Performance Enhancement of First COSI Data Challenge

Jamie Shin (*jamie.s@wustl.edu*), Jeremy Buhler, Marion Sudvarg, on behalf of the APT Collaboration

Washington University in St. Louis
James McKelvey School of Engineering

Funded by NASA award 80NSSC21K1741 and NSF award CNS-1763503

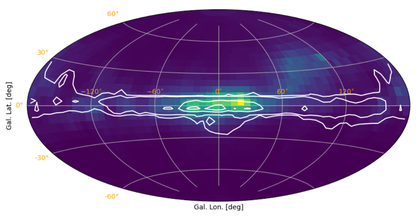
1 Introduction

COSI, the **Co**mpton **S**pectrometer and **I**mager [1], is a planned 0.2-0.5 MeV Compton telescope. Upon its launch in 2027, COSI will study gamma-ray sources within and beyond the Milky Way. The 2023 COSI Data Challenge (DC1) [2] asks the astrophysics community to image weak and diffuse sources, such as the Crab Nebula and the galactic plane, using gamma-ray data collected during a 2016 balloon flight.

DC1 exhibits many similarities to imaging computations that we expect to arise from our group's planned Advanced Particle-astrophysics Telescope (APT) and its balloon-borne Antarctic demonstrator (ADAPT) [*see poster #255*]. When we performed the DC1 imaging computations with the (mainly Python) code provided by COSI, we found that they required hours of computing time even on a large multicore server. We therefore sought to reduce the cost of these computations through parallelism and refactoring, while remaining in Python to retain the DC's ability to engage the broader astrophysics community as originally intended.

DC1 Image Reconstruction

- **Data D**: gamma-ray counts observed over time, binned according to properties of their Compton rings
- **Response R**: simulated detector response to gamma rays from different directions in sky
- **Goal**: estimate source image S *and* background B (all light not from source) that best explain data D as $R \otimes S + B$.
- Uses **Richardson-Lucy (R-L)** iterative deconvolution [5,6]



Reconstructed image of Al26 emissions from galactic plane, computed with DC1 code

Key Computational Bottlenecks in DC1 R-L Code

DC1 Imaging Task	511keV	Al26	Point Sources
R-L Total Time	**8h 30m**	**6h 16m**	**8h 22m**
R-L Iterations	150	77	150

Time to complete DC1 R-L block on AMD EPYC 7551P, 256 GB DRAM, Python 3.8.10

(1) Two **convolution kernels** (written in Python)
Cost in 511keV task is ~**140s** per iteration

(2) **Background fitting** uses numerical max likelihood estimation
Calls external Stan C++ optimizer – ~**60s** per iteration

(3) Code is **single-threaded**, even on our 32-processor CPU

2 Acceleration Strategies

1. JIT Compilation

- **Numba** library [3] can just-in-time compile NumPy code to *faster native code*
- Can easily *parallelize* across multiple CPU cores with "prange" construct

```
def convdelta(D,W,n_b,n_l):
    R = 0
    for i in range(n_b):
        for j in range(n_l):
            R += D[:,i,j,:] * W[i,j]
    return R
```

```
@njit(fastmath=True,parallel=True,nogil=True)
def convdelta(D,W,n_b,n_l,n_x,n_y):
    R = np.zeros((n_x,n_y))
    for c in prange(n_x):
        for i in range(n_b):
            for d in range(n_y):
                for j in range(n_l):
                    R[c,d] += D[i,c,d,j] * W[i,j]
```

- For best performance, must rewrite vectorized operations as nested loops
- Observed **40-60x speedup (on 8 CPU cores)** vs. original code for convolution kernels

2. JAXopt for Maximum Likelihood Estimation

- JAXopt optimizer backend [4] both issues **parallel function evaluations** *and* JIT-compiles Python objective function
- Also achieves **faster convergence** than Stan
- Requires *side-effect-free* rewrite of objective function

```
# objective function for MLE
def objective(params, data):
        (Abg, flux) = params
        (expc, cdelta, bg_model, bg_idx_arr, y, mu_flux, sigma_flux, mu_Abg, sigma_Abg) = data

        M = Abg[bg_idx_arr[:,None]] * bg_model + flux[0] * expc + flux[1] * cdelta

        # ensure that we don't accidentally use negative Possion means, which blows up likelihood
        M = jnp.maximum(M, 0)

        lp = jnp.sum(jstats.poisson.logpmf(y, M), axis=None) + \
             jnp.sum(jstats.norm.logpdf(flux, mu_flux, sigma_flux)) + \
             jnp.sum(jstats.norm.logpdf(Abg, mu_Abg, sigma_Abg))

        return -lp  # minimize to maximize LL
```

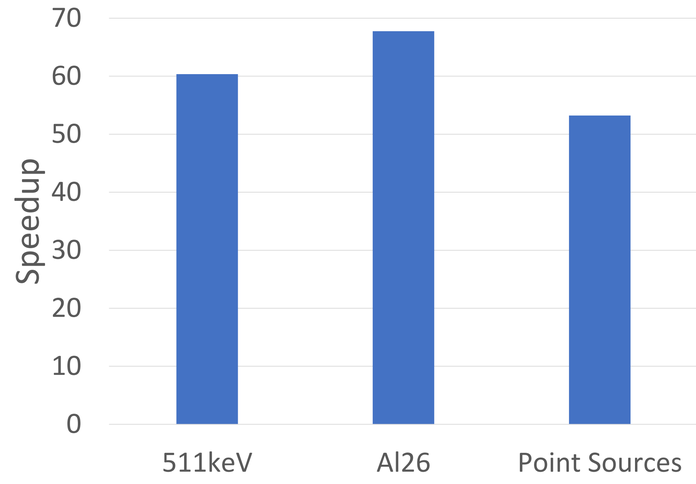
typical speedup vs. original: 12-60x (on 8 CPU cores)

