

Mojo: MLIR-Based Performance-Portable HPC Science Kernels on GPUs for the Python Ecosystem

William F Godoy*

Oak Ridge National Laboratory
Oak Ridge, TN, USA
godoywf@ornl.gov

Tatiana Melnichenko*

Innovative Computing Laboratory.
The University of Tennessee,
Knoxville.
Knoxville, TN, USA
Oak Ridge National Laboratory.
Oak Ridge, TN, USA
tdehoff@vols.utk.edu

Pedro Valero-Lara

Oak Ridge National Laboratory
Oak Ridge, TN, USA
valerolarap@ornl.gov

Wael Elwasif

Oak Ridge National Laboratory
Oak Ridge, TN, USA
elwasifwr@ornl.gov

Philip Fackler

Oak Ridge National Laboratory
Oak Ridge, TN, USA
facklerpw@ornl.gov

Rafael Ferreira Da Silva

Oak Ridge National Laboratory
Oak Ridge, TN, USA
silvarf@ornl.gov

Keita Teranishi

Oak Ridge National Laboratory
Oak Ridge, TN, USA
teranishik@ornl.gov

Jeffrey S Vetter

Oak Ridge National Laboratory
Oak Ridge, TN, USA
vetter@ornl.gov

Abstract

We explore the performance and portability of the novel Mojo language for scientific computing workloads on GPUs. As the first language based on the LLVM's Multi-Level Intermediate Representation (MLIR) compiler infrastructure, Mojo aims to close performance and productivity gaps by combining Python's interoperability and CUDA-like syntax for compile-time portable GPU programming. We target four scientific workloads: a seven-point stencil (memory-bound), BabelStream (memory-bound), miniBUDE (compute-bound), and Hartree–Fock (compute-bound with atomic operations); and compare their performance against vendor baselines on NVIDIA H100 and AMD MI300A GPUs. We show that Mojo's performance is competitive with CUDA and HIP for memory-bound kernels, whereas gaps exist on AMD GPUs for atomic operations and for fast-math compute-bound kernels on both AMD and NVIDIA GPUs. Although the learning curve and programming requirements are still fairly low-level, Mojo can close significant

gaps in the fragmented Python ecosystem in the convergence of scientific computing and AI.

CCS Concepts

- Computing methodologies → Parallel programming languages;
- General and reference → Performance;
- Software and its engineering → Very high level languages.

Keywords

Mojo, GPU, HPC, science kernels, performance portability, productivity, Python

ACM Reference Format:

William F Godoy, Tatiana Melnichenko, Pedro Valero-Lara, Wael Elwasif, Philip Fackler, Rafael Ferreira Da Silva, Keita Teranishi, and Jeffrey S Vetter. 2025. Mojo: MLIR-Based Performance-Portable HPC Science Kernels on GPUs for the Python Ecosystem. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA*. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3731599.3767573>

*Both authors contributed equally to this research.

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (<https://www.energy.gov/doe-public-access-plan>).



This work is licensed under a Creative Commons Attribution 4.0 International License.
SC Workshops '25, St Louis, MO, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1871-7/2025/11
<https://doi.org/10.1145/3731599.3767573>

1 Introduction

Mojo [42] is the latest technology for closing performance and productivity gaps in the Python [56] language and ecosystem, including in AI workloads. Developed by Modular Inc., Mojo is a stand-alone language that provides ahead-of-time (AOT) and just-in-time (JIT) compilation options based entirely on the Multi-Level Intermediate Representation (MLIR) [36], which is the next-generation of the widely adopted LLVM [35] compiler infrastructure. Mojo enables productivity through the reusability of Python's rich library ecosystem via module interoperability and offers performance portability across GPUs from different vendors. It currently supports the latest NVIDIA GPUs, and support was recently added for AMD's MI300

GPUs. Hence, similar to modern programming languages such as Julia [8] and Rust [40], Mojo aims to mitigate the costly fragmentation that arises when using multiple languages and ecosystems to achieve high-level characteristics such as productivity (e.g., Python) and safety combined with the low-level performance of compiled languages (e.g., C++, C, Fortran).

Heterogeneous computing architectures (e.g., CPUs + GPU accelerators) have been the primary drivers of the supercomputing and AI landscapes in recent decades. The TOP500's¹ Linpack [20] ranking from June 2025 shows that 9 out of the top 10 fastest supercomputers in the world are accelerated by GPUs. Moreover, massive investments in AI supercomputers (e.g., xAI's Colossus) continue to drive the demand for GPUs [47]. For this reason, GPU programmability and portability are crucial for the entire high-performance computing (HPC) ecosystem. Strategies to address portability needs have mainly targeted C, C++, and Fortran HPC languages through new standards, including OpenMP [44], OpenCL [53], OpenACC [58], and SYCL [18], and through third-party programming models, including Kokkos [54] and RAJA [5]. Since June of 2025, Mojo provides this capability of supporting portable GPU programming directly into the language standard library.

We attempt to answer the following research question: *Can scientific users benefit from Mojo's performance-portable GPU codes?* Despite the Mojo language being particularly novel, with many design decisions still in the development stages, and not completely open-source until 2026, we provide an initial exploration of Mojo's unique value proposition—using it to write performance-portable GPU kernels using the language's standard library. Unlike AI methods and math kernels that are well defined in scientific numerical libraries, custom science kernels can have unique computational characteristics that affect their performance. For this reason, we developed ports for four widely used scientific kernels—(i) seven-point stencil (memory-bandwidth bound), (ii) BabelStream (memory-bandwidth bound), (iii) miniBUDE (compute-bound), and (iv) Hartree–Fock (compute-bound with atomic operations)—so they can use Mojo's portable GPU code capabilities. The kernels' performance and profiling metrics (where appropriate) are then analyzed and compared with their equivalent CUDA and HIP baselines.

The rest of the paper is organized as follows: Section 2 provides background information on the Mojo language syntax and compile-time characteristics in connection with MLIR and an overview of the selected science kernels. Section 3 introduces our Methodology for porting the codes to Mojo's memory and kernel-launching programming model and describes the targeted benchmarks and systems used in our experiments. Section 4 describes the performance results and analysis of the Mojo-ported codes running on NVIDIA and AMD GPUs and compares the Mojo implementations with the corresponding vendor baselines in CUDA and HIP, respectively. We also summarize our comparisons by applying a performance-portability metric to all of our runs on the NVIDIA and AMD GPUs. Related work is presented in Section 5, including attempts to close performance-portability and productivity gaps. Section 6 presents the conclusions from our study. All software artifacts are provided for reproducibility in the Appendix section for Artifact Description and Evaluation. To the best of our knowledge, this is the first

work on Mojo's unifying portable GPU accelerator model to target scientific computing kernels.

2 Background

2.1 Mojo Programming

Mojo's value proposition is in combining the ease of use and flexibility of Python with the performance of compile-time programming languages such as C++. Mojo can achieve this by being the first language fully built on MLIR. Figure 1 illustrates Mojo's approach to tackling the fragmentation between compiled languages and the Python ecosystem. We are particularly interested in how Mojo performs with science kernels given the June 2025 announcement of support for the AMD MI300 series GPUs in addition to the well-supported NVIDIA Hopper and Ampere GPUs. Notably, reusing the same vendor-agnostic Mojo GPU modules adds a very unique characteristic for language-embedded GPU portability. This is in contrast with other HPC languages in which support is provided by standard third-party libraries. In fact, Mojo is unique in several ways:

- Features MLIR-based compile time, JIT or AOT, for performance-portable GPU programming
- Includes run-time interoperability with Python and its rich library ecosystem
- Implements memory safety via variable lifetime and ownership (not garbage collected)

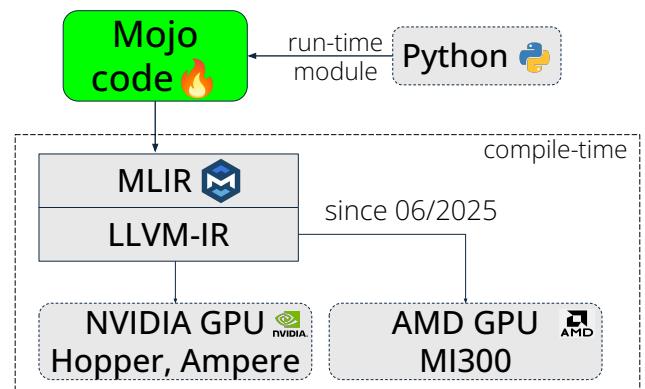


Figure 1: Mojo integrates Python at run time and performance-portable GPU code at compile time.

Listing 1 provides an example of Mojo's syntax, which illustrates these key features. Mojo uses a lower-level CUDA/HIP-like programming model for GPU device memory allocation and kernel launching. Because Mojo is intended to leverage compilation via MLIR, performance-critical information such as problem and block sizes, array layout, and variable types must be well-defined at compile time. The latter also makes JIT or AOT indistinguishable for the generated intermediate representation because both mechanisms will be exposed to the same information. This *fixed layout* programming style was well suited for the early days of computing, but a shift in the dominant HPC languages has since occurred, in particular C and Fortran 90, and allowed for dynamic

¹www.top500.org

memory allocations, which are the building blocks of the majority of HPC codes. In contrast, run-time interoperability with Python is done via modules, and it is not precompiled with MLIR, thus preserving Python's dynamic nature. Hence, Mojo introduces this fragmentation in the language by keeping a clear separation between (i) JIT- or AOT-MLIR compiled high-performance code and (ii) Python interoperability as a run-time-only entity.

```
# Mojo's gpu portable standard library module
from gpu.host import DeviceContext
from gpu.id import block_dim, block_idx, thread_idx
from layout import Layout, LayoutTensor
...
# Mojo's Python interoperability module
from python import Python

# Compile-time GPU programming
# Requires data tensor type, size and layout
alias dtype = DType.float32
alias Nx = 1024
alias layout = Layout.row_major(Nx)

alias block_size = 256
alias num_blocks = ceildiv(Nx, block_size)

# GPU kernel
fn fill_one(tensor: LayoutTensor[mut=True, dtype, layout]):
    var tid = block_idx.x * block_dim.x + thread_idx.x
    if tid < Nx:
        tensor[tid] = 1

fn main():
    # GPU memory model
    ctx = DeviceContext()
    d_u = ctx.enqueue_create_buffer(dtype)(nx)
    u_tensor = LayoutTensor[dtype, layout](d_u)

    # GPU kernel execution
    ctx.enqueue_function(fill_one)(u_tensor,
        grid_dim=num_blocks,
        block_dim=block_size
    )
    ctx.synchronize()

    # Python interoperability uses a separate runtime approach
    np = Python.import_module('numpy')
    array = np.array(Python.list(1, 2, 3))
    print(array)
```

Listing 1: Compile-time GPU programming and run-time Python interoperability in Mojo.

