickam, D. Lebrun-Grandié, J. Madsen, N. A. Awar, M. Gligoric, G. Shipman, and G. Womeldorff, "The kokkos ecosystem: Comprehensive performance portability for high performance computing," *Comput. Sci. Eng.*, vol. 23, no. 5, pp. 10–18, 2021. [Online]. Available: https://doi.org/10.1109/MCSE.2021.3098509

[3] Z. Jin and J. S. Vetter, "Performance portability study of epistasis detection using SYCL on NVIDIA GPU," in *BCB '22: 13th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics, Northbrook, Illinois, USA, August 7 - 10, 2022*. ACM, 2022, pp. 69:1–69:8. [Online]. Available: https://doi.org/10.1145/3535508.3545591

[4] B. Chapman, G. Jost, and R. Van Der Pas, "Using OpenMP," 2008.

[5] A. Marowka, "On the Performance Portability of OpenACC, OpenMP, Kokkos and RAJA," in *HPC Asia 2022: International Conference on High Performance Computing in Asia-Pacific Region, Virtual Event, Japan, January 12 - 14, 2022*. ACM, 2022, pp. 103–114. [Online]. Available: https://doi.org/10.1145/3492805.3492806

[6] G. Van Rossum *et al.*, "Python programming language." in *USENIX annual technical conference*, vol. 41, no. 1. Santa Clara, CA, 2007, pp. 1–36.

[7] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, Jan. 2017.

[8] R. Ihaka and R. Gentleman, "R: a language for data analysis and graphics," *Journal of computational and graphical statistics*, vol. 5, no. 3, pp. 299–314, 1996.

[9] W. F. Godoy, P. Valero-Lara, C. Anderson, K. W. Lee, A. Gainaru, R. F. da Silva, and J. S. Vetter, "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 2023, Denver, CO, USA, November 12-17, 2023*. ACM, 2023, pp. 1989–1999. [Online]. Available: https://doi.org/10.1145/3624062.3624278

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[11] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A LLVM-based Python JIT compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, 2015, pp. 1–6.

[12] A. Shajii, G. Ramirez, H. Smajlović, J. Ray, B. Berger, S. Amarasinghe, and I. Numanagić, "Codon: A compiler for high-performance pythonic applications and dsls," in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, ser. CC 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 191–202. [Online]. Available: https://doi.org/10.1145/3578360.3580275

[13] V. Churavy, W. F. Godoy, C. Bauer, H. Ranocha, M. Schlottke-Lakemper, L. Räss, J. Blaschke, M. Giordano, E. Schnetter, S. Omlin, J. S. Vetter, and A. Edelman, "Bridging HPC Communities through the Julia Programming Language," submitted for review, 2022.

[14] S. Byrne, L. C. Wilcox, and V. Churavy, "MPI.jl: Julia bindings for the Message Passing Interface," *Proceedings of the JuliaCon Conferences*, vol. 1, no. 1, p. 68, 2021. [Online]. Available: https://doi.org/10.21105/jcon.00068

[15] S. Yue and J. Gray, "SPOT: A DSL for extending fortran programs with metaprogramming," *Adv. Softw. Eng.*, vol. 2014, pp. 917 327:1–917 327:23, 2014. [Online]. Available: https://doi.org/10.1155/2014/917327

[16] P. Valero-Lara, S. Lee, M. G. Tallada, J. E. Denny, and J. S. Vetter, "KokkACC: Enhancing Kokkos with OpenACC," in *9th Workshop on Accelerator Programming Using Directives, WACCPD@SC 2022, Dallas, TX, USA, November 13-18, 2022*. IEEE, 2022, pp. 32–42. [Online]. Available: https://doi.org/10.1109/WACCPD56842.2022.00009

[17] W. F. Godoy, P. Valero-Lara, T. E. Dettling, C. Trefftz, I. Jorquera, T. Sheehy, R. G. Miller, M. G. Tallada, J. S. Vetter, and V. Churavy, "Evaluating performance and portability of high-level programming models: Julia, python/numba, and kokkos on exascale nodes," *CoRR*, vol. abs/2303.06195, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.06195

[18] L. Toledo, P. Valero-Lara, J. S. Vetter, and A. J. Peña, "Static graphs for coding productivity in openacc," in *28th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2021,*

Bengaluru, India, December 17-20, 2021. IEEE, 2021, pp. 364–369. [Online]. Available: https://doi.org/10.1109/HiPC53243.2021.00050

