GPUs and Python: A Recipe for Lightning-Fast Data Pipelines

Craig Warner
Christopher Packham
Stephen Eikenberry
Anthony Gonzalez

University of Florida

Astronomical amounts of data!

- Volume of data produced per night is increasing rapidly as arrays increase their pixel numbers and mosaics of arrays become more common.
- Looking forward, the Large Synoptic Survey Telescope (LSST) is expected to produce 30 TB of data per night!
- Current data reduction pipelines are unable to handle this amount of data flow.
- New streamlined and rapid data reduction processes are thus critical.

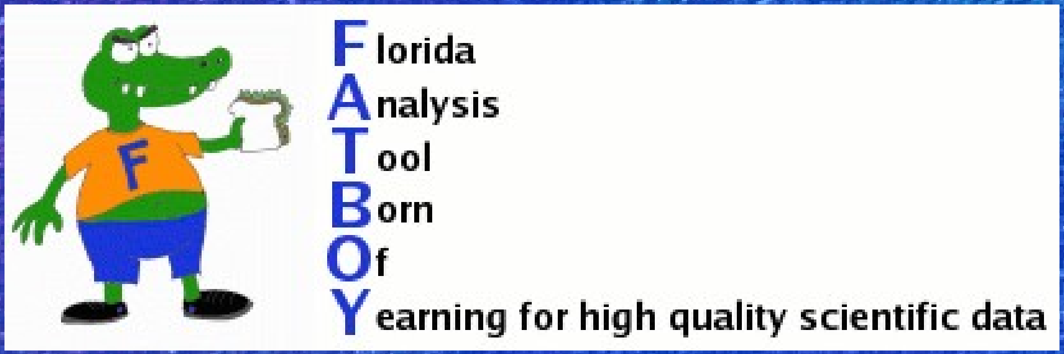
GPUs: A possible solution?

- Modern Graphics Processing Units (GPUs) contain hundreds of processing cores, each of which can process hundreds of concurrent threads

- Nvidia's Compute Unified Device Architecture (CUDA) platform allows developers to design massively parallel algorithms for their GPUs
- Parallelizing algorithms for GPUs can provide speed-ups of up to around 100X!!!

A Perfect Recipe

- Data pipelines are perfectly suited for massive parallelization because many algorithms are performed on a per-pixel basis.
- The PyCUDA module and python's native C-API allow CUDA code to be easily integrated into existing python data pipeline frameworks.
- We use an Nvidia 580 GTX for our tests



PyCUDA Samples

- PyCUDA's SourceModule allows CUDA code to be compiled and easily linked into python code

```
UFGpuOps_mod = SourceModule("""
__global__ void gpu_linearity_float(float *output, float *input, float *coeffs, int ncoeffs) {
        const int i = blockDim.x*blockIdx.x + threadIdx.x;
        int n = 1;
        output[i] = input[i]*coeffs[0];
        for (int j = 1; j < ncoeffs; j++) {
            n++;
            output[i] += coeffs[j] * pow(input[i], n);
        }
}
""")
```

- The above CUDA code will be compiled at import time and can be called as a python method

```
gpu_linearity = UFGpuOps_mod.get_function("gpu_linearity_float")
output = empty(data.shape, "Float32")
gpu_linearity(drv.Out(output), drv.In(data), drv.In(coeffs), int32(ncoeffs), grid=(blocks,1),
                block=(block_size,1,1))
```

