JULIA FOR PRODUCTIVITY AND PERFORMANCE: A COMPARATIVE ANALYSIS

Damian Camenisch, Eiman Alnuaimi, Noëmi Marty, Steffen Backmann, Thomas Creavin

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

The two-language problem is ever prevalent in the scientific computing domain. Thanks to its simplicity, Python is widely used for prototyping and scripting, whereas low-level languages like C are indispensable when performance is key. The Julia programming language claims to solve the two-language problem. It offers a high-level syntax for fast prototyping while at the same time making it possible to achieve high performance. In this paper, we evaluate Julia's overall effectiveness. We implement and optimize a diverse set of microbenchmarks from the PolyBench benchmark suite in both C and Julia, for CPU as well as GPU, and compare performance and development experience[1]. We find that Julia delivers on its claims, providing a superior development experience compared to C, with performance falling behind by only a few percentage points in general.

1. INTRODUCTION

Motivation. High-performance computing has long relied on the formidable speed of C, despite its trade-off in expressiveness. In recent years, Python has gained prominence despite its inherent performance limitations, particularly in domains like machine learning and data science, thanks to its simplicity and rich ecosystem. The Julia programming language emerges as a unique solution to this two-language problem [1]. Julia claims to deliver the performance of C and the expressiveness of Python at the same time, thus it presents itself as a compelling choice in the scientific computing domain.

As the computing landscape evolves, GPUs are asserting their significance, notably in machine learning where they play a pivotal role. As NVIDIA GPUs currently dominate the market, NVIDIA's CUDA framework dominates the space of GPU kernel development. CUDA is designed to work with languages like C and C++, and is used by many software libraries to harness the power of GPUs. For Julia, the CUDA.jl package has been developed, which enables developers to write kernel functions that execute on the GPU in Julia.

This paper explores the extent to which Julia's performance matches that of equivalent C programs and whether Julia facilitates a Python-like development experience both for CPU and GPU code. To address these questions, we port a set of eight diverse benchmarks from the PolyBench suite to both C and Julia and compare their performance. We further compare our versions against the NPBench [2] suite to gauge Python's performance in a broader context.

Related work. Existing research has extensively explored the development of the Julia language and its competitiveness in the HPC space. Among them, [3] investigates Julia's performance for various machine learning algorithms and finds that it closely rivals C, beating out other languages like Fortran and Go and demonstrating the language's competitiveness. As for its suitability for HPC, these researchers [4] achieved great success in writing Julia code to target the A64FX processor, Fujitsu's processor for supercomputers. They found that their MPI (Message Passing Interface) applications performed nearly identically to their C implementations. They herald praise for the boost in scientific productivity, particularly when developing generic numerical code that can effortlessly use different numerical data types without sacrificing performance.

[5] compares C and Python implementations of the PolyBench benchmarks. They focus on polyhedral optimizations to speed up the Python version. Additionally, works like [6] call for the use of a general-purpose high-performance language in scientific computing to enable more non-experts to contribute to this increasingly performance-demanding field. They demonstrate through a series of examples how naive Julia implementations of algorithms that leverage the GPU can achieve great performance with ease.

Our work takes this one step further by directly evaluating Julia's performance, both on CPU and GPU, against its rivals C and Python (NumPy) as well as speaking to our experience developing and optimizing code in these languages.

[1] https://github.com/Zhurgut/DPHPCProject

2. BACKGROUND

PolyBench. The Polyhedral Benchmark Suite (PolyBench) is a collection of micro-benchmarks from various domains [7].

NPBench. [2] This suite benchmarks scientific Python/ NumPy code, including PolyBench benchmarks. It utilizes a range of NumPy-accelerating compilers and frameworks to enhance the performance of Python codes.

CUDA. is a parallel computing platform and application programming interface (API) developed by NVIDIA [8]. It enables developers to leverage the parallel processing power of NVIDIA GPUs for general-purpose computing tasks, allowing for efficient acceleration of compute-intensive applications by offloading parallelizable workloads to the GPU.

Julia. [1] Julia is an open-source, performance-oriented, high-level programming language developed at MIT, designed from the outset to compete with the performance of C while being as expressive as Python. Julia leverages LLVM [9] to produce performant code. The CUDA.jl [10] package leverages Julia's CPU compiler infrastructure to seamlessly integrate GPU kernel development for NVIDIA GPUs into the Julia language.