[19] S. Catalán, X. Martorell, J. Labarta, T. Usui, L. A. T. Díaz, and P. Valero-Lara, "Accelerating conjugate gradient using ompss," in *20th International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2019, Gold Coast, Australia, December 5-7, 2019*. IEEE, 2019, pp. 121–126. [Online]. Available: https://doi.org/10.1109/PDCAT46702.2019.00033

[20] P. Valero-Lara, S. Lee, J. E. Denny, K. Teranishi, J. S. Vetter, and M. G. Tallada, "skokkos: Enabling kokkos with transparent device selection on heterogeneous systems using openacc," in *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPCAsia 2024, Nagoya, Japan, January 25-27, 2024*. ACM, 2024, pp. 23–34. [Online]. Available: https://doi.org/10.1145/3635035.3635043

[21] Valero-Lara, Pedro, Martínez-Pérez, Ivan, Sirvent, Raül, Peña, Antonio J., Martorell, Xavier, and Labarta, Jesús, "Simulating the behavior of the human brain on gpus," *Oil Gas Sci. Technol. – Rev. IFP Energies nouvelles*, vol. 73, p. 63, 2018. [Online]. Available: https://doi.org/10.2516/ogst/2018061

[22] S. Roychowdhury, S. T. Mahmud, A. X. Martin, P. Balogh, D. F. Puleri, J. Gounley, E. W. Draeger, and A. Randles, "Enhancing adaptive physics refinement simulations through the addition of realistic red blood cell counts," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2023, Denver, CO, USA, November 12-17, 2023*, D. Arnold, R. M. Badia, and K. M. Mohror, Eds. ACM, 2023, pp. 41:1–41:13. [Online]. Available: https://doi.org/10.1145/3581784.3607105

[23] P. Valero-Lara, F. D. Igual, M. Prieto-Matías, A. Pinelli, and J. Favier, "Accelerating fluid-solid simulations (lattice-boltzmann & immersed-boundary) on heterogeneous architectures," *J. Comput. Sci.*, vol. 10, pp. 249–261, 2015. [Online]. Available: https://doi.org/10.1016/j.jocs.2015.07.002

[24] P. Valero-Lara and J. Jansson, "LBM-HPC - an open-source tool for fluid simulations. case study: Unified parallel C (UPC-PGAS)," in *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*. IEEE Computer Society, 2015, pp. 318–321. [Online]. Available: https://doi.org/10.1109/CLUSTER.2015.52

[25] Y.-H. Qian, D. d'Humières, and P. Lallemand, "Lattice BGK models for Navier-Stokes equation," *EPL (Europhysics Letters)*, vol. 17, no. 6, p. 479, 1992.

[26] X. He and L.-S. Luo, "A priori derivation of the lattice Boltzmann equation," *Physical Review E*, vol. 55, no. 6, p. R6333, 1997.

[27] P. Valero-Lara and J. Jansson, "Heterogeneous CPU+GPU approaches for mesh refinement over lattice-boltzmann simulations," *Concurr. Comput. Pract. Exp.*, vol. 29, no. 7, 2017. [Online]. Available: https://doi.org/10.1002/cpe.3919

[28] P. Valero-Lara, "Reducing memory requirements for large size LBM simulations on gpus," *Concurr. Comput. Pract. Exp.*, vol. 29, no. 24, 2017. [Online]. Available: https://doi.org/10.1002/cpe.4221

[29] J. Gounley, M. Vardhan, E. W. Draeger, P. Valero-Lara, S. V. Moore, and A. Randles, "Propagation pattern for moment representation of the lattice boltzmann method," *IEEE Trans. Parallel Distributed Syst.*, vol. 33, no. 3, pp. 642–653, 2022. [Online]. Available: https://doi.org/10.1109/TPDS.2021.3098456

[30] P. Valero-Lara, "Accelerating solid-fluid interaction based on the immersed boundary method on multicore and GPU architectures," *J. Supercomput.*, vol. 70, no. 2, pp. 799–815, 2014. [Online]. Available: https://doi.org/10.1007/s11227-014-1262-2

[31] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," https://github.com/Mantevo/, 2022, online accessed 20-April-2022.

[32] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," USA, Tech. Rep., 1994.

[33] H. Ranocha, M. Schlottke-Lakemper, A. R. Winters, E. Faulhaber, J. Chan, and G. J. Gassner, "Adaptive numerical simulations with Trixi.jl: A case study of Julia for scientific computing," *Proceedings of the JuliaCon Conferences*, vol. 1, no. 1, p. 77, 2022.