CUDA and Python's C-API

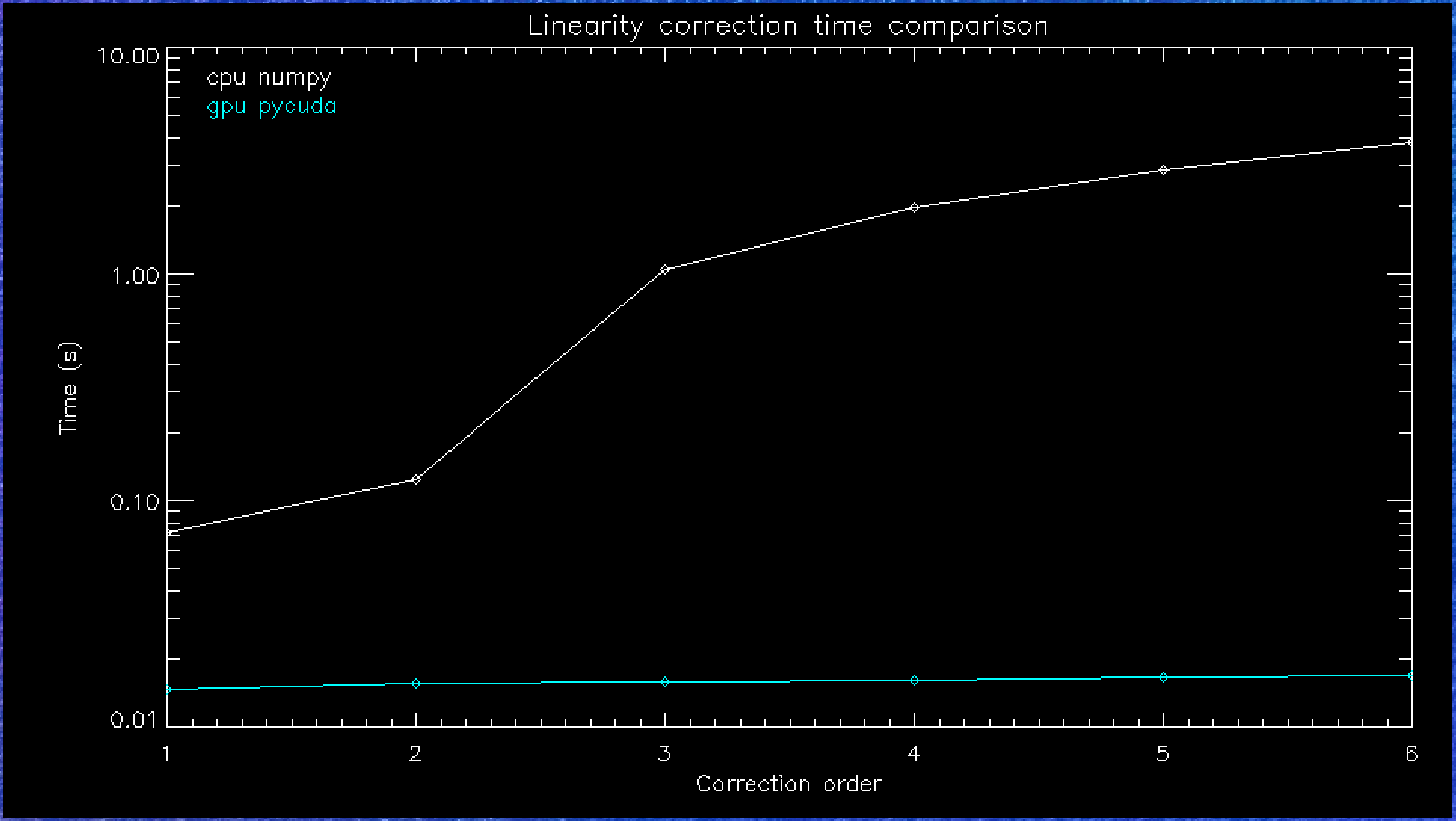
- Python's C-API can also be used to link in compiled C code with CUDA library calls

```
#include <thrust/device_vector.h>
#include <thrust/sort.h>
extern "C" {
  static PyObject * gpumedian(PyObject *self, PyObject *args, PyObject *keywds);
   void gpusort_float(float *data, int n) {
    thrust::device_vector<float> d_x(data, data+n);
    thrust::sort(d_x.begin(), d_x.end());
    thrust::copy(d_x.begin(), d_x.end(), data);
   }
  static PyObject * gpumedian(PyObject *self, PyObject *args, PyObject *keywds) {
  ... }
}
```

- First compile the .cu file with nvcc into a shared object. Then use g++ to link the .so file with libcuda and libcudart into a library that can be imported into python.

