URSI GASS 2021, Rome, Italy, 28 August - 4 September 2021

High-Performance Radio Telescope Array Data Processing Framework

Max W. Hawkins*(1), Daniel J. Czech(2), David H.E. MacMahon(2), Steve Croft(2,3), and Andrew P.V. Siemion(2,3)
(1) University of Alabama, Tuscaloosa, AL, 35401; email: mhawkins2@crimson.ua.edu
(2) University of California at Berkeley, Berkeley, CA, 94720; email: danielc@berkeley.edu; davidm@berkeley.edu
(3) SETI Institute, Mountain View, CA, 94043

Abstract

As radio telescope projects grow larger with more antennas observing wider bandwidths, data rates are rapidly increasing. The Square Kilometer Array and other next generation observatories will usher in an era of exascale data and beyond [1]. This necessitates an equivalent increase in data transfers, processing speeds, and emphasis on real-time analysis. Reducing the performance gap between high-level science algorithm development (frequently in Python) and real-time, production code would allow astronomers to better utilize the hardware available to them. To enable this, we create a high-level array data processing pipeline framework in the Julia programming language, featuring templates for modular data processing algorithms. We demonstrate its performance with a spectral kurtosis algorithm and show that the new interface does not introduce significant processing overhead. In future work, we will explore the signal processing potential of new hardware accelerators present in modern GPUs. Such accelerators promise improved performance along with new programming challenges.

1 Introduction

Currently, most radio astronomy data is reduced at the facility of origin before being sent to an astronomer for final processing. This data reduction is necessary for the final data volumes to be manageable, but it can also eliminate potential signals present in the raw data. As data rates increase, more scientific analysis must thus be done in real-time at the observatory. Our research will ultimately grant astronomers with limited programming knowledge the ability to write and integrate custom scientific algorithms into raw radio facility data processing pipelines.

1.1 Real-Time Data Processing

Most processing at other radio observatories follows, at the highest levels, a somewhat similar design: digitize, channelize, reduce, and store. The increased usage of commodity digital hardware and necessity for distributed computing will likely result in a trend in observatory pipelines to look more like the MeerKAT and SKA designs [2]. The Very Large Array is adding Ethernet multicast capabilities [3], and many other facilities are increasing the role of GPUs in their computing infrastructure [4].

At the MeerKAT array, our initial target deployment observatory, the real-time data pipeline runs as follows: After coarse channelization, packets of complex antenna voltage data are streamed to distributed compute nodes over a network interface where they are reassembled into sequential blocks of data and topped with a small header of metadata. Next, the data is reduced in some preselected fashion, stored in an on-site data silo for temporary storage, then eventually disseminated for final storage or processing.

Various low-level frameworks have been created to ease the creation of these data pipelines. HASHPIPE [5], PSRDADA [6], kotekan [7], and Pelican [8] are all low-level frameworks written in C or C++ for radio astronomy. As our work was initiated at the Berkeley SETI Research Center[1] and aimed for initial use at the MeerKAT array, we chose to base development on HASHPIPE.

HASHPIPE is an evolution of the GUPPI data acquisition pipeline, is written in C, and is used in various stages of deployment at the Green Bank Observatory, Parkes, MeerKAT, HERA, and the Allen Telescope Array [5]. It creates shared ring buffers, assembles network packets, manages processing threads' access to data through semaphore arrays, and continually updates block metadata and a global status buffer. HASHPIPE is performant and allows for custom plugin creation, but its inherently low-level nature makes it inaccessible to many astronomers who program primarily in Python. To allow for easier future development, extensibility, and astronomer use, a high-level framework is necessary.

There has been at least one such manifestation of this idea in the past: Bifrost, which is a framework deployed at the Long Wavelength Array in New Mexico [9]. However, it relies on Python as the high-level interface. We currently find Julia a more preferred language for future work.