Unlike most HPC software applications, in which much of the information is known only at run time (i.e., compile once, run many times), the reliance on MLIR optimizations restricts Mojo programs to providing performance-critical information (e.g., sizes, types, memory layouts, and variable lifetime) at compile time. Although this is desired for JIT approaches (e.g., Julia), it can be a challenge when deploying code that has access to only certain information at run time (e.g., real-time workflows, adaptive mesh codes, statistical random walks). Additionally, Python's stack is not compiled effectively with Mojo's AOT compilation (`mojo build`), thus keeping Python interactions dynamic and limited to run time through interoperability with CPython.

Mojo also utilizes a memory model based on value ownership, which provides both performance and memory safety without relying on a garbage collector. Heap-allocated values have exactly one owner at a time. When a value's lifetime ends, Mojo automatically deallocates it by calling its destructor, thus eliminating the need for a garbage collector or the C++ scope-based *resource allocation is initialization* pattern.

Mojo's approach to providing performance in the Python ecosystem is very different from other solutions. Rather than optimizing Python from within, Mojo makes a clear distinction between the rich compile-time performance-critical sections and the run-time Python interoperability. While this is already the case for Python

libraries based on C or C++ thanks to performance-portable and interoperable libraries (e.g., pybind11, nanobind), Mojo's unique approach combines these aspects into the language to enhance productivity by minimizing fragmentation in Python-like software projects that require GPU performance and CUDA-like kernels.

Another important aspect is Mojo's interoperability with GPU vendor tools and ecosystems. In our early experience, debugging Mojo code is only supported by NVIDIA's Nsight Compute CLI (ncu) or the CUDA-GDB debugger (Mojo provides its LLDB-based debugger). Unfortunately, we were unable to find an officially supported tool to successfully profile with AMD's rocprof. Tooling is an important aspect of developing HPC software, and it is still an active area of development in Mojo.

2.2 Science Kernels in Mojo

Below is an overview of the selected science kernels and the implementations that we will use to assess Mojo's performance-portable capabilities on GPUs. We focus on kernels from the following proxy applications:

- **Seven-point stencil**: memory-bandwidth bound kernel in diffusion transport phenomena²
- **BabelStream**: memory transfer operations benchmarks [17]
- **miniBUDE**: in-silico molecular docking for protein prediction [49] (compute-bound kernel)
- **Hartree-Fock**: quantum many-body approximation [23] (compute-bound with atomics kernel)

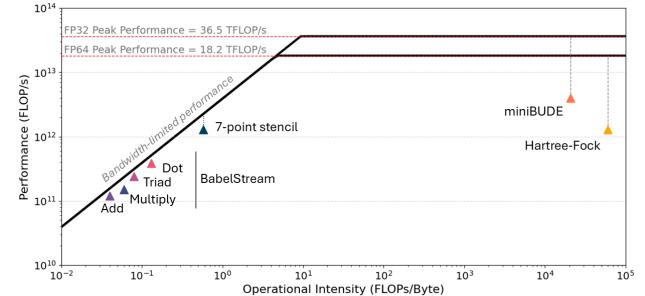


Figure 2: Roofline representation of the workloads implemented in the study using NVIDIA's CUDA and NSight on an NVIDIA H100 GPU.

These kernels provide initial coverage for memory and compute operations that are typical of codes used in many science domains. Figure 2 presents a roofline model, obtained with the NVIDIA NSight profiler, in which these kernels lie in the memory-and compute-bound regions for the NVIDIA H100 GPUs. As shown, the model confirms the coverage obtained through these scientific kernels. Because these kernels are well-established in HPC, CUDA and HIP C++ codes are readily available and functional for comparison. In the rest of this section, we provide a brief overview of each proxy application. We explain the nature of each kernel and the computational characteristics that drive performance. We also provide snippets of the Mojo code that resulted from our porting tasks.

²<https://github.com/amd/amd-lab-notes>

Seven-point stencil. As a fundamental kernel that results from applying the Laplacian operator on partial differential equations (PDEs) for modeling diffusion phenomena, the seven-point stencil is used in several science domains. Stencil calculations are an important class of PDE solvers on structured grids [15] and have been optimized for different hardware architectures, including GPUs [26, 28, 33]. The Mojo kernel implementation is presented in Listing 2 and is a straightforward port from AMD's original HIP baseline implementation. The latter is also translated into CUDA by using the same structure as AMD's HIP code.

```
alias precision = Float64
alias dtype = DType.float64
alias layout = Layout.row_major(L, L, L)
fn laplacian_kernel(f: LayoutTensor[mut=True, dtype, layout],
                    u: LayoutTensor[mut=False, dtype, layout],
                    nx: Int, ny: Int, nz: Int,
                    invhx2: precision, invhy2: precision,
                    invhz2: precision, invhxyz2: precision
                  ):
    var k = thread_idx.x + block_idx.x * block_dim.x
    var j = thread_idx.y + block_idx.y * block_dim.y
    var i = thread_idx.z + block_idx.z * block_dim.z

    if i > 0 and i < nx - 1 and j > 0 and j < ny - 1 and
       k > 0 and k < nz - 1:
        f[i,j,k] = u[i,j,k] * invhxyz2
        + (u[i-1, j, k] + u[i+1, j, k]) * invhx2
        + (u[i, j-1, k] + u[i, j+1, k]) * invhy2
        + (u[i, j, k-1] + u[i, j, k+1]) * invhz2
```

Listing 2: Seven-point stencil portable Mojo GPU implementation.

BabelStream. As a simple implementation of the Stream benchmark for GPUs, BabelStream measures the memory-bandwidth limits of five common array kernel operations—Copy, Multiply, Add, Triad, and Dot—each of which is measured independently. Listing 3 presents our initial efforts to port from the original CUDA and HIP codes to Mojo, highlighting how trivial this process can be for the first four operations because they allocate arrays exclusively on global memory. However, we expand on the use of shared block-memory capabilities in the Dot reduction operation. BabelStream provides the building blocks of several memory-bandwidth bound algorithms (e.g., conjugate gradients) that are a composition of these basic operations.

```
fn copy_kernel(a: UnsafePointer[Scalar[dtype]],
               c: UnsafePointer[Scalar[dtype]]):
    var i = block_dim.x * block_idx.x + thread_idx.x
    c[i] = a[i]

# add:      c[i] = a[i] + b[i]
# multiply: b[i] = scalar * c[i]
# triad:    a[i] = b[i] + scalar * c[i]

fn dot_kernel[size: Int](a: UnsafePointer[Scalar[dtype]],
                        b: UnsafePointer[Scalar[dtype]],
                        sums: UnsafePointer[Scalar[dtype]]):
    var tb_sum = stack_allocation[TBSize, Scalar[dtype]],
        address_space = AddressSpace.SHARED,
        ()()
    var i = block_dim.x * block_idx.x + thread_idx.x
    var local_tid = thread_idx.x

    threads_in_grid = block_dim.x * grid_dim.x
    while i < SIZE:
        tb_sum[local_tid] += a[i] * b[i]
        i += threads_in_grid

    var offset = block_dim.x // 2
    while offset > 0:
        barrier()
        if local_tid < offset:
            tb_sum[local_tid] += tb_sum[local_tid + offset]
        offset /= 2

    if local_tid == 0:
        sums[block_idx.x] = tb_sum[local_tid]
```

Listing 3: BabelStream portable Mojo GPU implementation.

miniBUDE. As a proxy application to the Bristol University Docking Engine (BUDE) code, miniBUDE's core kernel, fasten, simulates the computational workloads involved in predicting the structure and strength of the interaction of two molecules. This method is known as *in-silico molecular docking* and is used for drug discovery. The computational workload is driven by a few parameters: (i) poses per work-item (PPWI), (ii) work-group size, (iii) number of ligands, and (iv) number of proteins. The Mojo implementation is shown in Listing 4 and captures the same structure as the original C++ CUDA and HIP baseline codes. We found that Mojo still lacks support for plain-old-data allocations on GPU memory for the original Atom struct consisting of 3-Float32 and 1-Int elements. Our workaround consists of mapping this struct into a flattened array of 4-Float32 elements while casting back the last element as an Int inside the kernel. Although not exactly the same as the CUDA and HIP versions, the Mojo code retains its portability.

```
fn fasten_kernel[PPWI: Int](natlig: Int,
                           natpro: Int,
                           protein_molecule: UnsafePointer[Float32],
                           ligand_molecule: UnsafePointer[Float32],
                           transforms_0: UnsafePointer[Float32],
                           ...
                           transforms_5: UnsafePointer[Float32],
                           etotals: UnsafePointer[Float32],
                           global_forcefield: UnsafePointer[Float32],
                           num_transforms: Int):

    var ix = block_idx.x * block_dim.x * PPWI + thread_idx.x
    if ix >= num_transforms:
        ix = num_transforms - PPWI

    var etot = SIMD[dtype, PPWI]()
    var transform = InlineArray[Vec4f32, PPWI * 3](uninitialized=True)

    var lsz = block_dim.x
    for i in range(PPWI):
        var index = ix + i * lsz
        sx: Float32 = sin(transforms_0[index]) ...
        transform[i * 3].x = cy * cz
    # Loop over ligand atoms
    while (True):
        ...
        for i in range(PPWI):
            # Loop over protein atoms
            while (True):
                for i in range(PPWI):
                    # Write energy results
                    var td_base = block_idx.x * block_dim.x * PPWI + thread_idx.x
                    if td_base < num_transforms:
                        for i in range(PPWI):
                            etotals[td_base + i * block_dim.x] = etot[i] * Half
```

Listing 4: miniBUDE Mojo kernel structure.