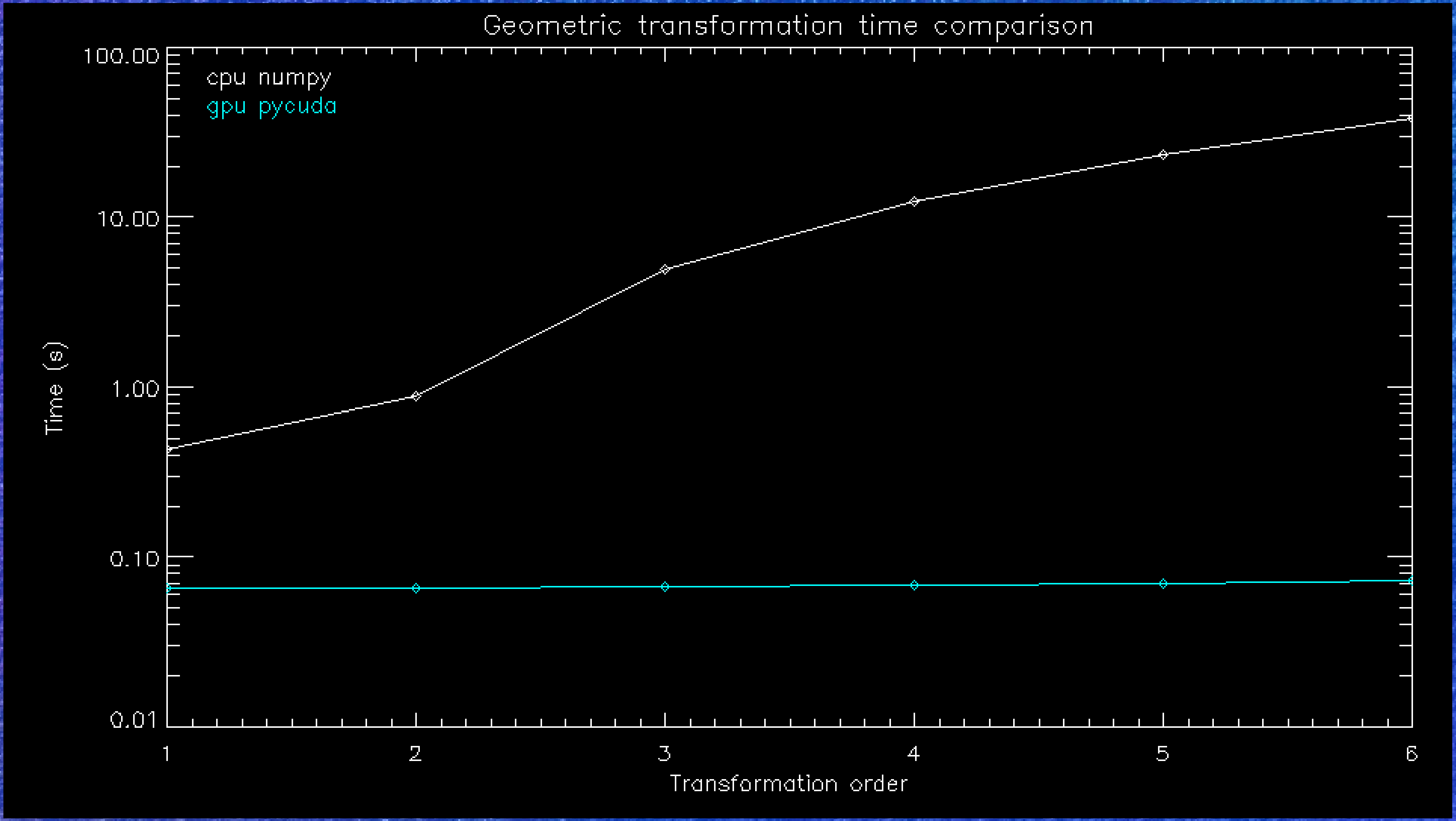
Results: Linearity correction

- 3^{rd} order linearity correction: 66 X faster!