[1] http://seti.berkeley.edu

1.2 Julia

The Julia programming language was chosen for this project because it offers a high-level interface that is very similar to Python, a common language used by many astronomers, while also bringing performance that often rivals C. Julia is dynamically typed, just-in-time compiled, features multiple dispatch, and has powerful REPL and interactive web notebook environments [10]. Its primary benefit over Python is its speed. As a compiled language (using LLVM), Julia's performance is often more comparable to C than Python. Julia also has capabilities to directly call or be called from Python and C code. This enables an easy transition for astronomers who do not have to forego premade astronomical Python libraries.

Additionally, Julia has excellent high and low-level GPU interfaces. By efforts to include GPU support into the Julia compiler toolchain, GPU programming in Julia can be simple and performant [11]. Python does have similar acceleration libraries available like Numba or CuPy, but these are implemented in a fundamentally different way. Julia's use of LLVM, which natively targets both CPUs and GPUs, requires much less device-specific code and enables a more wholistic compilation process. For more details, refer to Besard [11], which outlines many of the reasons we believe Julia has great potential in the future of radio astronomy data processing.

2 High-Level Framework

To create a starting point to assess the feasibility of our high-level framework, lightweight Julia wrapper functions and structs were written to provide the functionality of HASHPIPE. Current work is being done to integrate the Julia and C interfaces more seamlessly through cross-compiled binaries. Beyond an identical set of functions, the Julia API enables additional improvements by leveraging the expressive and data-focused nature of the language. For example, HASHPIPE uses the legacy hget/hput library to update header records. By using multiple dispatch functionality, a set of header functions that depend on input object type can be coalesced into a single function. This functionality eliminates a potential source of error but is not possible in C/C++.

To showcase this new Julia interface, we created and tested a demo HASHPIPE processing thread that interacts with a standard C-level HASHPIPE pipeline (Figure 1). This setup was meant to mimic what an astronomer might program as the final step in an existing real-time data processing pipeline at a telescope array.

Data is sent to a processing node via UDP packets that are then reassembled into a common astronomy data format by a HASHPIPE C-thread. Then, when this thread signals a portion of data is ready, the waiting Julia HASHPIPE thread takes ownership of the data, asynchronously transfers it to the GPU, runs the astronomer-selected algorithm on the data, and transfers the results back to the CPU. This cycle starts over again when the Julia thread frees the original CPU data for the preprocessing thread to overwrite with new data. The HASHPIPE status buffer must be updated by all threads during the entire process.

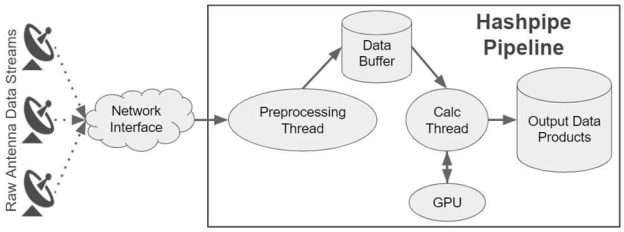

Figure 1. Flow diagram of a commensal MeerKAT data pipeline

In total, the code necessary to create the custom processing thread that interacts with an existing C-level HASHPIPE pipeline is ~120 lines – much of which can be used as a template for any thread. Our benchmarks also showed this code was sufficiently performant. The Julia processing thread that does not compute or transfer any data (null thread) runs in less than 0.5 milliseconds per data block. Compared to an equivalent C-only thread, the Julia thread incurred a negligible compute time penalty on the order of 10s of microseconds. By running our test thread setup, we validated the feasibility of a high-level data pipeline framework integrating with existing real-time HASHPIPE pipelines. Current development work involves creating an offline thread template for post-processing on personal computers to speed up astronomers' day-to-day processing. This will reduce the barrier of entry into our codebase and serve a more common workflow.