Hartree-Fock. This kernel provides a solution to the Hartree approximation of the Schrödinger equation for a quantum many-body system at ground state. Wavefunctions are approximated by using a Slater determinant eliminating higher-order terms similar to density functional theory. The key kernel presented in Listing 5 performs dense symmetric matrix calculations of the electron-repulsion term of the Fock operator using integrals over Gaussian functions. The kernel is composed of (i) one parallelizable loop proportional to the fourth power of the number of atoms (natoms) with nested loops over the Gaussian components (usually three) and (ii) six atomic operations over two square matrices of size natoms × natoms representing the density and Fock terms. Hence, the kernel is compute-bound with limited parallelism because of the atomic operations. The Mojo, HIP, and CUDA implementations are based on the original Fortran + OpenMP implementation [23], and porting is a straight-forward process.

```

fn hartree_fock(ngauss: Int, schwarz: UnsafePointer,
    xptn: UnsafePointer,
    coef: UnsafePointer,
    geom: LayoutTensor[mut=True, dtype, geom_layout],
    dens: LayoutTensor[mut=True, dtype, layout],
    fock: LayoutTensor[mut=True, dtype, layout]):
    var ijk1 = block_idx.x * block_dim.x + thread_idx.x
    # 4-levels nested ngauss loops
    for ib in range(ngauss): ...
        for jb in range(ngauss): ...
            for kb in range(ngauss): ...
                for lb in range(ngauss): ...
                    eri += Float64(dkl + fdt * pow(aijkl, 0.5))
    # 6 atomic Fock matrix updates
    _ = Atomic.fetch_add(fock.ptr.offset(i * natoms + j),
        rebind[Scalar[dtype]](dens[k, 1] * eri * 4.0))
    ...
    _ = Atomic.fetch_add(fock.ptr.offset(j * natoms + 1),
        rebind[Scalar[dtype]](dens[i, k] * eri * -1))

```

Listing 5: Basic Hartree-Fock Fortran kernel structure.

3 Methodology

Our initial effort is a straightforward evaluation of the novel performance-portable capabilities of Mojo when running on GPUs. Table 1 lists the hardware specifications of the NVIDIA H100 and AMD MI300A GPUs used in this study.

Table 1: GPU Hardware Used in This Study

GPU – Memory	Theoretical Peaks		
	Bandwidth GB/s	FP32 TFLOP/s	FP64 TFLOP/s
NVIDIA H100 NVL – 94 GB	3,900	60.0	30.0
AMD MI300A – 128 GB HBM3	5,300	122.6	61.3

Each application is ported from a baseline implementation that uses the vendor-specific CUDA (NVIDIA) or HIP (AMD) programming models. Mojo’s portable syntax shown in Listings 2, 3, and 5 is compared against available CUDA and HIP implementations. For each case, we use metrics that reflect a given application’s purpose.

Seven-point stencil. We use an effective memory bandwidth metric from the original baseline to represent the cell data advanced during a simulation step in GB/s. For simplicity, we use a constant L size in each direction: x , y , and z . The metric is defined in Eq. 2 as a function of L and the size of the element type ($T = \text{Float32}$ or Float64) used in our runs.

$$\begin{aligned}
fetch_size_{effective} &= [L^3 - 8 - 12(L - 2)] \cdot \text{sizeof}(T) \\
write_size_{effective} &= (L - 2)^3 \cdot \text{sizeof}(T) \\
bandwidth_{effective} &= \frac{(fetch_size + write_size)_{effective}}{\text{kernel_time}}
\end{aligned} \quad (1)$$

BabelStream. We use an effective memory bandwidth metric based solely on the size of the arrays (vector sizes) and the number of arrays for each operation. This method is summarized in Eq. 2 for each of the fundamental operations. Notably, although all operations allocate arrays on global device memory, the Dot operation exploits block-level shared memory for faster first-level reduction operations.

$$\begin{aligned}
bandwidth_{Array} &= \frac{\text{sizeof}(T) \cdot \text{Vector_size}}{\text{kernel_time}} \\
bandwidth_{Copy} &= 2 \cdot bandwidth_{Array} \\
bandwidth_{Mul} &= 2 \cdot bandwidth_{Array} \\
bandwidth_{Add} &= 3 \cdot bandwidth_{Array} \\
bandwidth_{Triad} &= 3 \cdot bandwidth_{Array} \\
bandwidth_{Dot} &= 2 \cdot bandwidth_{Array}
\end{aligned} \quad (2)$$

miniBUDE. Given the compute-bound nature of miniBUDE’s fasten kernel, the performance metric measures the floating-point operations per kernel execution time (GFLOP/s). We obtain this metric through a calculation from the original baseline:

$$\begin{aligned}
ops_{workgroup} &= 28 \text{ PPWI} + \\
&nligands \cdot [2 + 18 \text{ PPWI} + nproteins \cdot (10 + 30 \text{ PPWI})] \\
total_{ops} &= ops_{workgroup} \frac{poses}{PPWI} \\
miniBUDE - total_{gflops} &= \frac{total_{ops}}{time_{kernel}} \cdot 1E - 9
\end{aligned} \quad (3)$$

Hartree-Fock. The figure of merit for this kernel is directly related to speedups obtained by porting the original algorithm [23] to GPUs. To compare across languages, we measure kernel wall-clock times directly without further transformations. We select available benchmarks for systems of helium atoms up to 256 atoms and 3 Gaussian functions per atom. As shown in Listing 5, parallelization is heavily limited by the atomic operations inside this kernel. Our objective is to measure the impact of this pattern on Mojo’s GPU-portable atomic operations capabilities.

For all the selected cases, we can establish that our methodology balances performance and portability for a fair assessment of the Mojo GPU code. Our methodology can be summarized as follows:

- Warm-up steps: For all codes, we discarded the first step in our measurements to reduce JIT or caching effects.
- Variability: We used dedicated hardware and collected several runs, at least 100, in our experiments to capture variability in our results. Kernel times have been verified with profiling tools.
- One-to-one comparisons: GPU code was not fine-tuned and matches any existing baseline configuration (e.g., number of blocks, threads for GPUs). For BabelStream, we use the optimized vendor-specific CUDA version, whereas our Mojo port follows a hybrid vendor-independent version to ensure portability and the best possible trade-offs on NVIDIA and AMD GPUs.
- Profiling: We profiled cases of interest in which Mojo can either surpass the vendor programming model or when differences are consistent.
- Results consistency: We used color codes for each programming model and a uniform figure of merit across all of the figures that present the results of application runs.
- JIT vs. AOT: Owing to the compile-time nature of Mojo, we did not observe meaningful differences between JIT and AOT kernel executions. Profiling of Mojo code with NVIDIA’s

Nsight CLI (ncu) and AMD's rocprof is only possible with AOT-compiled Mojo code.

4 Results

This section presents the performance results from our Mojo implementations and the available vendor-specific CUDA and HIP codes.

Seven-point stencil. Figure 3 shows raw data measurements from the obtained bandwidth (defined in Eq. 1) and a comparison of Mojo versus CUDA on an NVIDIA H100 (3a) and HIP on an AMD MI300A (3b). The code comparisons used two large problem sizes, 512 and 1024, while using a different number of blocks on the first grid dimension to achieve maximum performance. Notably, tuning on the second or third dimension did not result in higher performance. As shown, Mojo is fairly competitive for both single and double precision, but is slightly slower on NVIDIA's H100, averaging 87% of the CUDA performance. This same variability is reflected in Mojo and the vendor-code runs on both GPUs but is more noticeable on the NVIDIA H100 when using double precision, hence our focus is on the overall differences across the raw data. Results on the MI300A are essentially on par with the AMD HIP implementation, including capturing outlier measurements. Interestingly, maximum bandwidth on the MI300A is reached with the same configuration, $L = 512$ on a $512 \times 1 \times 1$ grid as recommended on the MI250X from the original HIP code.³

To further understand differences between Mojo and CUDA, we profiled the codes with NVIDIA's ncu. Results are presented in Table 2 for the two cases shown in Fig. 3a. These cases were selected due to their more significant differences and overall slower performance. Overall, the CUDA-generated code exhibits more efficient use of memory resources, which is the most relevant aspect owing to the memory-bound nature of the kernel. Performance differences arise from the use of registers at the local cache level; in fact, global memory loads and stores on both models are consistent with the expected 7 reads and 1 write for each cell value. Moreover, Mojo kernels use more registers-per-thread (24) than CUDA (21) for the same operation, which explains the performance differences in L-caches and kernel duration. Notably, the arithmetic intensity (ai) and memory performance shown in the roofline analysis are consistent with our hypothesis. Hence, there is room to further optimize the Mojo codes at the lower memory cache levels, even for a simple but widely used GPU stencil kernel.

BabelStream. Figure 4 shows the obtained bandwidth metrics from Eq. 2 for the fundamental Copy, Mul, Add, Triad, and Dot operations and compares the Mojo GPU portable code with vendor implementations for a large vector size of $2^{25} = 33,554,432$. Surprisingly, except for the Dot operation, the Mojo implementation is slightly faster than CUDA, whereas Mojo's results on AMD are nearly indistinguishable from the HIP version. This requires further analysis, but it is somewhat expected because the Mojo portable Dot product is the only operation that is not identical to the CUDA and HIP optimized versions involving block-level shared memory. The baseline CUDA version provides heuristics to calculate block size for NVIDIA-specific GPUs based on multiprocessors counts.

³<https://github.com/amd/amd-lab-notes>

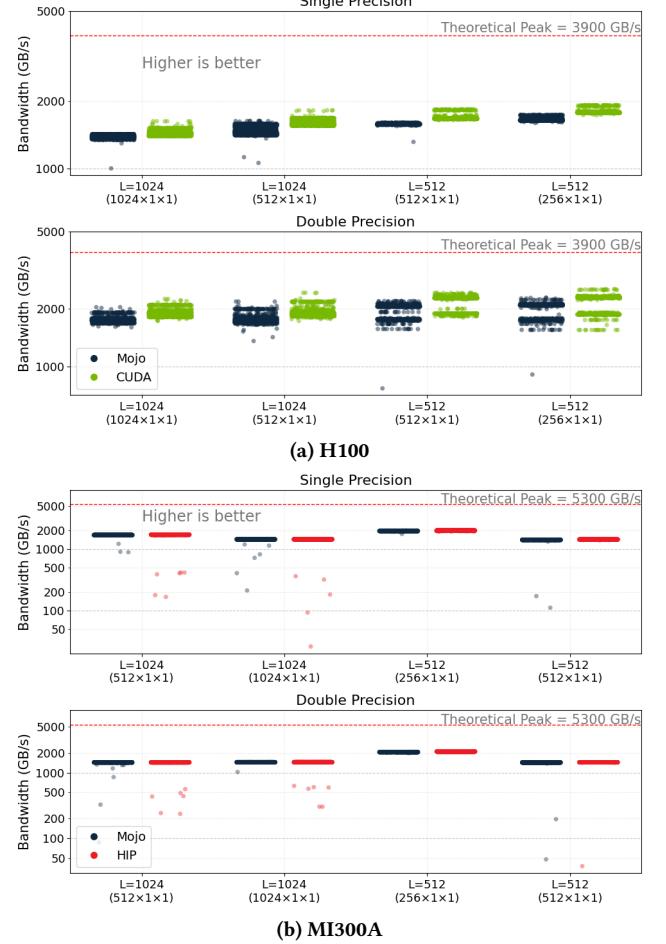


Figure 3: Mojo performance vs. CUDA and HIP for seven-point stencil kernels bandwidth.