3. BENCHMARKS AND OPTIMIZATIONS

In this study, we selected eight benchmarks from the PolyBench suite to transpose. For each benchmark, we implemented and optimized versions in both C and Julia, for both CPU and GPU. We chose a large number of benchmarks for the context of this project. This makes it possible to get a broader understanding of Julia's performance characteristics. On the other hand, this also means our optimization efforts have by no means been exhaustive, and there is still room for improvement as far as performance is concerned.

Each benchmark is introduced briefly, accompanied by concise descriptions of the effective optimizations applied during the implementation process.

COVARIANCE. This benchmark computes the empirical covariance matrix given a data matrix. It is computed by mean-adjusting each column, that is, computing the mean of each column and subtracting it from each value and then performing a matrix multiplication. The matrix multiplication operation dominates the majority of the runtime. It is bottlenecked by inefficient memory access. To overcome this, we transpose the data to reduce cache misses and use memory-concise access patterns. We achieve additional speedups by using accumulators and fusing smaller kernels. We gain slight improvements using the Julia Single Instruction, Multiple Data `@simd` and `@inbounds` macros.

DOITGEN. This benchmark is used in multiresolution analysis kernels and executes a sequence of in-place matrix-matrix multiplications along the last dimension. Its performance characteristics vary between compute-bound and memory-bound depending on its input parameters nr, nq, and np. Julia's column-major order yields an advantage in the straightforward implementation. Notable optimizations include precomputing a temporary variable, resulting in a sizeable improvement in the C implementation, and optimizing memory access patterns through coalescing, yielding a significant enhancement for both Julia and C. A slight speed-up for Julia is achieved by removing bounds checking for memory accesses with the `@inbounds` macro. By adjusting and applying well-known optimizations for matrix-matrix multiplications like the usage of shared memory to this similar problem, further improvements are likely.

FLOYD-WARSHALL. The FLOYD-WARSHALL algorithm solves the All Pairs Shortest Path (APSP) problem in a graph. The naive implementation rests on a straightforward parallelization strategy that iteratively employs a 2D thread grid for every node k, where each thread calculates the detour via node k for a specific source-destination pair. This is primarily limited by the need to serially iterate over every node k which results in suboptimal memory accesses. Here again, C's row-major memory layout leads to its straightforward implementation performing worse and the first notable optimization is transposing the thread index. Apart from that, optimizations like reducing branch divergence and disabling bounds checking in Julia yield only minimal improvements. A considerable speed-up for both C and Julia is only achieved when implementing a three-staged blocked version of the algorithm following the approach presented in [11].

General Matrix Multiplication (GEMM). This benchmark serves as a fundamental component for various operations in neural networks. It is defined as the operation $C_{M\times N} = \alpha \cdot (A_{N\times K} \cdot B_{K\times M}) + \beta \cdot C_{M\times N}$. Its optimization is challenged by efficient memory access. Some noteworthy optimization techniques include transposing the matrices, using accumulators, loop unrolling, SIMD, and global memory coalescing. Among these, the most impactful optimizations, include loop unrolling in Julia and using column-major order in C.

SYRK. Symmetric Rank-K Update. The operation $C = \alpha(AA^T) + \beta C$ is performed, where matrix A has significantly fewer columns ($K \ll N$). On GPU, each thread is tasked with computing the inner product between two rows of A, emphasizing the optimization of memory access patterns for enhanced efficiency.

JACOBI-2D. This benchmark implements a 2-D Jacobi stencil computation on a square matrix $A_{N\times N}$. With the matrix only being updated at the end of the iteration, each matrix element is averaged together with its four closest neighbours. This procedure is repeated a set number of times. Parallelizing this routine for GPU is straightforward, as using a second matrix to cache the previous state al-

