2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3503-1199-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPSW59300.2023.00068

Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes

William F. Godoy, Pedro Valero-Lara, T. Elise Dettling, Christian Trefftz, Ian Jorquera,
Thomas Sheehy, Ross G. Miller, Marc Gonzalez-Tallada, Jeffrey S. Vetter
Oak Ridge National Laboratory
{godoywf}, {valerolarap}, {dettlingte}, {trefftzci}, {jorqueraid}, {sheehytb}, {rgmiller}, {gonzaleztalm}, {vetter}@ornl.gov
Valentin Churavy
Massachusetts Institute of Technology
vchuravy@mit.edu

***Abstract*—We explore the performance and portability of the high-level programming models: the LLVM-based Julia and Python/Numba, and Kokkos on high-performance computing (HPC) nodes: AMD Epyc CPUs and MI250X graphical processing units (GPUs) on Frontier's test bed Crusher system and Ampere's Arm-based CPUs and NVIDIA's A100 GPUs on the Wombat system at the Oak Ridge Leadership Computing Facilities. We compare the default performance of a hand-rolled dense matrix multiplication algorithm on CPUs against vendor-compiled C/OpenMP implementations, and on each GPU against CUDA and HIP. Rather than focusing on the kernel optimization per-se, we select this naive approach to resemble exploratory work in science and as a lower-bound for performance to isolate the effect of each programming model. Julia and Kokkos perform comparably with C/OpenMP on CPUs, while Julia implementations are competitive with CUDA and HIP on GPUs. Performance gaps are identified on NVIDIA A100 GPUs for Julia's single precision and Kokkos, and for Python/Numba in all scenarios. We also comment on half-precision support, productivity, performance portability metrics, and platform readiness. We expect to contribute to the understanding and direction for high-level, high-productivity languages in HPC as the first-generation exascale systems are deployed.**

***Index Terms*—Julia, Python/Numba, Kokkos, OpenMP, LLVM, Performance, Portability, HPC, Exascale, GPU**

I. Introduction

High-level dynamic languages such as Python [1], Julia [2], and R [3] have been at the forefront of artificial intelligence/machine learning (AI/ML), data analysis, and interactive computing workflows in the last decade. Traditional high-performance computing (HPC) frameworks that power the underlying low-level computations for performance and scalability are written in compiled languages: C, C++, and Fortran. At the same time, parallel programming models written in these languages aim to address the increasing heterogeneity of the targeted HPC hardware, which is dominated by highly multithreaded CPUs and graphics processing units (GPUs) [4].

The emergence and adoption of LLVM [5] by major compiler vendors has led to unifying efforts to provide performance portable code across several languages and programming models. Julia and Python/Numba [6] reuse LLVM's modularity by generating intermediate representations (LLVM-IR) to achieve performance, from their high-level, dynamic programming models. Similarly, directive-based standard approaches (e.g., OpenMP [7], OpenACC [8]) provide a higher-level performance-portable model for HPC compiled languages, whereas metaprogramming approaches (e.g., Kokkos [9], Raja [10], [11], Thrust [12]) provides powerful portable interfaces that target C++ applications. These high-level models rely on highly optimized vendor back ends (e.g., OpenMP, CUDA [13], HIP [14]), and their performance portability and overhead trade-offs have become an active area of research [11]. Nevertheless, high-level programming models become an attractive alternative to the end-to-end codesign process, thereby making them vital to closing gaps in the convergence of AI/ML, data science, and HPC as more emphasis is placed on the performance, portability, and productivity of scientific workflows [15].

This work compares the performance, portability, and productivity of Julia, Python/Numba, and Kokkos high-level programming models for the CPU and GPU architectures that power upcoming exascale systems. We analyze single node scalability on two systems hosted at the Oak Ridge Leadership Computing Facility (OLCF)[1]—Wombat, which uses Arm Ampere Neoverse CPUs and 2 NVIDIA A100 GPUs, and Crusher, which is equipped with AMD EPYC 7A53 CPUs and 8 MI250X GPUs and serves as a test bed for Frontier, the first exascale system on the TOP500 list.[2] We run hand-rolled general matrix multiplication (GEMM) code for dense matrices using Julia, Python/Numba and Kokkos

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher acknowledges the US government license to provide public access under the DOE Public Access Plan (https://energy.gov/downloads/doe-public-access-plan).

[1] https://www.olcf.ornl.gov/

[2] https://www.top500.org/

979-8-3503-1199-0/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPSW59300.2023.00068

implementations and compare the performance with C for multithreaded CPU (OpenMP) and single GPU (CUDA/HIP) systems. GEMM is an important kernel in the Basic Linear Algebra Subprograms (BLAS) [16] used across several deep learning AI frameworks, for which modern GPU architectures have been heavily optimized via tensor cores [17]–[20]. The motivation for choosing a hand-rolled GEMM implementation is to i) isolate each programming model and environment in a simple kernel, and ii) to have a performance lower-bound point of reference that resembles custom scientific kernels in rapid prototyping formulations on dense matrices with many vector multiply and add operations. Results are presented for implementations of half- (when possible), single-, and double-precision floating point operations. We evaluate the productivity and performance portability of these high-level approaches by using a common metric to analyze the resulting code implementations.

The rest of the paper is organized as follows: Section II provides a summary of related efforts that have evaluated the performance and portability of these high-level programming models. Section III describes the numerical experiments conducted on the Crusher and Wombat nodes. Performance results and follow-up discussion are presented in Section IV, and the analysis of the performance portability is presented in Section V. Section VI summarizes the study. Description of the reproducible artifacts used in this study are provided in Appendix A.

II. Related Work

Recent efforts have attempted to understand the performance gaps between portable high-level programming models and their equivalent highly optimized, vendor-specific implementations. We classify this work according to the nature of the high-level implementation.

a) Dynamic Languages: Julia provides a dynamic, compiled frontend to LLVM targeting scientific computing and data science. Its use in HPC is still an area of active exploration and community engagement [21]. Ranocha et al. [22] present an assessment of their hyperbolic partial differential equation (PDE) solver at scale, Trixi.jl. They conclude that although similar or even more complex challenges apply to Julia when running at scale, performance is similar to traditional HPC languages. Meanwhile, Tomasi and Giordano [23] explore the shortcomings and benefits of Julia for astrophysics applications. Lin and McIntosh-Smith [24] use 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. Faingnaert et al. [25] provide optimized GEMM kernels in Julia that are competitive with cuBLAS and CUTLASS implementations. Ko et al. introduce DistStat.jl [26], which is a unified statistical computing environment in Julia for performance portability that has been tested on large-scale cloud systems. More recently, Giordano et al. [27] found competitive system performance for Julia's message passing interface (MPI) [28] MPI.jl [29] on the Fujitsu A64FX Arm-based Fugaku system. Gmys et al. [30] conclude that Julia and Python/Numba still present gaps for the scalability of multithreaded parallelizations when compared with Chapel [31].