Table 2: Seven-Point Stencil Mojo vs. CUDA NCU Profiling Metrics

Nsight Compute CLI (ncu) metric	Double Precision L=512 (512x1x1)		Single Precision L=1024 (1024x1x1)	
	Mojo	CUDA	Mojo	CUDA
Duration (ms)	1.10	0.96	8.74	7.21
Throughputs (%)				
- Compute SM	81.41	51.96	79.8	53.7
- Memory	67.98	76.72	37.7	43.9
L1 ai (FLOP/byte)	0.14		0.24	
L2 ai (FLOP/byte)	0.26		0.51	
L3 ai (FLOP/byte)	0.62		1.24	
L1-3 Perf (FLOP/s)	1.20 E12	1.38 E12	1.22 E12	1.48 E12
Registers	24	21	26	20
Load Global (LDG)	7		7	
Store Global (STG)	1		1	

We decided to keep the Mojo implementation portable and follow a hybrid approach between the CUDA and HIP versions to maximize performance portability. All the codes are provided in the

Appendix. Unlike the seven-point stencil cases, BabelStream runs show less variability, which might be related to its simpler 1-to-1 memory-access that uses a 1D computational grid.

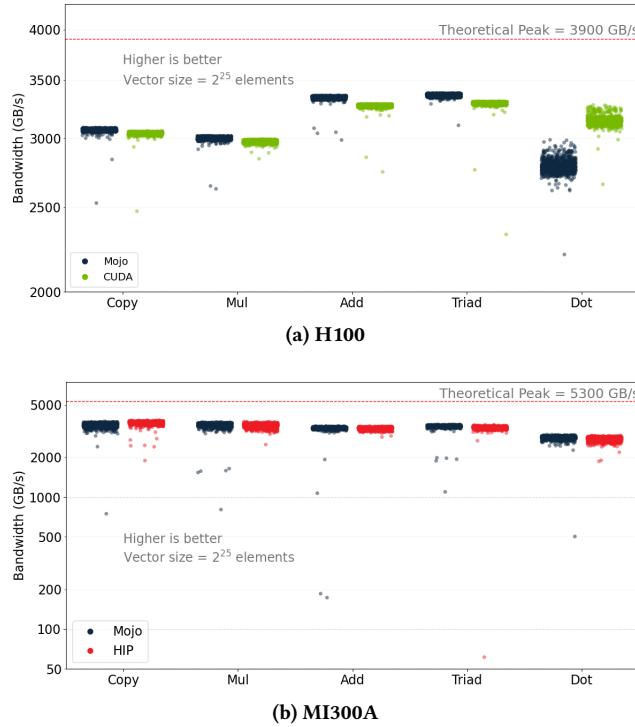


Figure 4: Mojo performance vs. CUDA and HIP for BabelStream’s kernels bandwidth.

We ran additional profiling analysis to understand the differences on the NVIDIA H100, as shown in Table 3. We confirmed that the more efficient use of low-level resources leads to slightly higher memory throughput and lower compute throughput, which is desired in memory-bandwidth workloads. Also, the kernel run characteristics, such as arithmetic intensity (FLOP/byte), performance (FLOP/s), number of registers per thread, and global memory activity, are consistent with the CUDA implementation.

Table 3: BabelStream Mojo vs. CUDA NCU Profiling Metrics

Nsight Compute CLI (ncu) metric	Copy		Mul		Add		Dot	
	Mojo	CUDA	Mojo	CUDA	Mojo	CUDA	Mojo	CUDA
Duration (ms)	0.202	0.205	0.203	0.208	0.264	0.269	0.215	0.168
Throughputs (%)								
- Compute SM	16.3	28.6	18.2	28.2	15.9	27.3	51.1	11.4
- Memory	69.7	68.9	69.2	68.0	81.7	80.5	69.9	87.6
L1 ai (FLOP/byte)	–	–	0.06	–	0.04	–	0.13	–
L2 ai (FLOP/byte)	–	–	0.08	–	0.05	–	0.14	0.13
L3 ai (FLOP/byte)	–	–	0.12	–	0.06	–	0.14	0.13
L1-3 Perf (FLOP/s)	–	–	1.64 E11	1.61 E11	1.26 E11	1.24 E11	3.5 E11	4.01 E11
Registers	16	–	16	–	16	–	26	20
Load Global (LDG)	1	–	1	–	2	–	2	–
Store Global (STG)	1	–	1	–	1	–	1	–

Figure 5 shows a deeper side-by-side assembly-level analysis that uses NVIDIA’s Nsight tool for the Triad kernel. This kernel, which is not shown in Table 3, exhibits performance similar to the

Add kernel, as expected. We made three important observations: (i) Mojo produces fewer load constant operations than CUDA and uses constant memory without further annotations in the code, (ii) Mojo reports fewer active live registers but reports more compute operations (IADD3) during the kernel’s main operation, and (iii) global memory loads (LDG) and global stores (STG) are consistent among codes. The presented analysis is enough to understand the performance and throughput differences, and any further analysis would be beyond the scope of our performance-portability focus.

miniBUDE. Figures 6 and 7 show the performance obtained in GFLOP/s from Eq. 3 in our experiments for different PPWI sizes and two work-group sizes (wg) (8 and 64) to vary the computational workload of the small bm1 benchmark. We observe less variability for all cases, so we show only the average value obtained from our runs. We also present CUDA and HIP runs without fast-math because this compute-bound workload is sensitive to such optimizations, and Mojo does not provide this option. Mojo did not reach the same performance levels as it did in previous cases, but it follows a similar trend. Mojo outperforms CUDA for a small PPWI and wg and is somewhere in between the fast-math optimized and non-optimized versions of CUDA on the NVIDIA H100. The same Mojo code underperforms both the HIP fast-math optimized and non-optimized versions on the AMD MI300A, similar to the small PPWI and wg case shown in Figure 7a.

Hartree–Fock. Our experience confirms the impact of the atomic operations preventing more parallelization. Because of this noticeable effect, we provide only the wall-clock times as a function of the number of atoms, a , and Gauss functions per atom, n_{gauss} , when comparing Mojo with CUDA and HIP (Table 4). Surprisingly, Mojo is $2.5\times$ faster than CUDA on operations of up to 256 in size, but performance decreases dramatically for the 1024 operation. Rather than drawing general conclusions, we believe that further analysis is necessary to understand these differences. The opposite happened on AMD GPUs: Mojo largely underperforms the HIP implementation in all cases, which leads us to believe that we are still hitting a corner case in the language. Thus, as Mojo continues to mature on AMD GPUs, these capabilities are expected to improve at the lower levels.

Table 4: Hartree–Fock Mojo vs. CUDA and HIP Metrics

Kernel execution duration (ms)	NVIDIA H100		AMD MI300A	
	Mojo	CUDA	Mojo	HIP
a=1024 ngauss=6	147,250	2,652	–	846
a=256 ngauss=3	187	472	25,266	178
a=128 ngauss=3	21	53	2,765	23
a=64 ngauss=3	3	7	436	4

4.1 Performance Portability Metric

Given the results across the two selected GPU platforms, we apply a performance portable-metric ($\bar{\Phi}$) that was discussed in several recent publications [38, 39, 45, 46]. We use the definition of $\bar{\Phi}$ for