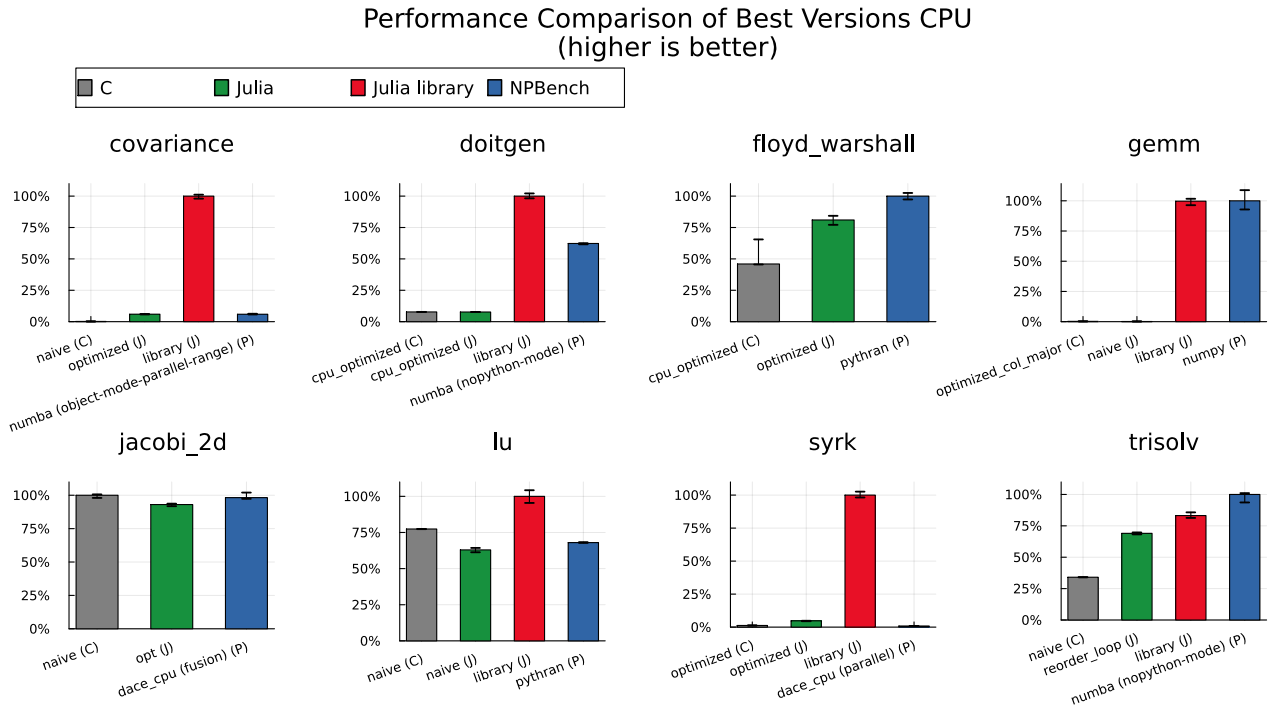

Fig. 1. Performance Comparison of the best CPU version for C, Julia, and NPBench. Performance is computed as $\frac{1}{Time}$ and re-scaled such that the overall best version for each benchmark achieves 100%. Our Julia versions achieve good performance, usually quite close to their C equivalents, and sometimes even compete with existing library implementations.

lows for a full iteration of the stencil computation to be performed all at once, without any in-loop dependencies. Further optimizations would require changing the stencil computation entirely to improve locality, thereby also introducing complex dependencies. On CPU, unrolling and reordering brings the Julia performance on par with C (where this is done by GCC automatically).

Lower–Upper (LU) Decomposition. This benchmark implements LU decomposition on a square matrix $A_{N\times N}$. The algorithm implements Doolittle's method [12] to iteratively calculate one row of the upper triangular matrix followed by a row of the lower triangular matrix until A is exhausted. Parallelizing the LU decomposition algorithm for GPU poses challenges primarily linked to potential race conditions. The intricate dependencies within the nested loops make it difficult to parallelize efficiently, as simultaneous updates to matrix elements by multiple threads may result in data inconsistency. The most high-performant optimization technique in both Julia and C is dividing the core loop into two kernels, effectively addressing lower triangular and upper triangular matrix updates separately to prevent race conditions during parallel execution [13].

TRISOLV. This benchmark solves a system of linear equations where the coefficient matrix is lower triangular. The key challenge for this benchmark is handling the inter-loop dependency that arises from forward-substituting each variable in the equation. The initial naive GPU implementation performed significantly worse than the CPU variant because it didn't parallelize well. Some Basic optimisations are necessary so that the GPU implementation can match the speed of the CPU variant. To improve the performance of the GPU implementation further, we fuse the forward-substitution and the scalar update into one kernel. This avoids the cost of copying the data to and from the CPU to update a single value each iteration. Additionally, we gain smaller speed-ups from using the Julia `@inbounds` and `@fastmath` macros.