Few recent efforts exist that leverage Python capabilities for performance via Numba. Mattson et al. present PyOMP [32], which is an OpenMP implementation for Numba with preliminary results on par with C implementations that bypasses the Python's global interpreter lock (GIL). Recent studies on GPU implementations of Python/Numba target NVIDIA CUDA-supported hardware. For example, Oden [33] identifies gaps when comparing Numba's CUDA against C CUDA performance due to Python's performance limitations, whereas Di Domenico et al. [34] show promising performance when assessing NASA Advanced Supercomputing parallel benchmark kernels with Python. Python/Numba recently deprecated AMD GPU support,[3] whereas PyCUDA, PyOpenCL [35], and Cupy [36] provide run-time access to NVIDIA and AMD GPU hardware by passing C or C++ custom kernel code for compilation using a strings interface.

b) Meta-Programming: Meta-programming has become an intensive line of research for both code and performance portability. Kokkos [9], Raja [10], [11], and Thrust [12] correspond to significant efforts in this area. They offer parallel dispatch options without specifying any detail of the target system. Despite being used by many applications, Kokkos performance portability is still an active research subject [37]–[40]. Kokkos rely on highly optimized back ends (e.g., OpenMP, OpenACC [41], CUDA/HIP) that are based on template instantiations. Thus hindering the deployment of kernel-specific optimizations (e.g., select the appropriate values for a number of blocks and threads per block, select the overlap of data transfers with computations). Templates set this kind of optimization, which happens earlier than the actual code generation phases. Every Kokkos' back end is an optimization of the common front end, it means that highly specialized techniques are used for parallel computation and memory management depending on the target back end or device (CPU or GPU). These may be different to the reference implementations used in this study, which can affect performance.

c) Directive-Based Languages: Directive-based languages are now ubiquitous within HPC. OpenMP and OpenACC are the de-facto standards for shared memory and accelerator programming. Code portability is addressed by compiler and run-time technology, which have proven sufficient to enable code porting across many HPC systems. For performance portability, however, this has not been the case. Both standards, and especially OpenMP, have increased their complexity by adding specialized constructs to guide the compiler in terms of what characteristics are available at the system level (e.g., unified virtual memory, vendor-specific features for the target device) [42]–[45].

d) Runtime Libraries: CUDA [13], HIP [14], OpenCL [46], and SyCL [47] have become common

[3] https://github.com/numba/numba/pull/6991

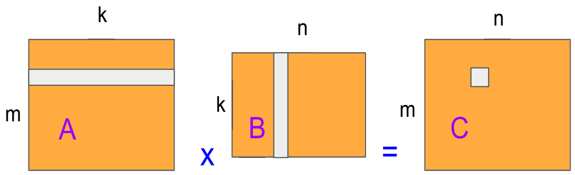


Fig. 1: Schematic representation of a hand-rolled matrix multiplication for simple GEMM kernels.

(a) C/OpenMP

```
#include "omp.h"
...
#pragma omp parallel for default(shared) \
        private(i, k, j, temp)
for (i = 0; i < A_rows; i++)
  for (k = 0; k < A_cols; k++)
    temp = A[i * A_cols + k];
    for (j = 0; j < B_cols; j++)
      C[i * B_cols + j] += temp * B[k * B_cols + j];
```

(b) Kokkos

```
Kokkos::parallel_for( "AxB=C", mdrange_policy( {0, 0}, {M,
↪   N}),  KOKKOS_LAMBDA ( int m, int n ){
    float tmp = 0.0;
    for ( int k = 0; k < K; k++ )
      tmp += A(m, k) * B(k, n);
    C(m, n) = tmp;
  }
);
```

(c) Julia

```
import Base.Threads: @threads

function gemm(A, B, C)
...
    @threads for j in 1:B_cols
        for l in 1:A_cols
            @inbounds temp = B[l, j]
            for i in 1:A_rows
                @inbounds C[i, j] += temp * A[i, l]
            end
        end
    end
end
```

(d) Python/Numba

```
from numba import njit, prange
import numpy as np

@njit(parallel=True, nogil=True, fastmath=True)
def gemm(A: np.ndarray, B: np.ndarray, C: np.ndarray):
...
    for i in prange(0, A_rows):
        for k in range(0, A_cols):
            temp = A[i, k]
            for j in range(0, B_cols):
                C[i, j] += temp * B[k, j]
```

Fig. 2: CPU multithreaded, coarse-granularity simple GEMM kernels for the programming models used in this study.

programming frameworks for HPC. For instance, Bertoni et al. [48] studied performance portability of OpenCL on Intel and NVIDIA systems. Similarly, Halver et al. [49] evaluated the portability of OpenCL for molecular dynamics applications. For SyCL, similar studies have been conducted to explore its portability to AI models [50] and sparse linear algebra kernels [51]. In general, all these efforts suffer from some limitations, as vendor-specific run-time primitives are used by programmers to achieve high levels of performance. These primitives are not portable across HPC configurations, so they must be annotated with conditional compilation adding complexity to the whole process of deploying portable code. For performance portability, the same limitation arises and for the same reason—target-specific run-time primitives.

III. Experiments

This section describes different parallelization strategies on the targeted programming models for CPUs and GPUs. The hand-rolled GEMM kernel example is shown in Fig. 1 as the product of two dense matrices. Multithreaded CPU code implementations use coarse granularity mapping larger subcomponents per thread. These subcomponents are either entire rows or columns based on whether a language is row-major (e.g., Python default numpy arrays) or column-major (e.g., Julia) to ensure equivalent computational workloads. Vectorized GPU code implementations use fine granularity mapping smaller subcomponents per thread. These subcomponents are singular elements defined by the 2D thread grid on GPU programming models. The rest of the section highlights the differences between programming models, compilation, and environment setup.

A. *CPU Implementations*

Coarse granularity for multithreaded CPU parallelization in C/OpenMP, Julia, and Python/Numba follows a similar approach to using metaprogramming directives on top of a serial *for loop*-based implementation. On the other hand, Kokkos requires an anonymous lambda function implementation written entirely from the ground up. Figure 2 shows a typical single-level parallel *for loop* implemented in C/OpenMP that provides pragmas with minimal modifications to a serial code version. As expected, index linearization of the multidimensional matrix is tracked by the programmer using non-safe memory access, whereas thread-private variables must be annotated to allow the compiler to find better optimizations. The number of threads is controlled with the `OMP_NUM_THREADS` environment variable. Additional thread policy is controlled by pinning the threads with the `OMP_PROC_BIND=true` and `OMP_PLACES=threads` environment variables, as shown in Appendix A.