Next, we developed an algorithm template for our framework. The goal is to have a modular set of performant algorithms that can be chosen and chained together by an astronomer. Rather than each astronomer having their own custom algorithms, there can be a standard set of publicly available algorithms that can be selected from and extended as necessary. Further, by doing this in the Julia programming language, a high-level GPU interface resembling common CPU Python code is provided directly to astronomers and allows for higher performance code development. If further speed is needed, the Julia language and LLVM compiler chain allow for low-level tweaks. In the future, we also plan on adding automatic memory cost calculations to determine if and how many simultaneous sets of the calculation pipelines can be run on the given hardware. This is especially beneficial for offline processing and can mitigate data transfer dead time – enabling better hardware utilization. With low arithmetic intensity operations, data transfers become the performance bottleneck.

To showcase this library, we created an example algorithm using spectral kurtosis (SK) - a signal discriminator based on the gaussianity of radio signals [12]. Naïve CPU and GPU SK implementations were written to showcase Julia's

expressibility. The only difference between the two functions was a single conversion to interpret the data array as either a normal array or a CUDA array. We then integrated these algorithms into the new Julia test pipeline and benchmarked their performance. Our test system included dual-socket Intel Xeon Silver 4110s, 96 gigabytes of RAM, and an NVIDIA 1080Ti GPU. The data we calculated spectral kurtosis on was very similar to the output data of the MeerKAT telescope array and was processed in 128 megabyte blocks of complex, 8-bit voltage samples across 2 polarizations, 32,768 samples, 16 coarse channels, and 64 antennas. Both spectral kurtosis algorithms calculated SK over the entire time window (largest integration length possible).

Table 1. Performance comparison of Julia spectral kurtosis implementations on 128 MB data blocks

Implementation Type	*Time per Block (ms)*	*Throughput (GBs^{-1})*
CPU	400	0.313
GPU	21	5.95

Table 1 shows the performance results of this experiment. By editing a single line of code and utilizing a functionality of Julia's GPU interface, we showed a 19x performance increase. This experiment highlights the possibilities Julia offers as a high-level language. An astronomer can greatly accelerate parallelizable algorithms easily. Although similar processes may be applied in Python, future work discussed in the next section will show the low-level integration capabilities other languages do not offer.

3 Tensor Cores for Signal Processing

With a high-level GPU API available to astronomers, work must be done to ensure efficient utilization of new generations of hardware. Specifically, future work will involve developing high-level astronomical algorithms in Julia that use tensor cores. This will further increase potential performance gains of our framework. With closer integration into the compilation sequence and native targeting of hardware accelerators, Julia is preferred over Python for this work [11].

Tensor cores are hardware accelerators in NVIDIA GPUs that execute fused matrix multiply-add calculations in a single clock cycle. They can greatly accelerate matrix-heavy algorithms and are especially beneficial for low precision data. On NVIDIA's A100 GPU, using tensor cores increases computing throughput on full precision floating point data by a factor of 8. For 8-bit data, the throughput increase is 32x [13]. Sub-byte data types offer further improvements but are still under active development.

Although intended for machine learning, tensor cores have potential to accelerate many signal processing algorithms. Radio astronomy's low precision (8-bit or lower) lends itself well to tensor core use while common array algorithms like correlation rely heavily on matrix multiplication/addition. Previous work has shown 100x increases in correlation throughput when compared to default CUDA floating point arithmetic [14].

However, because of their machine learning origin, digital signal processing algorithms have been relatively slow to adopt tensor core use. Support for optimized low-precision, complex number operations are slower to be added than common machine learning tasks. Julia 1.5 only supports half-precision base tensor core instructions, and CUBLAS and CUDA gemm support for integer tensor core processing is currently not fully developed. We aim to include 8-bit tensor core functionality in Julia and from there, build out high-level signal processing algorithms.

4 Conclusion

The future of radio astronomy signal processing holds many foreseeable trends like larger data volumes, more array-based observations, and increasing reliance on digital hardware for processing. Combined with external trends of cheaper commodity compute, the proliferation of hardware accelerators, and improved compiler toolchains, the necessity and usability of a high-level data pipeline framework becomes apparent. We present a data pipeline framework in Julia that preserves the performance of the underlying low-level code while providing an accessible high-level interface to enable astronomer access to on-site, real-time processing. This could prove especially useful to transient searches (FRBs or SETI), all-sky surveys, and any commensal array observation.