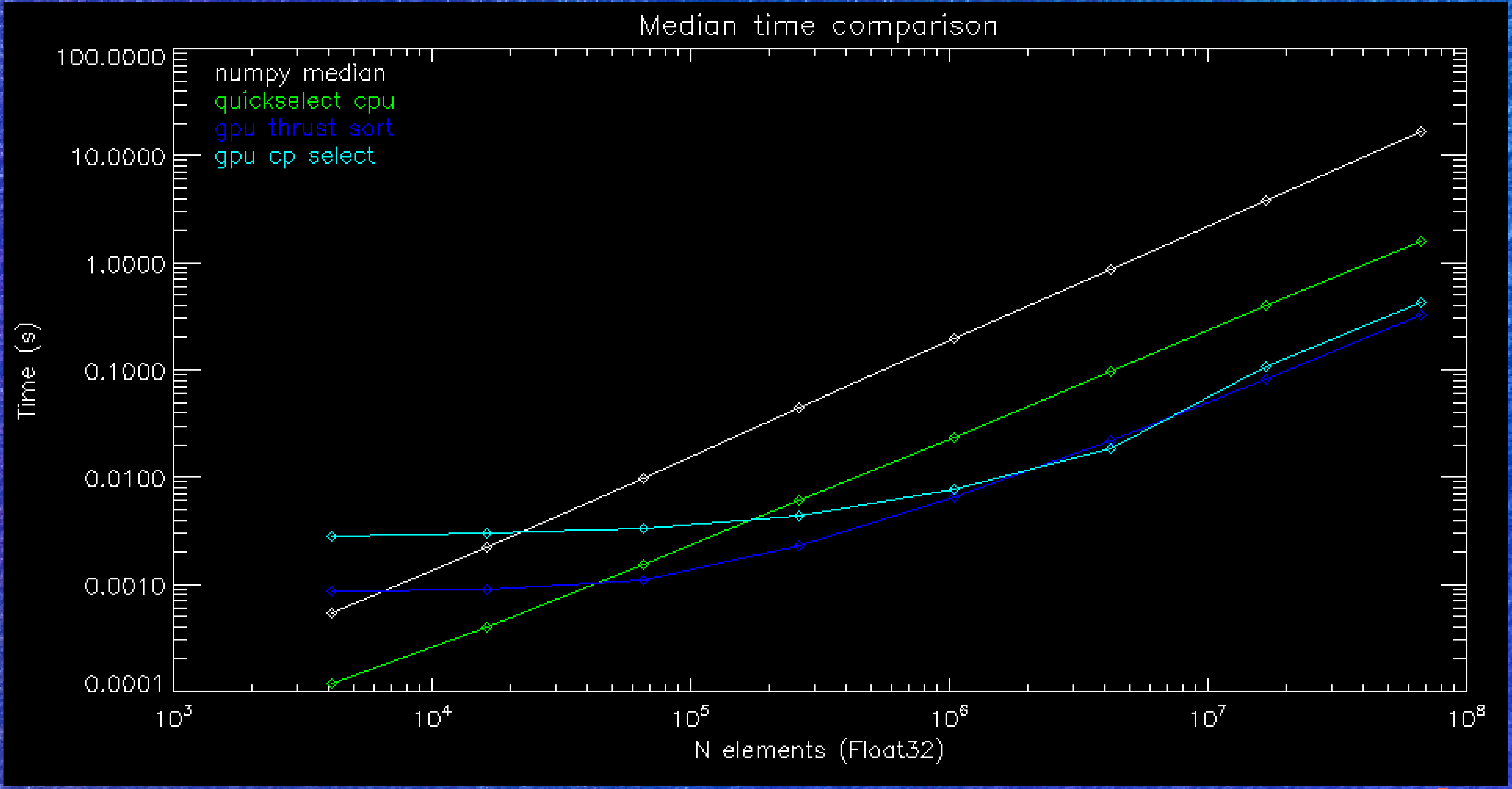
Results: Geometric transformation

- 5^{th} order geometric transformation: 339 X faster!!