Figure 2b illustrates Kokkos's programming model that uses a C++ lambda function to specify the calculations for an entry in the resulting matrix. Kokkos aims to be architecture agnostic to enable programmers to move past the low-level details of vendor- or target-specific programming models through template specialization. In practice, the target architecture is defined at compilation time with the `KOKKOS_DEVICES` flag. For instance, one must use `KOKKOS_DEVICES=Cuda` to generate binary code for NVIDIA GPUs. The Kokkos library and the source code are then compiled for the targeted architecture.

The equivalent Julia implementation is shown in Fig. 2c. Julia provides an even higher-level implementation that uses the built-in Threads module. As shown, the outer loop is immediately parallelized with the addition of the @threads macro without further specification. The downside of this approach is that the number of threads in Julia is immutable through an entire executable run because it is configured via a parameter, -t, to the Julia executable or the JULIA_NUM_THREADS environment variable. Owing to its numerical nature, Julia supports native multidimensional arrays and strong typing that can interoperate with the underlying multithreading back end implementation. The @inbounds macros prevent additional bound checks for array access. This can be configured at the executable level, but it is left in the code for illustration purposes. The JULIA_EXCLUSIVE environment variable is used to control thread policy in all runs and pin threads to cores in strict order.

As shown in Fig. 2d, Python/Numba provides a similar but slightly more invasive approach that uses metaprogramming decorators to mark the JIT compilation regions. The intended parallel for loop must be modified with the prange keyword. Although Numba supports numpy arrays, strong typing is not required within the JIT region. The NUMBA_NUM_THREADS environment variable allows one to select the number of threads, but there is currently no mechanism for setting a thread binding/pinning policy (unlike C/OpenMP and Julia).

The hand-rolled GEMM kernels shown in Fig. 2 were executed on two different CPU systems available at the OLCF: Wombat (Arm + NVIDIA) and Crusher (AMD). Table I lists the CPU specifications, the required C/OpenMP and Kokkos compilation flags, the latest Julia and Python/Numba versions, and the environment variables (i.e., ENV). We selected target-specific flags on the corresponding LLVM-based, vendor-provided compilers (e.g., ArmClang, AMDClang) to ensure maximum on-node performance by using all available cores for a range of system workloads as defined by the matrix size. Overall, Julia and Python/Numba follow a similar approach to OpenMP for multithreaded codes, whereas Kokkos creates a portable unified API for both, the CPU and GPU.

TABLE I: CPU experiment specs.

Programming/System Model	Wombat (Arm) Ampere Altra 80-core, 1-NUMA	Crusher (AMD) AMD Epyc 7A53 64-core, 4-NUMA
C OpenMP		
Compiler	ArmClang22	AMDClang14
Flags	-O3 -fopenmp	-O3 -fopenmp -march=native
C++ Kokkos	v3.6.01	
KOKKOS_DEVICES	OpenMP	
KOKKOS_ARCH	Armv8-TX2	Zen 3
Compiler	ArmClang++22	AMDClang++14
Flags	-O3 -fopenmp	-O3 -fopenmp -march=native
Julia	v1.7.2	v1.8.0-rc1
ENV	JULIA_EXCLUSIVE=1	
Python	v3.9.9	
Numba	v0.55.1	
ENV	NUMBA_OPT=3 (default)	

B. GPU Implementations

GPU implementations follow a fine granularity approach by mapping the computations required to calculate an element of the resulting matrix to a single thread. The simple matrix multiplication kernel is described in Fig. 3a for CUDA and HIP. HIP closely follows the CUDA kernel model, although the grid definition is based on the total number of launched threads, not blocks.

(a) CUDA/HIP

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
double sum = 0.0;
if( row < A_rows && col < B_cols )
{
  for(int i = 0; i < n; i++) {
    sum += A[row * n + i] * B[i * k + col];
  }
  C[row * k + col] = sum;
}
```

(b) Julia CUDA.jl

```
using CUDA
...
row = (blockIdx().x - 1) * blockDim().x + threadIdx().x
col = (blockIdx().y - 1) * blockDim().y + threadIdx().y

sum = zero(eltype(C))
if row <= size(A, 1) && col <= size(B, 2)
    for i in 1:size(A, 2)
        @inbounds sum += A[row, i] * B[i, col]
    end
    @inbounds C[row, col] = sum
end
return nothing
```

(c) Julia AMDGPU.jl

```
using AMDGPU
...
row = (workgroupIdx().x - 1) *
       workgroupDim().x + workitemIdx().x
col = (workgroupIdx().y - 1) *
      workgroupDim().y + workitemIdx().y

sum = zero(eltype(C))
if row <= size(A, 1) && col <= size(B, 2)
    for i in 1:size(A, 2)
        @inbounds sum += A[row, i] * B[i, col]
    end
    @inbounds C[row, col] = sum
end
```

(d) Python/Numba CUDA

```
from numba import cuda
from numba.cuda.cudadrv.devicearray import DeviceNDArray
import numpy as np