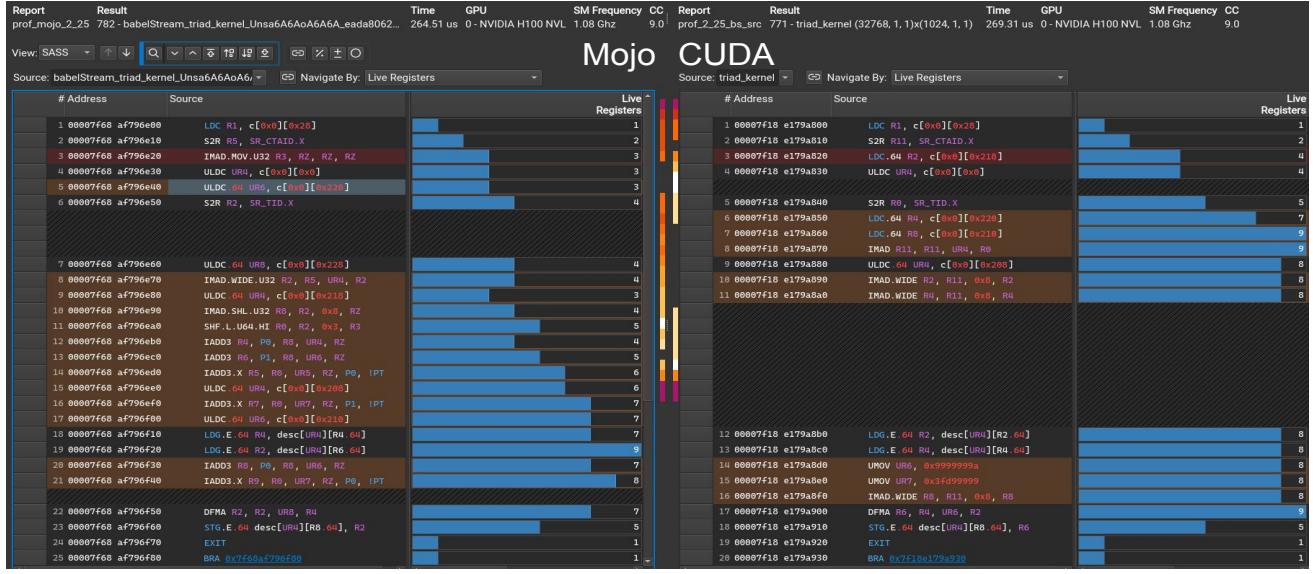
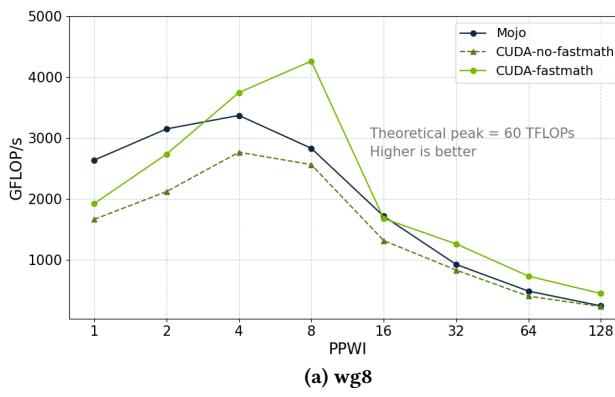
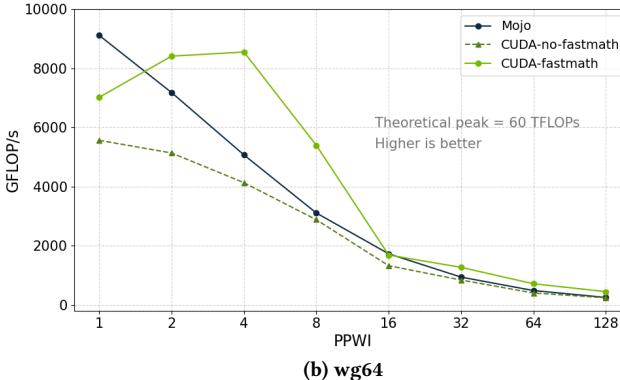


Figure 5: Mojo vs. CUDA generated assembly (SASS) from NVIDIA’s Nsight for BabelStream’s Triad kernel.



(a) wg8



(b) wg64

Figure 6: Mojo performance vs. CUDA for miniBUDE kernels on NVIDIA H100.

“application efficiency” as the arithmetic mean of an application’s performance efficiency observed across a set of platforms from the same architecture class. Thus, the metric is defined as the arithmetic mean of the best possible result:

$$\bar{\Phi}_{Mojo} = \frac{\sum_{i \in T} e_i(a)}{|T|} \quad (4)$$

$$e_i(a) = \frac{Mojo_{perf-i}}{vendor(CUDA/HIP)_{perf-i}},$$

where e is the efficiency of the Mojo performance metric defined as the ratio of the Mojo performance metric to the vendor-specific CUDA and HIP implementations. T is the subset of runs for a particular proxy and architecture class (e.g., GPUs only). The results from applying this metric are shown in Table 5.

The most important takeaway is that Mojo scores very high for seven-point stencil and BabelStream memory-bandwidth bound workloads. Gaps exist in the compute-bound miniBUDE for both vendors, and these gaps are associated with the need for further optimization (e.g., fast-math). Hartree–Fock results come from a special case with atomic operations in which Mojo outperforms CUDA but is far from achieving HIP’s performance. Additionally, the high value of the proposed performance methodology for the latter case can be misleading because it is the product of canceling out the outperforming of one vendor with the underperforming of another.

4.2 Discussion and Observations

Next, we summarize our observations and findings from the development and evaluation process used to analyze the novel Mojo language.

Observation 1: Performance portability. Without any further optimizations, Mojo’s single GPU code performance is on par with

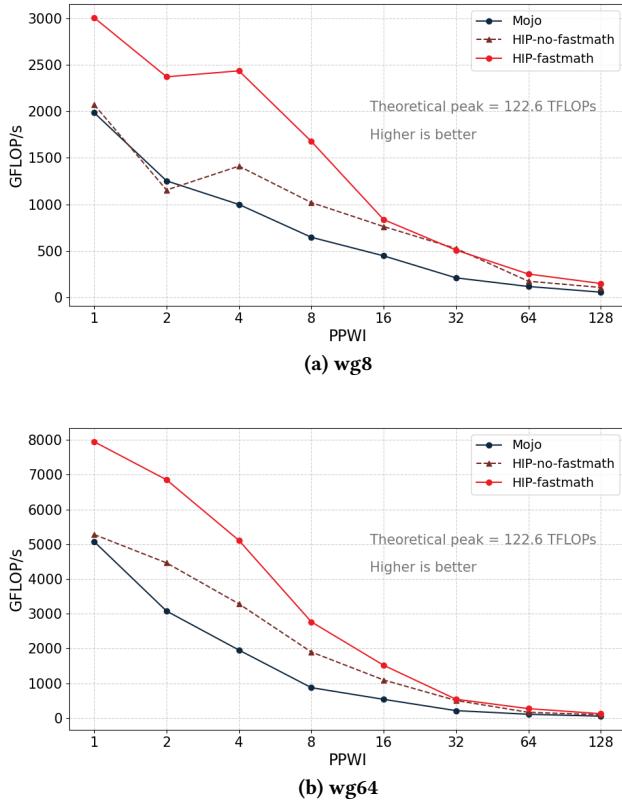


Figure 7: Mojo performance vs. HIP for miniBUDE kernels on AMD MI300A.

AMD’s HIP GPU code in all of our experiments for memory-bound kernels. Performance differences still arise in compute-bound kernels mainly due to the lack of fast-math capabilities. For the seven-point stencil operation, gaps exist between Mojo and NVIDIA’s CUDA because of CUDA’s additional register-level optimizations. Surprisingly, BabelStream operations (except for Dot) can be further optimized by Mojo to provide a slight performance benefit versus CUDA. This could be explained by Mojo’s MLIR optimizations, which reduce the number of required register memory operations, as confirmed by our profiler analysis. In the case of miniBUDE, there is significant sensitivity to fast-math compilation, which results in noticeable performance gaps in the Mojo implementation. We look forward to this capability being available for these workloads in the near future. Overall, our results suggest that Mojo can help narrow the performance gap in the Python ecosystem, but it still has gaps and a steep learning curve for low-level programming concepts.

Observation 2: Compile time and HPC. Mojo’s compile-time nature could be a learning curve for HPC practitioners. Although HPC has been largely dominated by codes that are provided with performance-critical information at run time and do not necessarily exploit this information at compile time, Mojo might present an opportunity to explore future compiler optimizations via MLIR through AOT or JIT, as demonstrated in the literature [3, 21, 22, 25].

Table 5: Mojo Performance Portability Metric

Mojo Efficiency	NVIDIA H100	AMD MI300A
7-point stencil		
FP32	0.82	1.00
FP64	0.87	1.00
$\bar{\Phi} = 0.92$		
BabelStream		
Copy	1.01	1.00
Mul	1.02	1.00
Add	1.01	1.00
Triad	1.01	1.00
Dot	0.78	1.00
$\bar{\Phi} = 0.96$		
miniBUDE		
PPWI=8 wg=8	0.82	0.38
PPWI=4 wg=64	0.59	0.38
$\bar{\Phi} = 0.54$		
Hartree-Fock		
a=1024 ngauss=6	17E-3	-
a=256 ngauss=3	2.52	7E-3
a=128 ngauss=3	2.52	8E-3
a=64 ngauss=3	2.33	8E-3
$\bar{\Phi} = 0.92$		

This connection still needs to be defined, and run-time dynamic allocation is still a desired feature—even at the expense of not fully exploiting MLIR optimization.

Observation 3: Python and HPC interoperability. Mojo offloads non-performance-critical code to the rich Python ecosystem. It is still unknown how this strategy will interoperate with an HPC stack. In particular, Mojo’s interoperability with MPI (Message Passing Interface) is an open question given that Python’s mpi4py is well-established [14] and Python’s interoperability is orthogonal to the compile-time nature of Mojo.

Observation 4: Libraries and ecosystem. Mojo is still a low-level programming alternative to NVIDIA’s CUDA and AMD’s HIP. Many of the HPC codes rely on interactions with highly optimized high-level libraries, and support for this functionality (e.g. portable BLAS operations) is still being defined for a wider adoption of the language. For the tools, it was extremely beneficial for our analysis, and for HPC code in general, that Mojo can generate a compiled executable that is out-of-the-box compatible with the NVIDIA ncu and AMD rocprof ecosystems without modifying any source code.

5 Related Work

Mojo. Owing to Mojo’s novelty, there are very few studies of the language and its capabilities. Akre et al. [1] provided a recent

overview of the language’s potential for different areas of computing and a curated list of libraries and frameworks available on GitHub. Their paper presents a summary of opportunities for the language and initial benchmarks that show orders-of-magnitude speedups against pure Python and Numpy code. Raihan et al. [50] proposed Mojobench to close the gaps for large language model-based Mojo code generation given the scant training data available for such a novel programming language. Huang et al. [29] introduced MojoFrame, the first dataframe library to exploit performance characteristics of the language, which highlights the novel nature of Mojo’s ecosystem.

In terms of high-level languages that target scientific computing, significant efforts have been made in the Python and Julia programming communities to close the gaps between high productivity and performance portability.

Python. Several efforts aim to enable performance-portable GPU programming without resorting to wrappers for portable code in C or C++. At the vendor level, NVIDIA’s recent native support for Python via `cuda-python`⁴ was followed by AMD’s support for interoperability, similar to HIP and CUDA, with the introduction of `hip-python-as-cuda`.⁵ Mattson et al. introduced PyOMP [41], an implementation of OpenMP for the Numba [34] JIT compiler, bypassing the Python global interpreter lock. This work was recently extended to GPUs by using OpenMP offload [24] and then benchmarked with the HeCBench [30] kernels. Similarly, Piñeiro and Pichel proposed OMP4Py [48] for a pure Python implementation of OpenMP directives combining CPU multi-core with MPI parallelization. Recently, PyOpenCL [32] and Cupy [43] were expanded to provide support for AMD GPUs in addition to the well-supported NVIDIA ecosystem by passing C or C++ custom kernel code for compilation through a strings interface. Similarly, pyKokkos [4] provides a performance-portable interface between Python and the Kokkos library [54] to generate C++ Kokkos kernels and language bindings for interoperability. More recently, Briand et al. [9] compared the performance of Julia and several programming models, including a Kokkos.jl wrapper, against C++ Kokkos and showed that the performance is on par with their selected computational fluid dynamics kernels.

