Computer Science Honours
Final Paper
2015

Title: GPU-Acceleration of the SoFiA Smooth-and-Clip source finding algorithm with CUDA

Author: Jarred de Beer

Project Abbreviation: GPUStar

Supervisor: Michelle Kuttel

Category	**Min**	**Max**	**Chosen**
Requirement Analysis and Design	0	20	0
Theoretical Analysis	0	25	3
Experiment Design and Execution	0	20	12
System Development and Implementation	0	15	10
Results, Findings and Conclusion	10	20	20
Aim Formulation and Background Work	10	15	15
Quality of Paper Writing and Presentation	10		10
Adherence to Project Proposal and Quality of Deliverables	10		10
Overall General Project Evaluation (*this section allowed only with motivation letter from supervisor*)	0	10	
Total marks		80	80

GPU-Acceleration of the SoFiA Smooth-and-Clip source finding algorithm with CUDA

Jarred de Beer
University of Cape Town

ABSTRACT

Location of sources in astronomical data cubes will become more costly as these cubes increase in size in future H1 surveys. In this project we accelerate the Smooth-and-Clip algorithm of the SoFiA source finding package using GPU processing with CUDA. SoFiA is developed as a general source finding package to be used in future H1 surveys at ASKAP and the Smooth-and-Clip implementation performs multiple Gaussian and Uniform filters using Scipy. We duplicate the Gaussian and Uniform execution from Scipy with serial-C versions achieving the same results but find that it performs at half the speed. The serial-C version is accelerated using OpenMP but focus is on the CUDA implementation developed using the Assess, Parallellise, Optimise, Deploy (APOD) methodology outlined in CUDA's Best Practices handbook. We obtain incremental speedup improvements from 1.5x to 24x for Gaussian and 6x to 15x for Uniform filtering. Integrating this into the SoFiA pipeline results in a reduction in execution time from 535 to 70 seconds using a 2.3GB data cube on a Tesla M2090 device, a speedup of 8.6x. We conclude that GPU processing is advantageous in accelerating the filtering processes of source finding applications.

1. INTRODUCTION

Radio astronomy surveys such as the Australian Square Kilometer Array Pathfinder (ASKAP) and Meer Karoo Array Telescope (MeerKAT) are under development. The surveys detect HI spectral emissions from spiral galaxies and celestial bodies and help scientists understand the cosmic neutral gas density of the universe and its evolution [3]. 'Meer' is Afrikaans for 'more' indicating that more telescopes will be used than previous generation surveys, generating Petabytes of data. MeerKAT telescopes are located in a radio-quiet reserve in the Karoo, powered by an RFI-silent (Radio Frequency Interference) grid [4]. Traditionally, detection of HI spectral emissions has been a manual process but automated solutions are now required. The SoFiA source finding framework is being developed in preparation for various ASKAP surveys: WALLABY, a wide survey covering 3/4 of the sky at z-0.25; DINGO, a deep survey reaching z-0.4; and APERTIF [8].

Data is stored in the 3D Flexible Image Transport System (FITS) format, adequate for MeerKAT and ASKAP surveys which will generate spectral data cube sizes of up to 2.5 Terabytes [1]. The 3D data is represented by two spatial dimensions which map to coordinates in the sky and one spectral dimension which maps to the spectral frequency of HI detections.

Radio frequency interference (RFI) make automated detection of source emissions difficult. Performance is measured by *completeness*: the number of detected sources divided by the total number of sources, and *reliability*: the number of true detections divided by the total number of detections [7]. RFI decreases the signal-to-noise ratio which lowers completeness and reliability and as a result most source finders contain a pre-processing step to smooth noise [7].

SoFiA is intended to search for emissions on multiple scales while also considering variations in noise level [8]. Three source finding algorithms are implemented: Simple threshold, Smooth-and-Clip (S+C), and Characterised noise HI (CNHI). SoFiA's modularity allows additional algorithms to be integrated.

Intensity threshold source finders such as S+C compare pixel values against an absolute threshold to classify source pixels. An inherent limitation is a decreased contribution of total flux in higher resolution data cubes as sources are distributed into a larger number of pixels [5].