5 Acknowledgements

Max Hawkins was supported by the National Science Foundation under the Berkeley SETI Research Center REU Site Grant No. 1950897.

Breakthrough Listen is managed by the Breakthrough Initiatives, sponsored by the Breakthrough Prize Foundation.

6 References

[1] A. M. M. Scaife, "Big telescope, big data: towards exascale with the Square Kilometre Array," *Philos. Trans. R. Soc. Math. Phys. Eng. Sci.*, vol. 378, no. 2166, p. 20190060, Mar. 2020, doi: 10.1098/rsta.2019.0060.

[2] J. R. Manley, "A scalable packetised radio astronomy imager," 2015, Accessed: Jan. 30, 2021. [Online]. Available: https://open.uct.ac.za/handle/11427/15573.

[3] J. Hickish *et al.*, "Commensal, Multi-user Observations with an Ethernet-based Jansky Very Large Array," *ArXiv190705263 Astro-Ph*, Jul. 2019,

Accessed: Jan. 30, 2021. [Online]. Available: http://arxiv.org/abs/1907.05263.

[4] D. C. Price, J. Kocz, M. Bailes, and L. J. Greenhill, "Introduction to the Special Issue on Digital Signal Processing in Radio Astronomy," *J. Astron. Instrum.*, vol. 05, no. 04, p. 1602002, Dec. 2016, doi: 10.1142/S2251171716020025.

[5] D. H. E. MacMahon *et al.*, "The Breakthrough Listen Search for Intelligent Life: A Wideband Data Recorder System for the Robert C. Byrd Green Bank Telescope," *Publ. Astron. Soc. Pac.*, vol. 130, no. 986, p. 044502, Apr. 2018, doi: 10.1088/1538-3873/aa80d2.

[6] J. Kocz *et al.*, "Digital Signal Processing Using Stream High Performance Computing," *J. Astron. Instrum.*, vol. 04, no. 01n02, p. 1550003, Mar. 2015, doi: 10.1142/S2251171715500038.

[7] A. Recnik *et al.*, "An efficient real-time data pipeline for the CHIME Pathfinder radio telescope X-engine," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, Jul. 2015, pp. 57–61, doi: 10.1109/ASAP.2015.7245705.

[8] B. Mort, F. Dulwich, C. Williams, and S. Salvini, "Pelican: Pipeline for Extensible, Lightweight Imaging and CAlibratioN," *Astrophys. Source Code Libr.*, p. ascl:1507.003, Jul. 2015.

[9] M. D. Cranmer *et al.*, "Bifrost: A Python/C++ Framework for High-Throughput Stream Processing in Astronomy," *J. Astron. Instrum.*, vol. 06, no. 04, p. 1750007, Sep. 2017, doi: 10.1142/S2251171717500076.

[10] J. Bezanson *et al.*, "Julia: dynamism and performance reconciled by design," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, p. 120:1-120:23, Oct. 2018, doi: 10.1145/3276490.

[11] T. Besard, C. Foket, and B. D. Sutter, "Effective Extensible Programming: Unleashing Julia on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 4, pp. 827–841, Apr. 2019, doi: 10.1109/TPDS.2018.2872064.

[12] G. M. Nita and D. E. Gary, "The generalized spectral kurtosis estimator: The generalized spectral kurtosis estimator," *Mon. Not. R. Astron. Soc. Lett.*, Jun. 2010, doi: 10.1111/j.1745-3933.2010.00882.x.

[13] V. Sarge and M. Andersch, "Tensor Core Performance: The Ultimate Guide," p. 36.

[14] J. Romein, "Tensor Cores: Signal Processing at Unprecedented Speeds," presented at the GTC Silicon Valley Session S9306, 2019.