3. Memory Reduction

- Simulated sky response matrix R is used in every convolution operation
- Originally stored in double precision → 70-100 GB of memory needed
- Comparable results achieved with single precision → **50% memory savings**
- (Now experimenting with half precision to target GPU implementation)

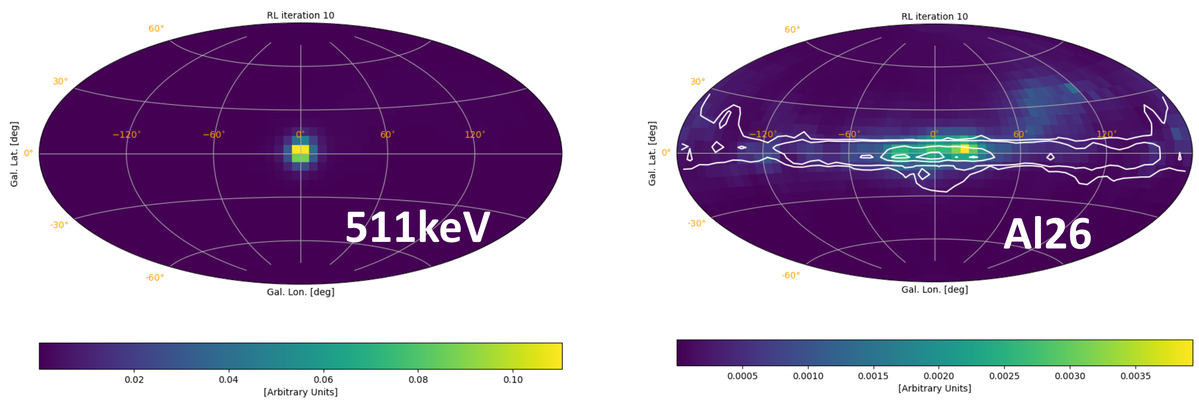
3 Results

We tested three DC1 imaging tasks using our accelerated Python implementation of R-L on 8 CPU cores. We measured speedups vs. (unparallelized) DC1 original code, keeping number of R-L iterations the same for each.



Average speedup for 3 DC1 imaging tasks: 60.1x

*For two of three tasks (511keV and Al26), switching to JAXopt-based optimizer caused R-L to converge to essentially the same result as the original in **many fewer** iterations, providing further **7-14x** speedup.*



Results after 11 R-L iterations – similar images, MAP likelihoods to results after 150 iterations of original DC1 R-L code

*R-L does not offer a good stopping criterion to terminate iteration. Continuing to run further improves MAP likelihood but **overfits background** and removes much of the actual source intensity. Finding a good stopping rule for image deconvolution is left to future work.*

4 Conclusion & Future Work

- Can **achieve parallelism, large speedups** on R-L imaging without leaving Python
- **Convergence with R-L is tricky** – faster code supports fine-tuning of approach
- To investigate: how will these techniques impact performance of *Second COSI Data Challenge* (released 3/25/2024)?

Prospects for GPU Acceleration

- JAXopt transparently uses GPU, but convolutions need JIT compiler such as CuPy [7]
- Response matrix R is **tens of gigabytes** in size; even at 16-bit precision, may not fit on GPUs available to most researchers
- Ordering of response matrix dimensions impacts **locality** of GPU memory access
- Empirically, CuPy performance and memory usage is highly sensitive to exact vector operations used to express convolutions; writing "raw" kernels may be preferable

References

[1] J. Tomsick, A. Zoglauer, C. Sleator, H. Lazar, et al. "The Compton Spectrometer and Imager." *Bulletin of the American Astronomical Society* 51(7):98, 2019.
[2] C. Karwin, S. Boggs, J. Tomsick, A. Zoglauer, et al. "The COSI Data Challenges and Simulations." *Bulletin of the American Astronomical Society* 54(3):2022n3i108p30, 2022.
[3] S. K. Lam, A. Pitrou, and S. Seibert. "Numba: a LLVM-based Python JIT Compiler." *Proc. 2nd Wkshp LLVM Compiler Infrastructure in HPC* 7, New York, NY 2015.
[4] M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, et al. "Efficient and Modular Implicit Differentiation." *Advances in Neural Information Processing Systems* 35, 2022.
[5] W. H. Richardson. "Bayesian-based Iterative Method of Image Restoration." *J. Optical Society of America* 672(1):55, 1972.
[6] L.B. Lucy. "An iterative technique for the rectification of observed distributions." *Astronomical J.* 79(6):745-754, 1974.
[7] R. Okta, Y. unno, D. Nishino, H. Shohei, et al. "CuPy: a NumPy-Compatible Library for NVIDIA GPU Calculations." *Proc. Wkshp. Machine Learning Systems at NeurIPS 31*, 2017.