SoFiA's pipeline consists of five stages: First, data is input and modified according to flags or weights; Second, filters are applied to reduce noise; Third, source detection which employs various source detection algorithms; Fourth, sources are merged and parametrised; Fifth, the results and binary mask containing sources are output.

Most source finding packages and algorithms execute sequentially and are CPU bound. The Scalable Source Finding Framework (SSoFF) by Westerlund et al. has been developed to distribute workload among a grid based cluster of nodes, but source finders need to be implemented individually. The SSoFF implementation of the Parallel Gaussian source finder (PGSF) is such an example [10]. PGSF was later implemented on the Graphics Processing Unit (GPU) by Westerlund et al. who report significant speedups over the CPU. Similar GPU results have been reported by Huang et al. of a 25x speedup and 9x less energy consumption than multi-threaded CPU implementations.

GPUs are designed for computational graphics and contain dedicated arithmetic units and a variety of memory hierarchies for efficient calculation. Data cubes are transferred in parts onto the GPU where they are processed and transferred back to the host. Processing is performed by thousands of threads which execute in blocks running in lockstep, performing Single Instruction Multiple Data (SIMD) instructions. The block coordinates of a thread are mapped to its pixel index in the data cube, which is loaded and

stored in either of the memory hierarchies. Memory latency is the biggest contributing factor to performance [2]. Wu et al. implemented a GPU version of K-Means which performed 35x faster than a four-threaded CPU implementation. However, the data cube fitted into device memory and performance would reduce significantly if data is too large to fit onto the device [11]. Matrix transposition, to transform row-based matrices into column-based matrices for coherent memory access along columns during filtering has also been performed in-place using the GPU [9].

Compute Unified Device Arthitecture (CUDA) is a technology developed by Nvidia which provides an API to perform general processing on GPUs (GPGPU). A similar, competing API is OpenCL which ran 16% to 67% slower and with greater memory latency than CUDA in all problem sizes [6].

The favorable performance and energy consumption of GPU processing makes it an attractive technology for use in radio astronomy surveys such as MeerKAT and ASKAP. We aim to accelerate the S+C implementation in SoFiA on the GPU using CUDA, and are interested in the amount of speedup and computational throughput obtainable. We hope to decrease astronomer's time-to-result when using the SoFiA package and increase the throughput of ASKAP and MeerKAT surveys.

2. BACKGROUND

2.1 Evaluating parallel speedup

CUDA development time can be optimised by understanding how an application scales to plan an incremental parallelisation strategy [2]. Strong Scaling is given by *Amdahl's law* (Equation 1), providing an upper bound on speedup when the problem size is fixed. Weak Scaling is given by *Gustafson's Law* (Equation 2), providing an upper bound on the problem size when time is fixed. In strong scaling we are interested in how much faster an existing data cube can be processed, decreasing astronomer's time to obtain results. In weak scaling we are interested in the amount of data that can be processed per interval by the the computing infrastructure.

Amdahl's Law is given by Equation 1, in which n is the number of threads and $B \in [0, 1]$ is the portion of strictly serial code. $T(n)$ and $S(n)$ are the time and speedup respectively using n threads.

$$\begin{aligned} \mathrm{T(n)} &= \mathrm{T(1)(B} + \frac{1}{n}(1-B)) \\ \mathrm{S(n)} &= \frac{T(1)}{T(n)} = \frac{T(1)}{T(1)(B+\frac{1}{n}(1-B))} = \frac{1}{B+\frac{1}{n}(1-B)} \end{aligned} \tag{1}$$

Gustafson's Law is given by Equation 2, in which T_s and T_p are the execution time of serial and parallel code respectively.

$$\begin{aligned} &\mathrm{T(n)} = \mathrm{T}_s + n \times T_p \\ &\mathrm{S(n)} = \frac{T(n)}{T(1)} = \frac{T_s + n \times T_p}{T_s + T_p} \\ &\text{Let f} = \frac{T_s}{T_s + T_p} \text{ , then} \\ &\mathrm{S(n)} = \mathrm{T}_s + n \times (1-f) = n - f \times (n-1) \end{aligned} \tag{2}$$

2.2 APOD development

The APOD process is described as follows [2]. In *Assess* the bulk execution time is located to be analysed with strong and weak scaling. *Parallelise* is concerned with either making use of existing GPU-optimised libraries or re-factoring parallelisable code onto the GPU. The developer then *Optimises* the implementation in an iterative process using profiling tools provided by Nvidia. Finally *Deploy* pushes the completed and working implementation to production [2]. This provides an evolutionary rather than revolutionary set of changes, maximising profit and minimises risk [2].