[34] H. Shang, L. Shen, Y. Fan, Z. Xu, C. Guo, J. Liu, W. Zhou, H. Ma, R. Lin, Y. Yang, F. Li, Z. Wang, Y. Zhang, and Z. Li, "Large-scale simulation of quantum computational chemistry on a new sunway supercomputer," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–14.

[35] W.-C. Lin and S. McIntosh-Smith, "Comparing Julia to Performance Portable Parallel Programming Models for HPC," in *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2021, pp. 94–105.

[36] W. F. Godoy, P. Valero-Lara, T. E. Dettling, C. Trefftz, I. Jorquera, T. Sheehy, R. G. Miller, M. G. Tallada, J. S. Vetter, and V. Churavy, "Evaluating performance and portability of high-level programming models: Julia, python/numba, and kokkos on exascale nodes," in *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023 - Workshops, St. Petersburg, FL, USA, May 15-19, 2023*. IEEE, 2023, pp. 373–382. [Online]. Available: https://doi.org/10.1109/IPDPSW59300.2023.00068

[37] T. Faingnaert, T. Besard, and B. De Sutter, "Flexible Performant GEMM Kernels on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 9, pp. 2230–2248, 2022.

[38] M. Giordano, M. Klöwer, and V. Churavy, "Productivity meets Performance: Julia on A64FX," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 2022, pp. 549–555.

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

APPENDIX

The code used for this study is hosted on GitHub: https://github.com/JuliaORNL/JACC.jl. It follows the typical structure of a Julia project, which includes packaging information in the `Project.toml` file (including compatibility) and the newly added `weakdependencies` feature for making CUDA.jl, AMDGPU.jl, and OneAPI.jl optional backend dependencies. JACC makes use of Julia's support for testing. Listing 2 contains the steps to test JACC from a Julia terminal.

Listing 1. Steps to run JACC.jl testing on NVIDIA systems.

```
# You need to be inside JACC.jl folder
using JACC
# Next you need to specify the backend to use (e.g., CUDA)
using CUDA
# Instantiate the project
] instantiate
# Run the tests
test
```

The `test` directory (in JACC.jl/test/) contains typical implementation examples and testing codes (one per backend). Continuous Integration (CI) is implemented by using GitHub Actions runners that check Julia's `weakdependencies` compilation and run the JACC testing in CPU (Base.Threads) and GPU (AMDGPU.jl, CUDA.jl, OneAPI.jl) modes. We plan to extend this testing to Intel GPU hardware in the near future.

Additionally, all the benchmarks used in the paper can be accessed via a public GitHub repository [5]. The steps to follow to run the different experiments are described in the README file of that repository:

Listing 2. Steps to run JACC benchmarks.

```
git clone git@github.com:pedrovalerolara/JACC-Test-Codes.git
# Access to the benchmark (one per backend)
cd JACC-Test-Codes/benchmarks/CUDA
julia --project=.
# Instantiate Environment
] dev ../.. JACC
# Execute Benchmarks
julia --project=. benchmark.jl >& cuda.txt
```

Finally, to run Julia on the Oak Ridge Leadership Computing Facility's systems (Summit and Frontier), the `scripts` directory (in Julia.jl/scripts/) provides examples for simple system-specific configurations such as the one presented in Listing 3 for the AMD-based Frontier system.

Listing 3. Configuration script to run JACC.jl on Frontier AMD systems.

```
#!/bin/bash

# Change these 3 lines accordingly
PROJ_DIR=/lustre/orion/proj-shared/csc383/$USER
export JULIA_DEPOT_PATH=$PROJ_DIR/julia_depot
# Location where you cloned JACC.jl
JACC_DIR=$PROJ_DIR/ProgrammingModels.jl/JACC.jl

# Remove existing generated files and existing modules
rm -f $JACC_DIR/Manifest.toml
rm -f $JACC_DIR/LocalPreferences.toml
module purge

# Load required modules
module load PrgEnv-cray/8.3.3 # has required gcc
module load rocm/5.4.0
module load julia # default is 1.9.0

# Required to point at underlying modules above
export JULIA_AMDGPU_DISABLE_ARTIFACTS=1

# Instantiate the project by installing packages in Project.toml
julia --project=$JACC_DIR -e 'using Pkg; Pkg.instantiate()'
```

[5] https://github.com/pedrovalerolara/JACC-Test-Codes