@cuda.jit
def gemm(A: DeviceNDArray, B: DeviceNDArray, C:
↪   DeviceNDArray):
...
    i, j = cuda.grid(2)
    if i < C.shape[0] and j < C.shape[1]:
        tmp = 0.
        for k in range(A.shape[1]):
            tmp += A[i, k] * B[k, j]
        C[i, j] = tmp
```

Fig. 3: GPU fine-granularity, hand-rolled GEMM kernels for the programming models used in this study.

Julia GPU programming models use the vendor-specific

CUDA.jl [52], [53] and AMDGPU.jl [54] implementations for NVIDIA and AMD GPUs, respectively. They provide high-level mechanics to define multidimensional arrays (CUArray and ROCArray) on GPU devices. Julia also provides the KernelAbstractions.jl [55] package for writing portable kernels while still maintaining dependence on either CUArray or ROCArray. Figures 3b and 3c show the corresponding kernel implementations on CUDA.jl and AMDGPU.jl, respectively. As shown, the close resemblance to the CUDA/HIP models for thread mapping makes for an easy transition for those familiar with these programming models. To its advantage, Julia uses multidimensional arrays and added functionality in the device kernel code to provide a powerful syntax for GPU programming.

Python/Numba provides a very simple interface to access CUDA kernel functionality, as illustrated in Fig. 3d. Unlike the CUDA/HIP model, it provides a simple `cuda.grid` mapping function between the row and column coordinate with a GPU thread. Similar to Julia, Python enables multidimensional matrix syntax on device kernels through the `devicearray` interface. As mentioned in Section II, Python/Numba support for AMD GPUs is currently deprecated.

TABLE II: GPU experiment specs.

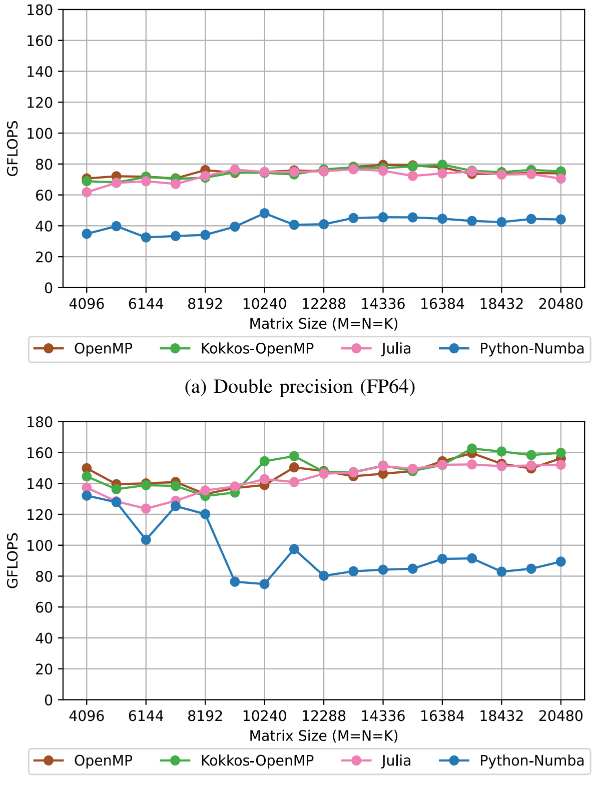
Programming/System Model	Wombat (NVIDIA) A100 Ampere	Crusher (AMD) MI250X
C CUDA/HIP		
Compiler	nvcc v11.5.1	hipcc v14.0.0
Flags	-arch=sm_80	–amdgpu-target=gfx908
C++ Kokkos	v3.6.01	
KOKKOS_DEVICES	Cuda	Hip
KOKKOS_ARCH	Ampere80	Vega908
Compiler	CUDA v11.5.1	HIP v14.0.0
Flags	-expt-extended-lambda -Xcudafe -arch=sm_80	–amdgpu-target=gfx908
Julia	v1.7.2 CUDA.jl	1.8.0-rc1 AMDGPU.jl
Python **Numba** Flags	v3.9.9 v0.55.1	Not supported

IV. Results

This section characterizes the results obtained in the experiments described in Section III. All numbers were obtained by running the GEMM kernels several (at least 5 or 10) times and excluding an initial warm-up step. This exclusion also discards initial communication (threads and GPUs) and JIT compilation overheads in Julia and Python/Numba. Due to the dedicated nature of the nodes, the results are the most likely performance value without doing an exhaustive variability analysis and only presenting the average expected value. We consider that variability is at face value a characteristic of the system, rather than an effect of the programming model per-se as it is the goal of this comparison. Nevertheless, reproducible artifacts are provided in Appendix A for independent verification.

A. *CPU Performance*

a) Crusher AMD EPYC 7A53: Figure 4 shows the results obtained for the multithreaded CPU implementations on the Crusher system for (a) double precision and (b) single precision. Overall, Kokkos/OpenMP and Julia threads perform comparably with the vendor ArmClang C/OpenMP implementation, whereas Python/Numba is still behind in terms of performance. OpenMP and Julia use environment flags to bind threads to CPU resources, as shown in Table I; this option is not available in the Python/Numba APIs.



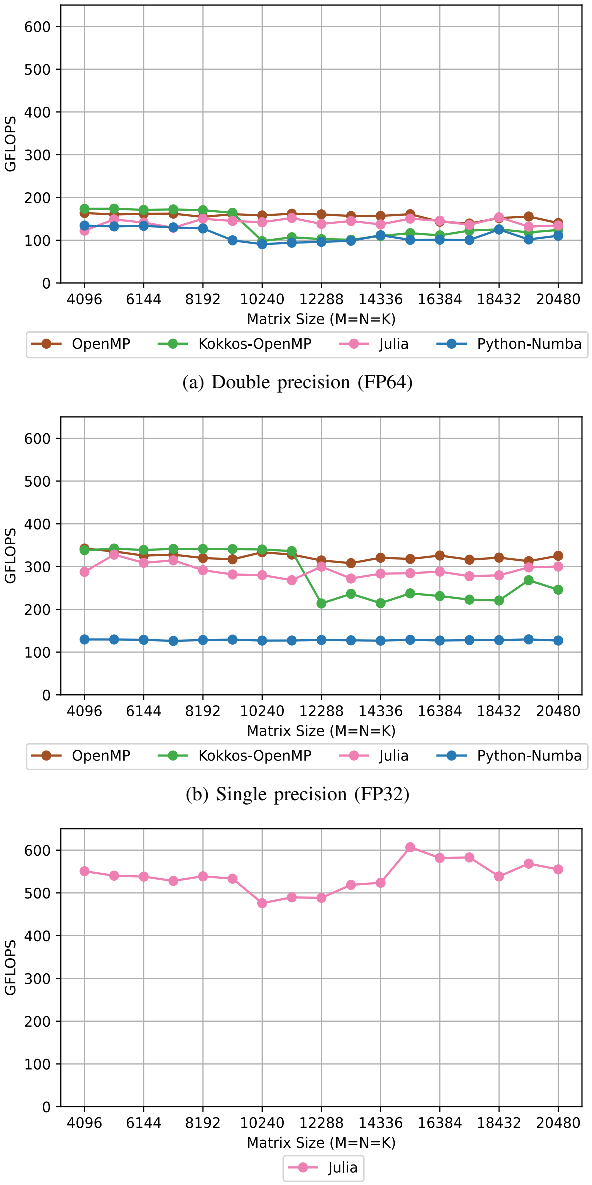
(a) Double precision (FP64)

(b) Single precision (FP32)

Fig. 4: Crusher multithreaded CPU performance using 64 threads across 4 NUMA regions.

b) Wombat Arm Altra Ampere: Performance results on Arm CPUs are shown in Fig. 5 for (a) double and (b) single precision. Notably, Kokkos, which is using the OpenMP back end, experiences a slowdown in both cases. Meanwhile, Julia's performance is almost on par with the vendor OpenMP implementations.

Half-precision floating point (FP16) is not supported for Python/Numba regions combined with numpy's `Float16` random number capabilities, so input matrices were populated with 1s. Half-precision support in Julia is under active development. We obtained very low performance on Crusher AMD CPUs (not reported in this work), and this is expected



(a) Double precision (FP64)

(b) Single precision (FP32)

(c) Half precision (FP16)

Fig. 5: Wombat multithreaded CPU performance using 80 threads.

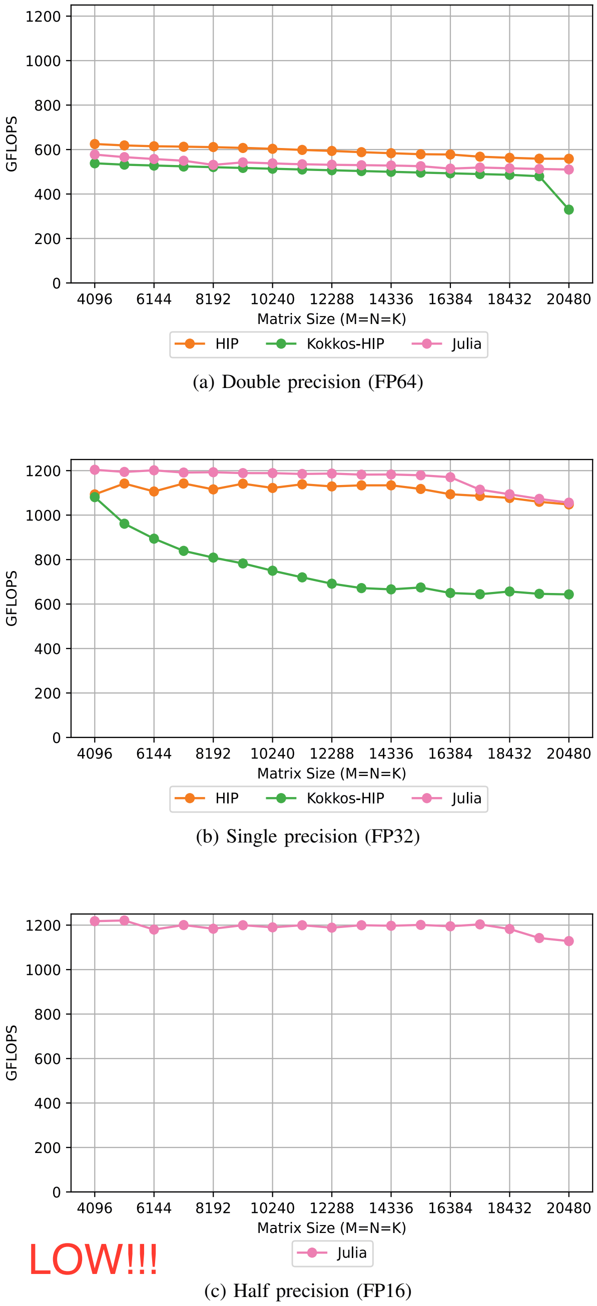
to improve as Julia's native FP16 support matures.[4] The Julia threads implementation on Arm worked seamlessly and provided the expected levels of performance, as shown in Fig 5c. The literature also contains a recent discussion of Julia's FP16 performance on Arm systems [27].

B. GPU Performance

a) Crusher AMD MI250X: Figure 6 presents the simple GEMM performance for HIP, Kokkos-HIP, and Julia using AMDGPU.jl for different floating-point precisions on Crusher's AMD MI250X GPU, which is similar to the GPUs found in the OLCF's Frontier system (Table II). Python/Numba is not supported on AMD GPUs. As shown in Fig. 6a, for double-precision runs, the vendor-provided HIP implementation achieves the highest performance. This is followed by Julia using AMDGPU.jl and Kokkos/HIP, both of which reach competitive levels but still do not match HIP for all matrix sizes because the overheads introduced appear to be constant. Kokkos has a repeatable slowdown at the largest size, and this might require further investigation on this system. The performance for single-precision floating point is shown in Fig. 6b. As expected, all models provide an increase in performance, but Kokkos + HIP exhibits a consistent decrease, which again requires further investigation. Interestingly, Julia with AMDGPU.jl shows slightly better performance than the vendor HIP implementation, although the differences become small for larger matrix sizes and this could simply be the variability on this particular system. Lastly, Julia AMDGPU.jl performance results are presented in Fig. 6c for half-precision multiplications stored on a single-precision matrix (Fig. 1c). No noticeable improvements are shown when compared to single-precision runs. Nevertheless, other programming models do not provide seamless half-precision support, whereas Julia currently supports random number generation and kernel computations on AMD GPUs.