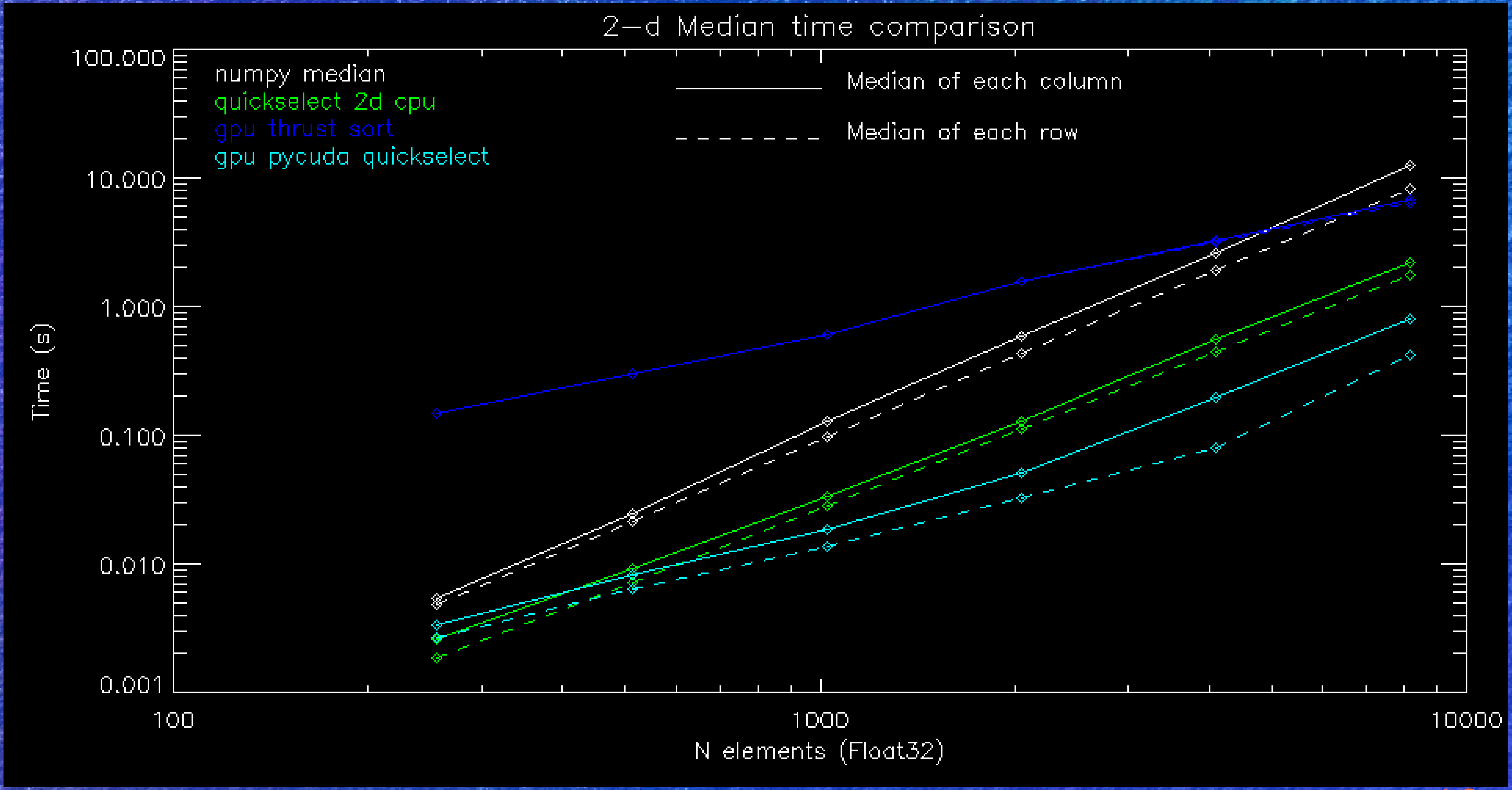
Results: 1-d median

- Median of 2048x2048 image: gpu thrust sort is 40 X faster than numpy's median (uses numpy's sort) and 4.4 X faster than C quickselect.



Results: 2-d median

- Median of rows in 2048x2048 image: PyCUDA quickselect implementation is 13.2 X faster than numpy and 3.5 X faster than C quickselect.



Comparisons: GPU FTW again!

Cosmic Ray Removal

Python	GPU
1.503s	0.048s

Finding shifts between images with xregister using full 2048x2048 frame

# of images	IRAF	Python	GPU
9	364.2s	169.9s	7.46s
23	912.6s	455.1s	19.42s

Drizzling images onto output grid while applying a 6th order geometric distortion correction and subpixel shifts between images

# of images	Drizzle kernel	IRAF drizzle	Python drihizzle	GPU drihizzle
9	point	139.44s	78.94s	2.00s
9	turbo	143.67s	126.20s	2.09s
23	point	371.95s	141.35s	5.17s
23	turbo	387.96s	261.00s	5.35s

Imcombine and Overall Results

Median combining images using 3 implementations of imcombine with different weightings and rejection criteria

# images	weight	reject	IRAF	Python	GPU
9	none	none	4.22s	2.46s	0.62s
9	median	none	4.60s	10.33s	1.12s
9	none	sigclip	5.53s	6.50s	0.63s
9	median	sigclip	6.71s	17.48s	1.14s
23	none	none	5.39s	8.00s	2.46s
23	median	none	10.64s	27.29s	4.17s
23	none	sigclip	16.18s	27.70s	2.71s
23	median	sigclip	24.60s	49.46s	4.29s

Comparison of overall times to process test data set:
Preliminary results are a speed up of 12 X with 1-pass sky subtraction and 7 X with 2-pass.

CPU 1-pass	GPU 1-pass	GPU 1-pass BE	CPU 2-pass	GPU 2-pass
754.8s	62.4s	75.5s*	1035.5s	155.2s

*BE = big endian – we achieve a 20% speed increase by overriding pyfits to save images in little endian format, avoiding the need to byteswap.

Implications and Future Work

- With further optimization, we believe it is possible to achieve an overall speed gain of up to a factor of 25!
- We believe we can achieve a similar speed gain by GPUizing spectroscopy algorithms.
- This factor would only increase as larger array sizes and newer GPUs provide for even higher degrees of parallelization.
- A speed gain of this magnitude would allow for near real-time data processing, concurrent with continuing observations, considerably optimizing the observing process!

Super-FATBOY??