4. EXPERIMENTAL RESULTS

Experimental setup. We use an Intel 7700K CPU (4.5GHz) and a NVIDIA 1080Ti GPU to collect our measurements. C codes are compiled using `gcc 10.5.0` with flags `-O3 -march=native`. To measure execution time, we use `clock_gettime(CLOCK_MONOTONIC_RAW)` in C and `time_ns()` in Julia. NPBench always collects 10 measurements for the Python versions. We opt to collect up to 200 to get narrower confidence intervals. Plots are based on the empirical median runtime. The error bars denote the 95% confidence intervals for the median which are calcu-

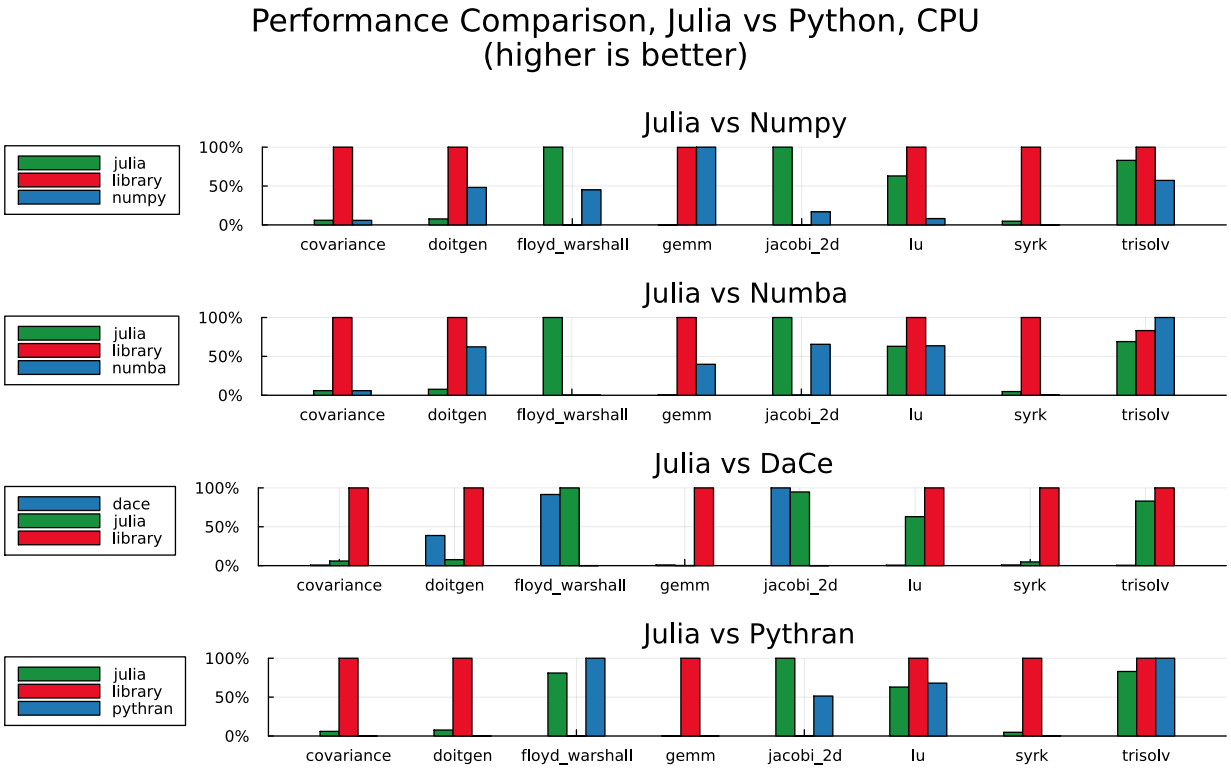

Fig. 2. Performance Comparison of Julia against the different NPBench frameworks on CPU. Performance is computed as $\frac{1}{Time}$ and re-scaled such that the best version for each benchmark achieves 100%. Unlike the various Python frameworks, Julia manages to deliver high performance consistently across benchmarks, if not through our custom versions, then thanks to its optimized standard library implementations.

lated according to [14], pages 32, 313.

Unlike C and Python, Julia stores multidimensional arrays in column-major order and so we transpose the benchmark input matrices where necessary. This is done before running the measurement to facilitate one-to-one comparisons between the C and Julia versions.

We used the following problem sizes when collecting the timing measurements.