Julia. The open-source LLVM-based Julia language also addresses fragmentation in the software development process, mainly targeting scientific computing and AI. Churavy et al. [13] summarized efforts to establish a unifying ecosystem for large-scale computing. Valero-Lara et al. [55] introduced JACC, a recently developed high-level, performance-portable library that abstracts away low-level, hardware-specific options on CPUs/GPUs while exhibiting nearly zero overhead versus Julia’s vendor-specific CUDA.jl [7], AMDGPU.jl [51], and oneAPI.jl [6] back ends. Similarly, Churavy et al. developed KernelAbstractions.jl [12], which enables portable GPU kernel programming by using a fine-granularity parallelization approach similar to the CUDA and HIP models. Lin and McIntosh-Smith [37] analyzed the performance of Julia’s programming models for BabelStream and miniBUDE on CPUs and GPUs. Their study showed that Julia is on par or slightly behind compiled languages

⁴<https://developer.nvidia.com/cuda-python>

⁵https://rocm.docs.amd.com/projects/hip-python/en/latest/user_guide/2_cuda_python_interop.html

on several CPUs and GPUs. Godoy et al. [27] evaluated the performance portability of Julia, Numba, and Kokkos programming models across CPUs and NVIDIA and AMD GPUs. The work demonstrated that Numba was lacking in performance and portability on AMD GPUs, but Julia’s GPU capabilities showed significant potential—even with some gaps in performance for a simple matrix multiplication kernel. De la Calle et al. [16] introduced the Juliana tool to add GPU performance portability in Julia by porting CUDA.jl code to KernelAbstractions.jl, which introduces a small overhead on several kernels, including miniBUDE and BabelStream.

Other efforts. Efforts to narrow gaps of productivity, performance, and portability have long existed in HPC. The Defense Advanced Research Projects Agency’s High Productivity Computing Systems [19] program funded three languages—Chapel [10], IBM’s X10 [52], and Sun’s Fortress [2]—and favored a partitioned global address space model. Only Chapel remains in active development as it evolves toward vendor-neutral GPU programming [31]. On the proprietary side, parallel Matlab [11] is an early effort to leverage the popular Matlab language and parallel computing.

Hence, Mojo aligns with the search for productive, vendor-neutral programming models to drive future extremely heterogeneous computing systems [57].

6 Conclusions

We present our early experiences with the novel Mojo programming language. In particular, we describe the added capability to write performance-portable kernels for NVIDIA and AMD GPUs. We targeted four common science kernels—(i) seven-point stencil (memory-bound), (ii) BabelStream (memory-bound), (iii) Hartree-Fock (compute-bound + atomic), and (iv) miniBUDE (compute-bound)—and compared the same Mojo code with CUDA and HIP on modern NVIDIA H100 and AMD MI300A GPUs, respectively.

Our results indicate that the performance of the Mojo low-level implementation is on par with AMD’s HIP for memory-bandwidth bound cases. We observed lower performance when comparing Mojo against CUDA-specific seven-point stencil (89%) and the shared-memory operations in the Dot kernel (78%) in BabelStream. On the plus side, Mojo is highly optimized for simple 1D array operations in BabelStream and even slightly surpassed CUDA’s performance thanks to fewer cached memory operations. The differences between Mojo and the CUDA implementation were observed by comparing profiling information from vendor tools. These differences need to be understood at lower compiler levels to avoid sacrificing the potential of Mojo’s portability. For the compute-bound workloads in miniBUDE and Hartree-Fock, which included atomic operations, Mojo still exhibits several performance gaps versus CUDA and HIP. The timeline for adding the fundamental `fast-math` optimizations to Mojo is still an unknown, as these optimizations are critical to the tested compute-bound kernels. Additionally, atomic operations must be further analyzed and better understood given Mojo’s peculiar results of overperforming CUDA and underperforming HIP in these tasks.

Based on the performance results, we applied a performance-portability metric, but we expect the actual performance portability to evolve as the language matures, adds features to fully leverage

MLIR, and integrates feedback from the open-source HPC community.

Unlike other programming models that attempt to speed up Python or compile to LLVM from the ground up (e.g., Julia, Rust), Mojo closes performance gaps by offering a superset language that combines rich compile-time metaprogramming via MLIR, performance portable GPU programming capabilities in their standard library, and run-time interoperability with the Python ecosystem. Our expectation is that Mojo will continue to evolve under this programming paradigm—with the promise of becoming open-source by 2026⁶—as it moves toward becoming a viable HPC + AI programming language.

Acknowledgments

This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research's Computer Science Competitive Portfolios program, MAGMA/Fairbanks project; and the Next Generation of Scientific Software Technologies program, PESO and S4PST projects. This research used resources of the Oak Ridge Leadership Computing Facility and the Experimental Computing Laboratory at the Oak Ridge National Laboratory, which are supported by the Office of Science of the US Department of Energy under Contract No. DE-AC05-00OR22725. This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Science Undergraduate Laboratory Internships Program (SULI). WFG and TM would like to thank Alex Smith from the University of Wisconsin-Madison for providing the CUDA and HIP ports of Hartree-Fock.

References

- [1] Parth Dhananjay Akre and Utkarsha Pacharaney. 2025. A Comprehensive Review of Mojo: A High-Performance Programming Language. In *2025 6th International Conference on Mobile Computing and Sustainable Informatics (ICMCSI)*. 861–867. <https://doi.org/10.1109/ICMCSI64620.2025.10883176>
- [2] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt, Joao Dias, Carl Eastlund, et al. 2005. The Fortress language specification. *Sun Microsystems* 139, 140 (2005), 116.
- [3] Aksel Alpay and Vincent Heuveline. 2025. Adaptivity in AdaptiveCpp: Optimizing Performance by Leveraging Runtime Information During JIT-Compilation. In *Proceedings of the 13th International Workshop on OpenCL and SYCL (IWOC'25)*. Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. <https://doi.org/10.1145/3731125.3731127>
- [4] Nader Al Awar, Neil Mehta, Steven Zhu, George Biros, and Milos Gligoric. 2022. PyKokkos: performance portable kernels in Python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings* (Pittsburgh, Pennsylvania) (ICSE '22). Association for Computing Machinery, New York, NY, USA, 164–167. <https://doi.org/10.1145/3510454.3516827>
- [5] David A. Beckingsale et al. 2019. RAJA: Portable Performance for Large-Scale Scientific Applications. In *IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 71–81. <https://doi.org/10.1109/P3HPC49587.2019.00012>
- [6] Tim Besard. 2025. *oneAPI.jl*. <https://doi.org/10.5281/zenodo.14931098>
- [7] Tim Besard, Christophe Foket, and Bjorn De Sutter. 2019. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (2019), 827–841. <https://doi.org/10.1109/TPDS.2018.2872064>
- [8] Jeff Bezanson et al. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev* 59, 1 (Jan. 2017), 65–98. <https://doi.org/10.1137/141000671>
- [9] Luc Briand, Hervé Jourdren, and Marc Pérache. 2025. Julia versus Kokkos for performance portable Cartesian CFD solvers on heterogeneous architectures. *The International Journal of High Performance Computing Applications* 39, 4 (2025), 481–501. <https://doi.org/10.1177/10943420251341179>
- [10] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312. <https://doi.org/10.1177/1094342007078442>
- [11] R. Choy and A. Edelman. 2005. Parallel MATLAB: Doing it Right. *Proc. IEEE* 93, 2 (2005), 331–341. <https://doi.org/10.1109/JPROC.2004.840490>
- [12] Valentin Churavy. 2025. *KernelAbstractions.jl*. <https://doi.org/10.5281/zenodo.14724294>
- [13] Valentin Churavy et al. 2022. Bridging HPC Communities through the Julia Programming Language. *arXiv:2211.02740 [cs.DC]*
- [14] Lisandro Dalcín, Rodrigo Paz, Mario Storti, and Jorge D'Elia. 2008. MPI for Python: Performance improvements and MPI-2 extensions. *J. Parallel and Distrib. Comput.* 68, 5 (2008), 655–662. <https://doi.org/10.1016/j.jpdc.2007.09.005>
- [15] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. 2008. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. 1–12. <https://doi.org/10.1109/SC.2008.5222004>
- [16] Enrique de la Calle and Carlos Garcia. 2025. Evaluation of Juliana Tool: A translator for Julia's CUDA.jl code into KernelAbstraction.jl. *Future Generation Computer Systems* (2025), 107813. <https://doi.org/10.1016/j.future.2025.107813>
- [17] Tom Deakin et al. 2018. Evaluating attainable memory bandwidth of parallel programming models via BabelStream. *International Journal of Computational Science and Engineering* 17, 3 (2018), 247–262.
- [18] Tom Deakin and Simon McIntosh-Smith. 2020. Evaluating the performance of HPC-style SYCL applications. In *International Workshop on OpenCL*. 1–11.
- [19] Jack Dongarra, Robert Graybill, William Harrod, Robert Lucas, Ewing Lusk, Piotr Luszczek, Janice Mcmahon, Allan Snavely, Jeffrey Vetter, Katherine Yelick, Sadaf Alam, Roy Campbell, Laura Carrington, Tzu-Yi Chen, Omid Khalili, Jeremy Meredith, and Mustafa Tikir. 2008. DARPA's HPCS Program: History, Models, Tools, Languages. In *Advances in COMPUTERS*. Advances in Computers, Vol. 72. Elsevier, 1–100. [https://doi.org/10.1016/S0065-2458\(08\)00001-6](https://doi.org/10.1016/S0065-2458(08)00001-6)
- [20] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience* 15, 9 (2003), 803–820.
- [21] Alexix Engelke and Martin Schulz. 2020. Robust Practical Binary Optimization at Run-time using LLVM. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. 56–64. <https://doi.org/10.1109/LLVMHPCHiPar51896.2020.00001>
- [22] Hal Finkel, David Poliakoff, Jean-Sylvain Camier, and David F. Richards. 2019. ClangJIT: Enhancing C++ with Just-in-Time Compilation. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. 82–95. <https://doi.org/10.1109/P3HPC49587.2019.00013>
- [23] Graham D Fletcher et al. 2021. Basic Hartree-Fock Proxy Application. <https://doi.org/10.11578/dc.20210517.2>
- [24] Giorgis Georgakoudis, Todd A Anderson, Stuart Archibald, Bronis de Supinski, and Timothy G Mattson. 2024. *PyOMP: Programming GPUs with OpenMP and Python*. Technical Report. Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States).
- [25] Giorgis Georgakoudis, Konstantinos Parasiris, and David Beckingsale. 2025. Proteus: Portable Runtime Optimization of GPU Kernel Execution with Just-in-Time Compilation. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) (CGO '25). Association for Computing Machinery, New York, NY, USA, 507–522. <https://doi.org/10.1145/3696443.3708939>
- [26] William F. Godoy and Xu Liu. 2011. Introduction of Parallel GPGPU Acceleration Algorithms for the Solution of Radiative Transfer. *Numerical Heat Transfer, Part B: Fundamentals* 59, 1 (2011), 1–25. <https://doi.org/10.1080/10407790.2010.541359>
- [27] William F Godoy, Pedro Valero-Lara, T. Elise Dettling, Christian Trefftz, Ian Jorquera, Thomas Sheehy, Ross G Miller, Marc Gonzalez-Tallada, Jeffrey S Vetter, and Valentin Churavy. 2023. Evaluating performance and portability of high-level programming models: Julia, Python/Numba, and Kokkos on exascale nodes. In *2023 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 373–382. <https://doi.org/10.1109/IPDPSW59300.2023.00068>
- [28] Justin Holewinski, Louis-Noël Pouchet, and P. Sadayappan. 2012. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing* (San Servolo Island, Venice, Italy) (ICS '12). Association for Computing Machinery, New York, NY, USA, 311–320. <https://doi.org/10.1145/2304576.2304619>
- [29] Shengya Huang, Zhaocheng Li, Derek Werner, and Yongjoo Park. 2025. MojoFrame: Dataframe Library in Mojo Language. *arXiv preprint arXiv:2505.04080* (2025).
- [30] Zheming Jin and Jeffrey S. Vetter. 2023. A Benchmark Suite for Improving Performance Portability of the SYCL Programming Model. In *2023 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 325–327. <https://doi.org/10.1109/ISPASS57527.2023.00041>
- [31] Engin Kayraklıoglu and Andy Stone. 2024. Productive, Vendor-Neutral GPU Programming Using Chapel. In *SC24-W: Workshops of the International Conference*