b) Wombat NVIDIA A100: Figure 7 presents the simple GEMM performance for (1) CUDA, Kokkos/CUDA, and Julia using CUDA.jl and (2) Python/Numba using CUDA with different floating-point precisions on Wombat's NVIDIA A100 GPUs (Table II). Double-precision runs shown in Fig. 7a show that Julia using CUDA.jl has a constant overhead when compared to the vendor-provided CUDA implementation. The generated low-level Parallel Thread Execution (PTX) instruction set architecture (ISA), not shown here, indicated a difference in unrolled loop instructions, 2 for CUDA.jl and 4 in the native CUDA. Deeper investigation is required to generate more effective heuristic models for different kernel workloads. Kokkos and Python/Numba using a CUDA back end consistently underperform, which raises questions about the configuration and/or actual GPU runs. Both Kokkos and Python/Numba were verified by using NVIDIA's `nvprof` profiler to corroborate GPU activity. Figure 7b shows the performance for single-precision runs. As expected, the performance of the vendor-provided CUDA implementation increases significantly, whereas other implementations still present gaps for this case. Julia, Kokkos/CUDA, and Python/Numba show small performance increases of around 10% between double- and single-precision runs. Lastly, we show the half-precision results for the supported Julia with CUDA.jl and Python/Numba implementations. The Python half-precision implementation must be modified because ran-

[4] https://github.com/JuliaLang/julia/issues/45542

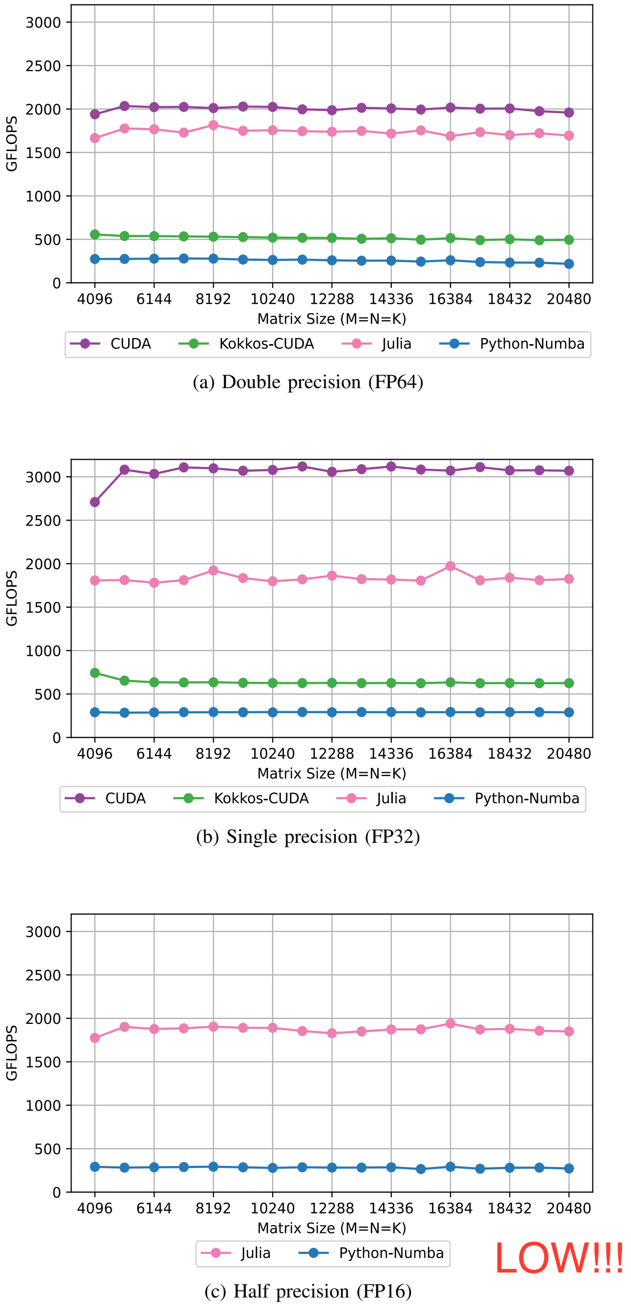


(a) Double precision (FP64)

(b) Single precision (FP32)

(c) Half precision (FP16)

Fig. 6: Simple GEMM performance on Crusher AMD MI250X GPU using 32 × 32 thread block sizes.



(a) Double precision (FP64)

(b) Single precision (FP32)

(c) Half precision (FP16)

Fig. 7: Simple GEMM performance on Wombat NVIDIA A100 using 32 × 32 thread block sizes.

dom number generation is not supported using numpy's `float16` type. Nevertheless, we observed no performance gains over the single-precision counterparts.

V. Performance Portability

Although, there is no agreed-upon metric for performance portability, we reference some of the proposals found in the current literature for parallel applications. However, the present work focuses on evaluating the performance portability of different programming models.

One of the first attempts to do so took place at the US Department of Energy's meeting on performance portability [56], during which different definitions were proposed. Pennycook et al. [57] proposed a unique definition, which has since been adopted by the HPC community: "A measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set." Since then, a set of different metrics and formulas was defined [58]. Recently, this definition was extended to define performance portability of a programming model as *the ratio of the mean performance of a portable model and the mean performance of a non-portable one* [11].

The metric proposed to compute the performance portability of a programming model, M, was defined as follows:

$$\Phi_M = \frac{\sum_{i \in T} e_i(a)}{\mid T \mid}, \tag{1}$$

where, in our case, $e_i(a)$ is the performance efficiency of the matrix-matrix multiplication of the portable programming model, M (i.e., Kokkos, Julia, or Python-Numba), divided by the vendor-specific implementation. For example, the performance efficiency of Julia on an MI250X AMD GPU would be formulated as follows:

$$e_{MI250x} = \frac{Julia\ Performance}{HIP\ Performance}. \tag{2}$$

The computed efficiencies for this simple kernel on each hardware target and the overall programming model, in Eq. (1) are shown in Table III. The vendor C/OpenMP performance was used as the architecture-specific reference model for CPU analysis, and CUDA and HIP performance was used as the architecture-specific reference model for NVIDIA and AMD GPUs, respectively. The performance efficiency of Kokkos and Julia is similar, with the exception of e_{A100}. Python/Numba has the lowest numbers based on the performance results when considering that AMD GPUs are not supported. As for programming models efficiency for this simple kernel, Julia has the best scores followed by Kokkos and Python/Numba. No significant difference is seen when comparing double-precision against single-precision analysis, and the portability of all models is slightly lower for single-precision floating point computations.

TABLE III: Performance Efficiency of Kokkos, Julia, and Python/Numba for each experiment.

Architecture	Kokkos	Julia	Python/Numba
Double precision			
$e_{Epyc\ 7A53}$	**0.994**	0.912	0.550
$e_{Ampere\ Altra}$	0.854	**0.907**	0.713
e_{MI250x}	0.842	**0.903**	-
e_{A100}	0.260	**0.867**	0.130
Φ_M	0.738	**0.897**	0.348
Single precision			
$e_{Epyc\ 7A53}$	**1.014**	0.976	0.655
$e_{Ampere\ Altra}$	0.836	**0.900**	0.400
e_{MI250x}	0.677	**1.050**	-
e_{A100}	0.208	**0.600**	0.095
Φ_M	0.684	**0.882**	0.288

VI. Conclusions

We studied the high-productivity, dynamic languages Julia and Python/Numba as high-level interfaces to LLVM and compared them with portable implementations of C/OpenMP and Kokkos. Performance results and efficiency metrics for a simple hand-rolled gense matrix multiplication are presented on exascale node architectures—Wombat, which uses Arm Ampere CPUs and 2 NVIDIA A100 GPUs, and Crusher (Frontier's test bed), which is equipped with AMD EPYC 7A53 CPUs and 8 MI250X GPUs. Results for double- and single-precision floating point operations indicate that the default Julia implementations have comparable performance on these platforms. For CPUs, Julia performance was comparable to C/OpenMP combined with LLVM-based ArmClang and AMDClang vendor compilers. For the AMD GPUs, Julia AMDGPU.jl performance was comparable to HIP. Julia's productivity and performance benefits are the result of being designed from the ground up to leverage LLVM. Nevertheless, there is still a performance gap on NVIDIA A100 GPUs for single-precision