COVARIANCE	M=2000	N=2400	
DOITGEN	NR=250	NQ=280	NP=320
FLOYD-WARSHALL	N=1400		
GEMM	NI=2000	NJ=2200	NK=2400
JACOBI-2D	TSTEPS=300	N=1600	
LU	N=600		
SYRK	M=1600	N=2000	
TRISOLV	N=14000		

Julia Library Versions. Julia ships with the extensive LinearAlgebra and Statistics packages as part of its standard library. We use these packages to implement library versions of our benchmarks where possible. These versions serve both as a reference for peak performance but also highlight Julia's focus on providing high performance for scientific computing. It is incredibly easy in Julia to achieve state-of-the-art performance, without even needing to install a package.

These libraries are optimized for single-core CPU performance. The CUDA package extends much of the functionality from the LinearAlgebra package with respective equivalents, optimized for GPU. It is thus possible to write code that runs on GPU using a high-level syntax, foregoing the need to write custom kernels. Here is an example of how we implemented the library version for the COVARIANCE benchmark on GPU:

```
1 function cov(m, n, D::CuMatrix{Float64})
2     mean = 1/n .* sum(D, dims=1)
3     D .-= mean
4     return 1/(n-1) .* (D' * D)
5 end
```

We use this high-level approach to implement library versions for GPU. While we expect the CPU library versions to perform near optimally, this approach can cause significant chunks of performance to be left on the table, as operations are performed one after the other, even when they

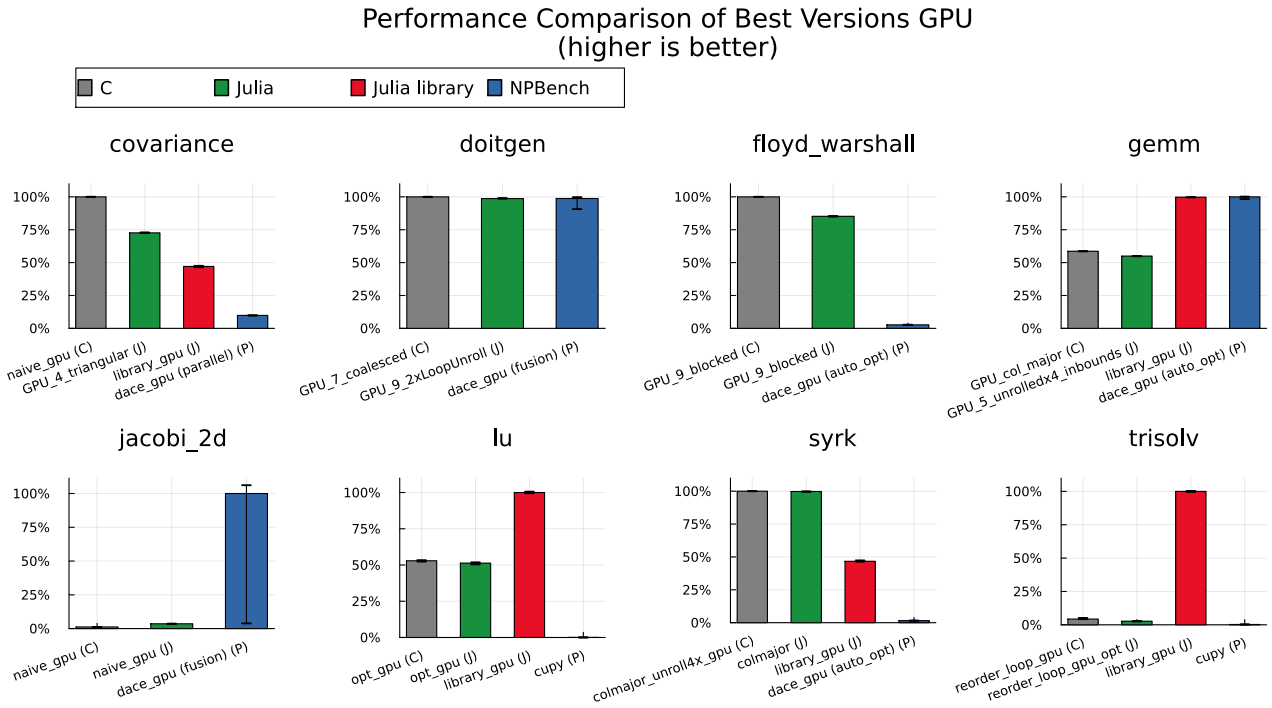

Fig. 3. Performance Comparison of the best GPU version for C, Julia and NPBench. Performance is computed as $\frac{1}{Time}$ and re-scaled such that the overall best version for each benchmark achieves 100%. Julia kernels usually achieve very similar performance to their respective C counterparts.

could in theory be fused into a single kernel. This performance loss can be seen in Fig. 3, where library versions for both COVARIANCE and SYRK perform significantly worse than our custom kernels. Nevertheless, many applications enjoy a large speedup when porting a CPU implementation to a GPU one in this manner, with very little development overhead.