⁶<https://docs.modular.com/mojo/faq>

- for High Performance Computing, Networking, Storage and Analysis. IEEE, 1914–1922.
- [32] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. 2012. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel computing* 38, 3 (2012), 157–174.
- [33] Marcin Krotkiewski and Marcin Dabrowski. 2013. Efficient 3D stencil computations using CUDA. *Parallel Comput.* 39, 10 (2013), 533–548. <https://doi.org/10.1016/j.parco.2013.08.002>
- [34] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: a LLVM-based Python JIT compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (Austin, Texas) (LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. <https://doi.org/10.1145/2833157.2833162>
- [35] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86.
- [36] Chris Lattner and Jacques Pienaar. 2019. MLIR primer: A compiler infrastructure for the end of Moore’s law. In *Compilers for Machine Learning. C4ML workshop at CGO 2019*. 100.
- [37] Wei-Chen Lin and Simon McIntosh-Smith. 2021. Comparing Julia to Performance Portable Parallel Programming Models for HPC. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*. 94–105. <https://doi.org/10.1109/PMBS54543.2021.00016>
- [38] Ami Marowka. 2021. Toward a Better Performance Portability Metric. In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 181–184. <https://doi.org/10.1109/PDP52278.2021.00036>
- [39] Ami Marowka. 2025. Portability efficiency approach for calculating performance portability. *Future Generation Computer Systems* 170 (2025), 107826. <https://doi.org/10.1016/j.future.2025.107826>
- [40] Nicholas D Matsakis and Felix S Klock. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. 103–104.
- [41] Timothy G Mattson, Todd A Anderson, and Giorgis Georgakoudis. 2021. Pyomp: Multithreaded parallel programming in python. *Computing in Science & Engineering* 23, 6 (2021), 77–80.
- [42] Modular. [n. d.]. Mojo, Powerful CPU+GPU Programming. <https://www.modular.com/mojo>
- [43] ROYUD Nishino and Shohei Hido Crissman Loomis. 2017. Cupy: A numpy-compatible library for nvidia gpu calculations. *31st conference on neural information processing systems* 151, 7 (2017).
- [44] OpenMP Architecture Review Board. 2021. OpenMP Application Program Interface Version 5.2. <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf>
- [45] S John Pennycook and Jason D Sewall. 2021. Revisiting a metric for performance portability. In *2021 International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*. IEEE, 1–9.
- [46] Simon J Pennycook, Jason D Sewall, and Victor W Lee. 2019. Implications of a metric for performance portability. *Future Generation Computer Systems* 92 (2019), 947–958.
- [47] Konstantin F. Pilz, James Sanders, Robi Rahman, and Lennart Heim. 2025. Trends in AI Supercomputers. arXiv:2504.16026 [cs.CY] <https://arxiv.org/abs/2504.16026>
- [48] César Piñeiro and Juan C. Pichel. 2026. OMP4Py: A pure Python implementation of openMP. *Future Generation Computer Systems* 175 (2026), 108035. <https://doi.org/10.1016/j.future.2025.108035>
- [49] Andrei Poenaru, Wei-Chen Lin, and Simon McIntosh-Smith. 2021. A performance analysis of modern parallel programming models using a compute-bound application. In *International Conference on High Performance Computing*. Springer, 332–350.
- [50] Nishat Raihan, Joanna C. S. Santos, and Marcos Zampieri. 2025. MojoBench: Language Modeling and Benchmarks for Mojo. In *Findings of the Association for Computational Linguistics: NAACL 2025*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 4109–4128. <https://doi.org/10.18653/v1/2025.findings-naacl.230>
- [51] Julian Samaroo et al. 2023. *JuliaGPU/AMDGPU.jl*: v0.7.3. <https://doi.org/10.5281/zenodo.10040461>
- [52] Vijay A. Saraswat, Vivek Sarkar, and Christoph von Praun. 2007. X10: Concurrent Programming for Modern Architectures. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (San Jose, California, USA) (PPoPP '07)*. Association for Computing Machinery, New York, NY, USA, 271. <https://doi.org/10.1145/1229428.1229483>
- [53] John E Stone et al. 2010. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering* 12, 3 (2010), 66.
- [54] Christian R. Trott et al. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. <https://doi.org/10.1109/TPDS.2021.3097283>
- [55] Pedro Valero-Lara et al. 2024. JACC: Leveraging HPC Meta-Programming and Performance Portability with the Just-in-Time and LLVM-based Julia Language. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1955–1966. <https://doi.org/10.1109/SCW6324.2024.000245>
- [56] Guido Van Rossum et al. 2007. Python Programming Language.. In *USENIX annual technical conference*, Vol. 41. Santa Clara, CA, 1–36.
- [57] Jeffrey S. Vetter, Ron Brightwell, Maya Gokhale, Pat McCormick, Rob Ross, John Shalf, Katie Antypas, David Donofrio, Travis Humble, Catherine Schuman, Brian Van Essen, Shinjae Yoo, Alex Aiken, David Bernholdt, Suren Byna, Kirk Cameron, Frank Cappello, Barbara Chapman, Andrew Chien, Mary Hall, Rebecca Hartman-Baker, Zhiling Lan, Michael Lang, John Leidel, Sherry Li, Robert Lucas, John Mellor-Crummey, Paul Peltz Jr., Thomas Peterka, Michelle Strout, and Jeremiah Wilke. 2018. Extreme Heterogeneity 2018 - Productive Computational Science in the Era of Extreme Heterogeneity: Report for DOE ASCR Workshop on Extreme Heterogeneity. (12 2018). <https://doi.org/10.2172/1473756>
- [58] Sandra Wienke et al. 2012. OpenACC—first experiences with real-world applications. In *Euro-Par 2012 Parallel Processing: 18th International Conference*. Springer, 859–870.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

A Overview of Contributions and Artifacts

A.1 Paper's Main Contributions

The main goal of the paper is to understand the recently added performance portability capabilities of the MLIR (Multi-Level Intermediate Representation)-based Mojo on modern NVIDIA and AMD (since June 2025) GPUs. Mojo is a novel programming language that aims to close performance gaps using an interoperable compile-time superset that is interoperable at run-time with the full Python ecosystem. All the codes used for this study are provided on a single repository: <https://github.com/tdehoff/Mojo-workloads> hosted on GitHub with a permissive MIT open-source software license.

- C_1 Understand the performance of GPU-portable Mojo code against CUDA and HIP targeting widely-used kernels across science domains: Seven-point stencil, BabelStream, Hartree-Fock and miniBude.
- C_2 Understand observed performance differences (over and under-performance) on NVIDIA H100 and AMD MI300A GPUs at a lower-level using NVIDIA's Nsight profiling tools.
- C_3 Develop, document and provide readily-available open-source Mojo ports of relevant these science kernels to the community for experimentation with this novel language.

A.2 Computational Artifacts

- A_1 New Mojo ports, and existing CUDA and HIP codes:
<https://github.com/tdehoff/Mojo-workloads>

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1 and C_3	Tables 2-5 Figure 2-7

B Artifact Identification

B.1 Computational Artifact A_1

Relation To Contributions

All the provided Mojo codes have been developed to generate the results on this paper. The listed A_1 repository contains the following subdirectories, each containing a relevant proxy applications used in our evaluations and will produce the attached relevant metric listed as follows:

- seven-point stencil (memory bandwidth)
- babelstream (memory bandwidth)
- miniBude (compute-bound)
- hartree-fock (kernel wall-clock time)

Each of these directories contains subdirectories with the portable Mojo, and the vendor-specific CUDA, and HIP implementations used for the performance study. Hence Figures 2 to 7 will be straightforward plots of the resulting runs containing 1000 iterations to capture computational variability.