floating point cases and further investigation is needed into the default low-level PTX ISA code generated. We observe that Python/Numba implementations still lack the support needed to reach comparable CPU and GPU performance on these systems, and AMD GPU support is deprecated. Kokkos provides an interesting approach for performance portability, which still depends on the back end and several compilation and policy settings. Overall, Julia and Python/Numba programming models provide high-productivity CPU and GPU APIs for easy access to write LLVM-compiled kernels. Additionally, their powerful ecosystems for data analysis and workflows in HPC, seamless half-precision floating point support, and interoperability with vendor back ends all add value to the scientific discovery process. Future work should continue to explore their use in more complex HPC workloads as their LLVM-based implementations and supportive ecosystems become more mature to achieve desired performance portability on heterogeneous hardware.

Acknowledgment

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the US Department of Energy Office of Science and the National Nuclear Security Administration. This research used resources of the Oak Ridge Leadership Computing Facility 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.

References

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

[2] 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.

[3] 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.

[4] J. S. Vetter, R. Brightwell, M. Gokhale, P. McCormick, R. Ross, J. Shalf, K. Antypas, D. Donofrio, T. Humble, C. Schuman, B. Van Essen, S. Yoo, A. Aiken, D. Bernholdt, S. Byna, K. Cameron, F. Cappello, B. Chapman, A. Chien, M. Hall, R. Hartman-Baker, Z. Lan, M. Lang, J. Leidel, S. Li, R. Lucas, J. Mellor-Crummey, P. Peltz Jr., T. Peterka, M. Strout, and J. Wilke, "Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity," 12 2018.

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

[6] 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.

[7] OpenMP Architecture Review Board, "OpenMP application program interface version 5.2," November 2021. [Online]. Available: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf

[8] OpenACC Architecture Review Board, "OpenACC application program interface version 3.1," November 2020. [Online]. Available: https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.1-final.pdf

[9] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731514001257

[10] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "Raja: Portable performance for large-scale scientific applications," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 71–81.

[11] 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

[12] NVIDIA, "The API reference guide for Thrust, the CUDA C++ template library." May 2022. [Online]. Available: https://docs.nvidia.com/cuda/thrust/index.html

[13] ——, "CUDA Toolkit Documentation - v11.7.0," May 2022. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[14] AMD, "AMD ROCm v5.2 Release," June 2022. [Online]. Available: https://rocmdocs.amd.com/en/latest/Current_Release_Notes/Current-Release-Notes.html#amd-rocm-v5-2-release

[15] T. Ben-Nun, T. Gamblin, D. S. Hollman, H. Krishnan, and C. J. Newburn, "Workflows are the new applications: Challenges in performance, portability, and productivity," in *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2020, pp. 57–69.

[16] L. S. Blackford, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry *et al.*, "An updated set of basic linear algebra subprograms BLAS," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 135–151, 2002.

[17] H. Kim, H. Nam, W. Jung, and J. Lee, "Performance analysis of CNN frameworks for GPUs," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 55–64.

[18] X. Li, G. Zhang, H. H. Huang, Z. Wang, and W. Zheng, "Performance Analysis of GPU-Based Convolutional Neural Networks," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 67–76.

[19] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "NVIDIA Tensor Core Programmability, Performance & Precision," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 522–531.

[20] C. Brown, A. Abdelfattah, S. Tomov, and J. Dongarra, "Design, Optimization, and Benchmarking of Dense Linear Algebra Algorithms on AMD GPUs," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, 2020, pp. 1–7.

[21] 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.

[22] 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.

[23] M. Tomasi and M. Giordano, "Towards new solutions for scientific computing: the case of Julia," Dec. 2018.

[24] 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.

[25] 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.

[26] S. Ko, H. Zhou, J. Zhou, and J.-H. Won, "Diststat.jl: Towards unified programming for high-performance statistical computing environments in julia," Oct. 2020.

[27] 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.

[28] 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.

[29] 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

[30] J. Gmys, T. Carneiro, N. Melab, E.-G. Talbi, and D. Tuyttens, "A comparative study of high-productivity high-performance programming languages for parallel metaheuristics," *Swarm and Evolutionary Computation*, vol. 57, p. 100720, sep 2020.

[31] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[32] T. G. Mattson, T. A. Anderson, and G. Georgakoudis, "Pyomp: Multithreaded parallel programming in python," *Computing in Science & Engineering*, vol. 23, no. 6, pp. 77–80, 2021.

[33] L. Oden, "Lessons learned from comparing C-CUDA and Python-Numba for GPU-Computing," in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2020, pp. 216–223.

[34] D. Di Domenico, G. G. H. Cavalheiro, and J. V. F. Lima, "Nas parallel benchmark kernels with python: A performance and programming effort analysis focusing on gpus," in *2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)* 2022, pp. 26–33.

[35] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to gpu run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157–174, 2012.