CPU Performance Results. We find that for CPU, Julia can generate code that is as fast as C code effortlessly. Julia offers macros such as `@inbounds` to remove bounds-checking in arrays, `@views` to convert array slices into array views and `@.` to vectorize code, which causes function calls to be fused and executed in a single loop. With judicious usage of these macros, it is often possible to convert a pseudo-code-style prototype into a performant, allocation-free program with ease.

We see in Fig. 1 how our Julia versions can keep up with and sometimes even beat out C. It has to be noted that we focused on developing the GPU versions, so the performance of different languages and versions is largely limited by optimization effort. Nevertheless, we still show that Julia can produce code with similar performance to C. An excellent example of this is the LU benchmark in Fig. 1: the naive triple loop implementation in Julia is only slightly behind its C equivalent and achieves about 60% of the performance of the LU library call. Note that in Fig. 1, the NPBench implementations for GEMM and DOITGEN use the `@` operator for matrix-matrix or matrix-vector multiplication, which is why only our library implementations are able to compete with NPBench for these two benchmarks.

Fig. 2 compares Julia against the various Python frameworks implemented in NPBench on CPU. Python usually relies on C libraries such as NumPy for performance. This approach does not always offer the required flexibility as the desired computation might not be feasible through library calls alone. Hence, there are many frameworks that aim to accelerate Python programs or compile them to efficient code. NPBench offers a great opportunity to compare Julia against this "fast Python". We can see in Fig. 2 that each framework can produce great code for some benchmarks but none perform well across all our benchmarks. Julia on the other hand consistently delivers great results. Where Python uses NumPy for matrix-matrix multiplication, such as in GEMM and DOITGEN, Julia offers its own library versions to match, and when the code is written with nested loops and array slices, our custom Julia versions never fall far behind even the best NPBench counterpart.

GPU Performance Results. Our evaluation primarily focused on GPU benchmarks, revealing varied performance levels across our custom C and Julia implementa-

tions, especially when compared to NPBench and Julia's library versions. As can be seen in Fig. 3, COVARIANCE, FLOYD-WARSHALL, LU, SYRK and TRISOLV significantly outperform the GPU NPBench versions. COVARIANCE and SYRK also show a noticeable improvement over the Julia library implementation. However, in the case of GEMM and JACOBI-2D, our C and Julia implementations lag behind NPBench, while the DOITGEN versions align closely with NPBench's performance.

It's crucial to note the inconsistent performance of NPBench across different frameworks. Although `dace_gpu` variants perform best in most benchmarks among the different Python versions, they fall short in tasks like LU and TRISOLV where CuPy is better. This inconsistency underscores the potential variability in Python-based GPU performance that does not come with Julia or C.

Our findings indicate that, across all benchmarks, Julia's custom versions generally parallel the performance of their C counterparts, albeit with some minor discrepancies. Generally, the Julia versions perform slightly worse albeit within a range of less than 15% performance loss (Exceptions: TRISOLV - 37% performance loss, COVARIANCE - 27% performance loss). For JACOBI-2D, Julia even provides a performance gain of 194%, however as the JACOBI-2D versions are still very basic and also perform worse than the NPBench implementation, this suggests there is still significant room for improvement left. These results critically address our research query regarding Julia's GPU performance when compared with C, highlighting Julia's capability to deliver comparable results to C, with only marginal losses in most cases.

Julia and Productivity. In the realm of Julia and productivity, certain nuances of the language may pose initial challenges for users. The 1-based indexing and the column-major storage order for n-dimensional arrays, diverging from the conventions in many contemporary languages, require a period of adaptation.

Julia's high-level syntax, designed for expressiveness and readability, may not always align seamlessly with performance optimization goals, particularly in the context of GPU kernels. To address this, users often incorporate macros such as `@inbounds` and `@view` to ensure Julia generates more lightweight kernel code, akin to the default behavior in languages like C. These macros are crucial not only for GPU development but also on CPUs, ensuring the production of allocation-free code.