Expected Results

The expected results are the performance metrics, in Eqs. 1 for seven-point stencil, 2 for BabelStream, 3 for miniBUDE, and wall-clock time for Hartree-Fock, that are part of the output when running the listed codes. As shown in the Figures 2-6 and Tables 2-4, we expect that Mojo codes will be on-par with the HIP versions on AMD GPUs. Whereas differences reported with CUDA code on NVIDIA GPUs should be reproducible. Table 2-4 results require the use of the NVIDIA Nsight CLI (ncu) profiling tool. We profile using the ncu command line as detailed in the next section.

Expected Reproduction Time (in Minutes)

Each of the runs can be executed in less than 1 minute per case. Users can select the number of iterations to test the kernels.

Artifact Setup: all codes are hosted publicly on GitHub, or just downloading them from Zenodo. CUDA and HIP C++ codes use Makefiles for the appropriate target, e.g. make cuda. Compilation of all codes should be take 1-5 seconds as these are proxy apps.

Artifact Execution:

- Seven-point stencil: 5 s
- BabelStream: 5 s
- miniBude: 5 s
- Hartree-Fock: 5s, he1024 case take 124 s on H100

Artifact Analysis: We provide Python plotting scripts that take the generated code output data as inputs. It captures performance numbers as text and generates Figures 2-3 shown in the paper. It should take less than 2 s to generate each plot.

Profiling. we use NVIDIA Nsight CLI (ncu) that ships as part of NVIDIA HPC SDK (nvhpc). The following command is a straightforward use-case the same code at a smaller number of iterations (e.g., 10).

```
$ ncu -f -o output --set roofline ./ babelstream
```

Mojo code need to be pre-compiled (e.g. program) by running the following commands:

```
$ pixi shell
(mojo) $ mojo build babelstream.mojo
(mojo) $ ./ babelstream
```

Artifact Setup (incl. Inputs)

Hardware. The GPUs used in this study are listed in Table 6.

GPU - Memory	Theoretical Peaks		
	Bandwidth GB/s	FP32	FP64
		TFLOPS/s	TFLOPS/s
NVIDIA H100 NVL - 94 GB	3,900	60.0	30.0
AMD MI300A - 128 GB HBM3	5,300	122.6	61.3

Table 6: GPU hardware used in this study

Software. We use a single self-contained repository for all the codes used in this study:

- Mojo-workloads with A_1 codes: <https://github.com/tdehoff/Mojo-workloads>

Datasets / Inputs. No special datasets are used as inputs. Hartree-Fock codes use the original dataset files provided by the original Fortran proxy app in <https://github.com/gdfletcher/basic-hf-proxy/tree/main/tests>. These files are provided under the hartree-fock subdirectory. Inputs can be directly modified on each file (e.g. problem size and types) to match the Figures in the paper. It should be very intuitive as each kernel only depend on a few problems sizes, types and number of iterations.

Installation and Deployment. Code subdirectories in A_1 can be considered stand-alone executables. They need the following Mojo and NVIDIA CUDA and AMD ROCM compiler versions:

- Mojo: Mojo 25.5.0.dev2025062505 <https://docs.modular.com/mojo/manual/gpu/intro-tutorial/>
- C++ CUDA codes require NVIDIA HPC SDK (nvhpc) v24.9 <https://developer.nvidia.com/hpc-sdk-releases>
- C++ HIP codes require AMD ROCM v6.4.0 <https://github.com/ROCM/ROCM/tree/rocm-6.4.0>

Artifact Execution

All the results can be easily reproducible. We summarize the typical runs and configuration parameters for each Mojo port of the following proxy applications:

- seven-point stencil
- BabelStream
- miniBUDE
- Hartree-Fock (input data in test directory)

Modifying configuration parameters is straightforward and they represent job characteristics related to problem size, type (where appropriate), and GPU threads-per-block (TBSIZE). Vendor-specific CUDA and HIP implementations are part of the directories structure and each proxy and model combination has its own makefile that contains the relevant compiler flags. The following are the common parameters and the instructions for building and running the novel Mojo implementations.

Seven-point stencil. uses the following parameters that can be edited in the source code.

- Problem sizes of $L = 512$ and 1024
- TBSIZE = $(1024, 1, 1)$ or $(512, 1, 1)$
- Iterations, num_iter = 1000
- Precision = Float32 or Float64

```
JIT run
$ pixi run mojo laplacian.mojo --csv
AOT run
$ pixi shell
(mojo)$ mojo build laplacian.mojo
(mojo)$ ./laplacian --csv
```

Listing 6: "Seven-point stencil run options"

BabelStream. the main parameter is the problem size. In our results, we use a size of $2^{25} = 33,554,432$ allowing us to saturate GPU device memory. Since we use a 1D computational grid, we only set the first dimension to a fixed value of 1,024. BabelStream will run all the kernels in the benchmark: Copy, Mul, Add, Triad and Dot.

- Problem sizes of $L = \text{pow}(2,15)$
- TBSIZE = $1024 (1024, 1, 1)$
- Iterations, num_iter = 1000
- Precision = Float32 or Float64

```
JIT run
$ pixi run mojo babelstream.mojo --csv
AOT run
$ pixi shell
(mojo)$ mojo build babelstream.mojo
(mojo)$ ./babelstream --csv
```

Listing 7: "miniBUDE run options"

miniBUDE. we limit the scope to the bm1 benchmark. The main parameters are the poses per work-item (PPWI), and (ii) work-group (wg) size, all other parameters remain constant including the number of poses set to a large value of 65,536. We use sizes of PPWI that are a power of 2 from 1 to 128, while wg is set for 2 values of 8 and 64. Floating Precision is fixed due to the nature of the kernel.

- Problem sizes PPWI = $1,2,4,8,16,32,64,128$, wg = 8,64
- num_poses = 65,536
- TBSIZE = $1024 (1024, 1, 1)$
- Iterations, num_iter = 1000

```
JIT run
$ pixi run mojo minibude.mojo
AOT run
$ pixi shell
(mojo)$ mojo build minibude.mojo --csv
(mojo)$ ./minibude --csv
```

Listing 8: "miniBUDE run options"

Hartree-Fock. runs are set to a fixed size of atoms modeling a system of Helium atoms. The datasets needed for each run are included inside the test directory and each file needs to be passed as an argument. We ran successfully up to a number of atoms of 256, while we observe abnormal behaviors in terms of performance for the 512 and 1024 cases. All other parameters are kept constant. The parameters natoms and ngauss must be modified in the source code to match the corresponding file in the test directory.

- Problem sizes natoms = 64, 128, 256, 1024 ngauss = 3 or 6 (1024 only)
- Iterations, num_iter = 1000

```
JIT run
$ pixi run mojo hartree-fock.mojo .../tests/he256
AOT run
$ pixi shell
(mojo)$ mojo build hartree-fock.mojo
(mojo)$ ./hartree-fock .../tests/he256 --csv
```

Listing 9: "Hartree-Fock run options for 256 atoms test"

Artifact Evaluation (AE)

C.1 Computational Artifact A_1

Artifact Setup (incl. Inputs)

The single source code repository in <https://github.com/tdehoff/Mojo-workloads>, for Mojo and vendor-specific CUDA and HIP versions and scripts for plotting the results, is structured as follows:

- <science-workload>/
- Mojo
- cuda
- hip
- plotting

where <science-workload> are: 7-point-stencil, babelstream, hartree-fock, and miniBUDE subdirectories.

Thus each Mojo, CUDA and HIP version runs independently, respectively. CUDA and HIP versions posses a makefile, whereas Mojo versions have a single mojo source code and pixi toml files for reproducibility. Plotting files are simple Python's pandas, matplotlib, and seaborn scripts to generate Figs. 3, 4 and 6 for each application, except Hartree-Fock for which wall-clock experiments' runs results are presented in Table 4.

C.2 Compiling libraries and code

Each Mojo, CUDA, HIP rely on standard installations of the language, and compiler toolchains widely available for any Linux distribution system:

Mojo (both NVIDIA and AMD GPUs): <https://docs.modular.com/mojo/manual/gpu/intro-tutorial/#1-create-a-mojo-project>

- pixi 0.48.2
- Mojo 25.5.0.dev2025070105

CUDA: <https://developer.nvidia.com/hpc-sdk-releases>

- nvhpc v24.9
- cuda 12.6
- gec/g++ 13.3.0

HIP: <https://rocm.docs.amd.com/projects/install-on-linux/en/latest/>

- rocm v6.4.0
- gec/g++ 13.3.0

To compile Mojo code (e.g. Seven-point stencil's laplacian.mojo file):

```
JIT run
$ pixi run mojo laplacian.mojo --csv
AOT run
$ pixi shell
(mojo)$ mojo build laplacian.mojo
(mojo)$ ./laplacian --csv
```

Listing 10: "Seven-point stencil run options"

To compile CUDA or HIP code execute the corresponding Makefile (using make) inside the <science-workload>/cuda and the <science-workload>/hip subdirectories. The following code presents the Makefile for BabelStream using CUDA on a NVIDIA H100 which will generate the corresponding executable:

```
$ make
Makefile :

CC = nvcc
FLAGS=-forward-unknown-to-host-compiler
      -arch=sm_90 -DNDEBUG -O3 -march=
      native -std=c++11

SRCS = main.cpp CUDAStream.cu
HEADERS = Stream.h CUDAStream.h
TARGET = babelstream_cuda

all: $(TARGET)

$(TARGET): $(SRCS) $(HEADERS)
$(CC) $(FLAGS) -o $@ $(SRCS)

clean:
rm -f $(TARGET)
```

Listing 11: "BabelStream CUDA makefile"

C.3 Workflow description

Example using the seven-point stencil case as a reference to generate Figs. 3a and 3b:

1. Clone the GitHub repo:

```
$ git clone https://github.com/tdehoff/Mojo-workloads
```

2. Run the Mojo code with csv output (e.g. JIT run)

```
cd 7-point-stencil/Mojo
$ pixi run mojo laplacian.mojo -csv
```

3. Run the CUDA (or HIP) code

```
cd ../cuda set csv_output to true in the code. $ make
$ ./laplacian_kernel
```

4. Merge outputs from Step 2 and 3 into a single csv file and run the plotting scripts to generate Fig. 3:

```
python ./plot_NVIDIA.py
- or -
python ./plot_AMD.py
```

Examples for CSV files used in this study are provided in the plotting subdirectories.

Repeat the same process for BabelStream and miniBUDE.