[36] R. Nishino and S. H. C. Loomis, "Cupy: A numpy-compatible library for nvidia gpu calculations," *31st confernce on neural information processing systems*, vol. 151, no. 7, 2017.

[37] K. Teranishi, D. M. Dunlavy, J. M. Myers, and R. F. Barrett, "Sparten: Leveraging kokkos for on-node parallelism in a second-order method for fitting canonical polyadic tensor models to poisson data," in *2020 IEEE High Performance Extreme Computing Conference, HPEC 2020, Waltham, MA, USA, September 22-24, 2020.* IEEE, 2020, pp. 1–7.

[38] R. Halver, J. H. Meinke, and G. Sutmann, "Kokkos implementation of an ewald coulomb solver and analysis of performance portability," *J. Parallel Distributed Comput.*, vol. 138, pp. 48–54, 2020.

[39] S. Rajamanickam, S. Acer, L. Berger-Vergiat, V. Q. Dang, N. D. Ellingwood, E. Harvey, B. Kelley, C. R. Trott, J. Wilke, and I. Yamazaki, "Kokkos Kernels: Performance Portable Sparse/Dense Linear Algebra and Graph Kernels," *CoRR*, vol. abs/2103.11991, 2021. [Online]. Available: https://arxiv.org/abs/2103.11991

[40] J. A. Ellis and S. Rajamanickam, "Scalable Inference for Sparse Deep Neural Networks using Kokkos Kernels," in *2019 IEEE High Perfor-*

mance Extreme Computing Conference, HPEC 2019, Waltham, MA, USA, September 24-26, 2019. IEEE, 2019, pp. 1–7.

[41] 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

[42] S. Catalán, T. Usui, L. Toledo, X. Martorell, J. Labarta, and P. Valero-Lara, "Towards an Auto-Tuned and Task-Based SpMV (LASs Library)," in *OpenMP: Portable Multi-Level Parallelism on Modern Systems - 16th International Workshop on OpenMP, IWOMP 2020, Austin, TX, USA, September 22-24, 2020, Proceedings*, ser. Lecture Notes in Computer Science, K. F. Milfeld, B. R. de Supinski, L. Koesterke, and J. Klinkenberg, Eds., vol. 12295. Springer, 2020, pp. 115–129. [Online]. Available: https://doi.org/10.1007/978-3-030-58144-2_8

[43] P. Valero-Lara, D. Andrade, R. Sirvent, J. Labarta, B. B. Fraguela, and R. Doallo, "A fast solver for large tridiagonal systems on multi-core processors (lass library)," *IEEE Access*, vol. 7, pp. 23 365–23 378, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2900122

[44] P. Valero-Lara, S. Catalán, X. Martorell, T. Usui, and J. Labarta, "sLASs: A fully automatic auto-tuned linear algebra library based on OpenMP extensions implemented in OmpSs (LASs Library)," *J. Parallel Distributed Comput.*, vol. 138, pp. 153–171, 2020. [Online]. Available: https://doi.org/10.1016/j.jpdc.2019.12.002

[45] P. Valero-Lara, S. Catalán, X. Martorell, and J. Labarta, "BLAS-3 optimized by ompss regions (lass library)," in *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2019, Pavia, Italy, February 13-15, 2019*. IEEE, 2019, pp. 25–32. [Online]. Available: https://doi.org/10.1109/EMPDP.2019.8671545

[46] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science and Engg.*, vol. 12, no. 3, p. 66–73, may 2010.

[47] "Sycl 2020 specification revision 5," May 2022. [Online]. Available: https://www.khronos.org/files/sycl/sycl-2020-reference-guide.pdf

[48] C. Bertoni, J. Kwack, T. Applencourt, Y. Ghadar, B. Homerding, C. Knight, B. Videau, H. Zheng, V. A. Morozov, and S. Parker, "Performance portability evaluation of opencl benchmarks across intel and NVIDIA platforms," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*. IEEE, 2020, pp. 330–339. [Online]. Available: https://doi.org/10.1109/IPDPSW50202.2020.00067

[49] R. Halver, W. Homberg, and G. Sutmann, "Function portability of molecular dynamics on heterogeneous parallel architectures with opencl," *J. Supercomput.*, vol. 74, no. 4, pp. 1522–1533, 2018. [Online]. Available: https://doi.org/10.1007/s11227-017-2232-2

[50] M. Tanvir, K. Narasimhan, M. Goli, O. E. Farouki, S. Georgiev, and I. Ault, "Towards performance portability of AI models using SYCL-DNN," in *IWOCL'22: International Workshop on OpenCL, Bristol, United Kingdom, May 10 - 12, 2022*. ACM, 2022, pp. 23:1–23:3. [Online]. Available: https://doi.org/10.1145/3529538.3529999

[51] T. Sabino and M. Goli, "Toward performance portability of highly parametrizable TRSM algorithm using SYCL," in *IWOCL'21: International Workshop on OpenCL, Munich Germany, April, 2021*, S. McIntosh-Smith, Ed. ACM, 2021, pp. 5:1–5:10. [Online]. Available: https://doi.org/10.1145/3456669.3456694

[52] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[53] T. Besard, V. Churavy, A. Edelman, and B. De Sutter, "Rapid software prototyping for heterogeneous and distributed platforms," *Advances in Engineering Software*, vol. 132, pp. 29–46, 2019.

[54] J. Samaroo, V. Churavy, W. Phillips, A. Ramadhan, J. Barmparesos, J. TagBot, L. Räss, M. Schanen, T. Besard, A. Smirnov, T. Arakaki, S. Antholzer, Alessandro, C. Elrod, M. Raayai, and T. Hu, "JuliaGPU/AMDGPU.jl: v0.4.1," Aug. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6949520

[55] V. Churavy, D. Aluthge, L. C. Wilcox, J. Schloss, S. Byrne, M. Waruszewski, J. Samaroo, A. Ramadhan, Meredith, S. Schaub, J. Bolewski, A. Smirnov, C. Kawczynski, C. Hill, J. Liu, O. Schulz, Oscar, P. Haraldsson, T. Arakaki, and T. Besard, "JuliaGPU/KernelAbstractions.jl: v0.8.3," Jun. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6742177

[56] U. D. of Energy, "DOE Centers of Excellence Performance Portability Meeting," April 2016. [Online]. Available: https://asc.llnl.gov/sites/asc/files/2020-09/COE-PP-Meeting-2016-FinalReport_0.pdf

[57] S. Pennycook, J. Sewall, and V. Lee, "Implications of a metric for performance portability," *Future Generation Computer Systems*, vol. 92, pp. 947–958, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X17300559

[58] A. Marowka, "Toward a better performance portability metric," in *29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2021, Valladolid, Spain, March 10-12, 2021*. IEEE, 2021, pp. 181–184. [Online]. Available: https://doi.org/10.1109/PDP52278.2021.00036

Appendix A
Artifact Description for Reproducibility

The code used for this study is hosted on GitHub: https://github.com/williamfgc/simple-gemm. Each implementation has its own directory: C, Kokkos, Julia, and Python. The `scripts` directory contains the configurations for each experiment on OLCF systems. Figures 8 and 9 show examples of scripts to run C/OpenMP and Julia experiments on Wombat.