One distinct advantage of Julia lies in its memory management, offering a more user-friendly experience compared to C. Julia namely is a garbage-collected language, the programmer does not have to free memory manually like in C. This convenience contributes significantly to overall productivity, allowing developers to focus more on algorithmic aspects rather than intricate memory handling.

Julia simplifies the CUDA installation process, making it remarkably straightforward. With a simple package addition, CUDA functionality becomes seamlessly integrated, even on personal laptops. Additionally, Julia provides a basic yet functional built-in CUDA profiler, offering a convenient tool for performance analysis during development. These aspects collectively make Julia an appealing choice, not just for its high-level syntax but also for the ease it brings to GPU development and memory management.

Julia's CUDA documentation, though generally good, lacks the comprehensive detail found in CUDA C. Additionally, resource availability for CUDA in Julia such as blog articles, StackOverflow discussions and GitHub repositories, though steadily growing, are more extensive for CUDA C.

Julia's CUDA.jl package also does not interface with all CUDA functionality. E.g. support for special floating point types is very limited. As a result, CUDA C is still indispensable when the goal is to achieve the absolute best performance possible.

5. CONCLUSIONS

In conclusion, Julia emerges as a compelling choice for practitioners seeking to write performant code that leverages GPU capabilities. It can indeed provide C-like performance and excels in offering a Python-like developer experience. For the development of custom kernels, Julia provides a more seamless and iterative process, delivering performance akin to C. However, the Julia ecosystem, while rapidly growing, still lacks the maturity and breadth of resources available for established languages like Python and C. In essence, Julia strikes a balance by providing a versatile environment where performance and ease of development coexist, making it a valuable tool for GPU-accelerated computing tasks.

Possible future works include a more thorough analysis of why some Julia benchmarks fall short of their C counterparts and whether this reveals a general pattern or whether it is simply an artefact of those problems or a shortcoming in our optimization techniques. Another direction would be the integration of Julia into the NPBench framework so that it can be easily compared to NumPy/Python on a variety of scientific benchmarks.

6. REFERENCES

[1] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[2] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler, "Npbench: A benchmarking suite for high-performance numpy," in *Pro-*

ceedings of the ACM International Conference on Supercomputing, 2021, pp. 63–74.

[3] Kaifeng Gao, Gang Mei, Francesco Piccialli, Salvatore Cuomo, Jingzhi Tu, and Zenan Huo, "Julia language in machine learning: Algorithms, applications, and open issues," *Computer Science Review*, vol. 37, pp. 100254, Aug. 2020.

[4] Mose Giordano, Milan Klower, and Valentin Churavy, "Productivity meets Performance: Julia on A64FX," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, Heidelberg, Germany, Sept. 2022, pp. 549–555, IEEE.

[5] Miguel Á. Abella-González, Pedro Carollo-Fernández, Louis-Noël Pouchet, Fabrice Rastello, and Gabriel Rodríguez, "Polybench/python: Benchmarking python environments with polyhedral optimizations," in *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, New York, NY, USA, 2021, CC 2021, p. 59–70, Association for Computing Machinery.

[6] Tim Besard, Valentin Churavy, Alan Edelman, and Bjorn De Sutter, "Rapid software prototyping for heterogeneous and distributed platforms," *Advances in Engineering Software*, vol. 132, pp. 29–46, June 2019.

[7] Louis-Noël Pouchet, "Polybench/c 3.2," http://www.cse.ohio-state.edu/ pouchet/software/polybench/.

[8] David Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 2008, pp. 836–838.

[9] Chris Lattner and Vikram Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, San Jose, CA, USA, Mar 2004, pp. 75–88.

[10] Tim Besard, Christophe Foket, and Bjorn De Sutter, "Effective extensible programming: Unleashing Julia on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[11] Ben Lund and Justin W. Smith, "A multi-stage CUDA kernel for floyd-warshall," *CoRR*, vol. abs/1001.4108, 2010.

[12] RC Mittal and A Al-Kurdi, "Lu-decomposition and numerical structure for solving large sparse nonsymmetric linear systems," *Computers & Mathematics with Applications*, vol. 43, no. 1-2, pp. 131–155, 2002.

[13] D. S. V. Bandara and Nalin Ranasinghe, "Effective gpu strategies for lu decomposition," 2011.

[14] Jean-Yves Le Boudec, *Performance Evaluation of Computer and Communication Systems*, EPFL Press, Lausanne, Switzerland, 2010.