Julia with intelligent runtime for Heterogeneous Computing

Narasinga Rao Miniskar, Valero-Lara Pedro, William Godoy, Keita, Teranishi, Jeffrey S. Vetter
Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, 37830, USA

Abstract

Julia, a high-performance, high-level language, harnesses dynamic typing and LLVM's Just-in-Time compiler to match the speed of C and Fortran in production. Meanwhile, IRIS serves as a heterogeneous runtime that discovers devices dynamically and schedules concurrent work on CPUs, GPUs, FPGAs, and DSPs today. Integrating Julia with IRIS unlocks high-performance, portable, and productive computing for workloads. This synergy simplifies kernel APIs for both data-parallel and task-parallel execution, and it also builds task graphs with intelligent flow-dependency detection via kernel analysis to optimize performance across multiple device types. Additionally, users may implement core kernels in pure Julia, employ the Kernel Abstraction layer, or use JACC, vendor-provided CUDA, HIP, and Thread APIs, and also write native C++, CUDA, HIP, OpenMP, or FPGA HLS kernels for high-performance, scalable, and hardware-agnostic code generation across diverse architectures. We report early results of AXPY executing on CUDA GPUs today. A tiled heterogeneous math library for DGEMM uses vendor kernels, demonstrating the system's versatility.

1 Introduction

High-performance scientific computing has long relied on low-level languages such as C, Fortran, or hand-tuned vendor APIs to extract peak performance from contemporary accelerators. Julia, a dynamically typed, high-level language that compiles to native code through an LLVM Just-in-Time (JIT) pipeline, has emerged as a compelling alternative: it delivers execution speeds comparable to statically compiled languages while preserving an expressive, concise syntax. Despite these advantages, Julia's ecosystem traditionally requires separate backends or hand-written device kernels to exploit GPUs, FPGAs, DSPs, and other accelerators, leading to fragmented codebases and steep learning curves.

To bridge this gap, we pair Julia with *IRIS* [2], a heterogeneous runtime that performs dynamic device discovery and orchestrates concurrent execution across CPUs, GPUs, FPGAs, and DSPs. The integration yields a unified programming model that combines Julia's ease of use with IRIS's hardware-agnostic scheduling, enabling simplified task- and data-parallel APIs and automatic task-graph construction through intelligent dependency analysis. Kernels can be expressed directly in core Julia, through a kernel-abstraction layer, or via the JACC interface (vendor-provided CUDA/HIP/Thread APIs), or in native C++ for CUDA, HIP, OpenMP, and FPGA HLS (Xilinx/Intel) targets. Early experiments with the Sappy framework on CUDA GPUs, together with a tiled heterogeneous DGEMM library that leverages vendor kernels, demonstrate the practicality and performance gains of this combined approach.

2 Related Work

Task-based programming frameworks such as *Dagger* [1] have been widely adopted in the Julia ecosystem to manage dependencies and parallelism. However, Dagger's current implementation relies on explicit data movement and does not provide automatic orchestration of data across heterogeneous resources. In particular, it lacks mechanisms for on-device multi-stream synchronization, which are essential for exploiting concurrent execution on GPUs from both NVIDIA and AMD vendors. Moreover, Dagger does not support the simultaneous execution of kernels on mixed device families—e.g., running a CUDA stream on a NVIDIA GPU while a Xilinx FPGA kernel advances in parallel—leading to fragmented runtime logic. The framework also offers a relatively low-level API for constructing task graphs, requiring developers to manually encode dependencies and create multi-stream pipelines. Consequently, there is a clear need for a higher-level, device-agnostic tasking interface that automatically handles data movement, stream synchronization, and heterogeneous device coordination, while also providing distinct kernels for data-parallel, task-parallel, and other execution paradigms.

3 Approach

The Julia-IRIS integration exposes a dual-interface paradigm that decouples host-side orchestration from device-side execution. The *forward interface* allows Julia applications to describe work by creating and submitting tasks, task graphs, and data memory objects (DMEM) directly to the IRIS runtime. These constructs are expressed in native Julia syntax and are translated into IRIS-compliant APIs that the runtime can schedule across any available accelerator. Complementing this, the *reverse interface* provides callback mechanisms whereby IRIS invokes Julia's own compiler to instantiate and launch Julia kernels on the target device within the same runtime context. This bidirectional glue eliminates the need for boilerplate host code for kernel invokation and enables seamless integration of Julia-written kernels into heterogeneous pipelines.

Kernel support within this framework is intentionally heterogeneous. On the Julia side, developers may write *native Julia kernels* using the language's *Threads*, *CUDA*, or *HIP* packages, leveraging familiar Julia abstractions while benefiting from JIT-generated machine code. For devices that lack direct Julia bindings, the *Kernel Abstraction* / *JACC* layer allows a single source kernel to be compiled and deployed across multiple back-ends (CUDA, HIP, Thread) without code duplication. Finally, for performance-critical segments or legacy modules, developers can write *native IRIS kernels* in traditional programming models such as CUDA, HIP, or OpenMP, which the runtime then schedules alongside Julia kernels in a unified task graph. Together, these APIs and kernel pathways provide a flexible, high-performance, and portable programming model for heterogeneous computing.

4 Results

The Julia–IRIS integration automatically provisions heterogeneous memory handlers. On the AXPY benchmark, a direct Julia kernel call outperformed the Julia with IRIS-backed version by a small margin; however, when the Julia code is executed through the IRIS

scheduler the overall runtime is comparable for realistic input sizes ‘$\approx 10^6$‘. Moreover, the kernel written in Julia/JACC is type-invariant, accepting FP64, FP32, and INT32 arguments without recompilation. Compared with IRIS native calls, Julia kernels—whether written in pure Julia or through JACC—obtain higher performance thanks to the LLVM JIT’s ability to specialize code for the target device at runtime, thereby eliminating the overhead that typically plagues static vendor wrappers.

JMATRIS, a heterogeneous math library built atop Julia and IRIS, demonstrates that the same approach scales to large, compute-dense workloads. Using 2-D, 1-D, and 3-D tiling macros, load distribution for BLAS routines is automated, and the library can expose iterators and groups that allow users to implement arbitrary tiled algorithms. In a DGEMM test on a $32kx32k$ matrix, JMATRIS.DGEMM achieved performance indistinguishable from MATRIS.DGEMM, reaching 62 TFLOPS on four NVIDIA A100 GPUs. This performance is achieved by delegating device-level GEMM calls to vendor-optimized BLAS kernels while retaining Julia-managed orchestration and tiling logic.

5 Summary

We integrated Julia’s high-performance JIT-compiled language with the IRIS heterogeneous runtime, yielding a dual-interface API that automatically manages memory, data-flow, and scheduling across CPUs, GPUs, FPGAs, and DSPs. Benchmarks show that Julia kernels, whether written in pure Julia or via JACC, match or exceed native IRIS performance on AXPY and maintain comparable performance on larger DGEMM workloads through a tiled math library that exploits vendor BLAS kernels. This combination delivers a unified, type-invariant, and developer-friendly platform for portable, high-performance heterogeneous computing.

References

[1] Rabab Alomairy, Felipe Tome, Julian Samaroo, and Alan Edelman. 2024. Dynamic task scheduling with data dependency awareness using julia. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[2] Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S Vetter. 2021. IRIS: A portable runtime system exploiting multiple heterogeneous programming systems. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.