```bash
#!/bin/bash

EXECUTABLE=../simple-gemm/c/gemm-dense-openmp64-armclang
module load ARM_Compiler_For_HPC/22.0
Ms=( 4096 5120 ... 19456 20480 )

export OMP_PROC_BIND=true
export OMP_PLACES=threads
export OMP_NUM_THREADS=80

for M in "${Ms[@]}"; do
  salloc -N 1 -p Ampere -t 10:00:00 srun -n 1 -c $t \
    $EXECUTABLE $M $M $M 5 >
    ↪  Ampere-ARMClang22-${t}t-${M}M_5s_threads.log 2>&1
done
```

Fig. 8: Wombat CPU C/OpenMP launch script.

```bash
#!/bin/bash

module load nvhpc-nompi/22.1
module load julia/1.7.3
export JULIA_CUDA_USE_BINARYBUILDER=false
GemmDenseCUDADIR=../../simple-gemm/julia/GemmDenseCUDA
EXECUTABLE=$GemmDenseCUDADIR/gemm-dense-cuda.jl

Ms=( 4096 5120 ... 19456 20480 )

for M in "${Ms[@]}"; do

  salloc -N 1 -p Ampere -t 10:00:00 --gres=gpu:1 \
    srun -n 1 julia -O3 --project=$GemmDenseCUDADIR \
    $EXECUTABLE $M $M $M 5 \
    > A100-Julia1_7_3-${M}M_5s_F64.log 2>&1 &
done
```

Fig. 9: Wombat GPU Julia launch script.

Tables I and II describe the software stack and compilation flags used to generate all the experiments. Kokkos implementations are found in `simple-gemm/cpp/Kokkos/` with its own compilation framework.