2.3 The Smooth-and-Clip algorithm

The Smooth-and-Clip (S+C) algorithm in SoFiA performs a 1D filter convolution in a specific order along each individual axis. A Gaussian filter is first applied along the Y-axis, then again along the X-axis, and lastly either a uniform or a gaussian filter is applied along the Z-axis. After the data cube has been smoothed the pixel values are fitted against a Gaussian distribution from which the Root Mean Square (RMS) is calculated. The RMS value is the intensity threshold to add pixels to the mask. Algorithm 1 provides pseudocode for the S+C algorithm:

Algorithm 1 Smooth and Clip algorithm

```
1: procedure SMOOTH AND CLIP(in, kernels)
2:     mask ← initMask(in.size)
3:     rms ← getRMS(in)
4:     for k in kernels do
5:         kz ← k[0]
6:         ky ← k[1]
7:         kx ← k[2]
8:         cpy ← copy(in)
9:         cpy ← gaussianFilter(cpy, y, ky)
10:        cpy ← gaussianFilter(cpy, x, kx)
11:        cpy ← uniformFilter(cpy, z, kz)
12:        for i in cube do
13:            if cube[i] > rms then
14:                mask[i] = 1
15:            end if
16:        end for
17:    end for
18:    return mask
19: end procedure
```

SoFiA uses the Gaussian and Uniform filter provided by scipy's image processing module. scipy is a python library with calls to C and Fortran code optimised for compute.

2.3.1 Gaussian filter

The Gaussian filter calculates the weighted average pixel intensity within a given radius of a pixel. Weights are generated according to a 1 dimensional gaussian distribution (see equation 3) which is symmetric and centered at the pixel to be filtered. Weights are multiplied with neighbouring pixels according to their offset from the filtering pixel before being summed and averaged.

$$g(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \quad (3)$$

2.3.2 *Uniform filter*

The Uniform filter calculates the average pixel intensity for a given radius of a pixel. An optimised serial implementation of this algorithm calculates the average of the first pixel and keeps a running window that updates the average at each pixel by removing the oldest and adding the newest value in the window. This technique is described by equation 4:

$$\text{avg[i]} = \sum_{n=-r}^{r} (cube[i+n]/2r)$$
$$\text{avg[i + 1]} = \text{avg[i]} + \text{(cube[i + 1 + r] - cube[i - r]) / 2r} \quad (4)$$

2.3.3 *Root Mean Square*

The Root Mean Square (RMS) method bins pixel values into a histogram and fits the resulting histogram to a Gaussian distribution. The standard deviation of this distribution is returned as the RMS value.

3. DESIGN

We aim to replace CPU intensive filtering from S+C with CUDA implementations in C to be processed on the GPU. The implementation is designed to be integrated back into SoFiA with minimal overhead to its existing pipeline.

We design for individual compute nodes equipped with a CUDA enabled GPU device. When data exceeds device memory, the data cube will be split and processed in parts on the GPU. Compute nodes contain at least 4GB of memory to process data cubes of up to 2GB. Data cubes are in the Flexible Image Transport System (FITS) file format. The implementation is developed against version 0.4.0 of SoFiA's github repository. SoFiA's default parameter set for S+C will be supported and fall back gracefully to original serial execution where either the control flow has not been implemented for the input parameters or no CUDA enabled device is detected.

3.1 Approach

In the first pass, the execution time of the S+C algorithm is analysed to detect slow sub-routines, which are re-implemented with a serial-C library with matching API. Test cases for each implementation verify output against the original version. In the second pass we parallelise the serial-C version in two phases: First with a multi-threaded OpenMP implementation and second with a CUDA implementation.

3.1.1 *System Architecture*

SoFiA's original architecture remains unchanged: Our implementation integrates with SoFiA's pipeline as shown in Figure 1. Here, the left column represents the S+C pipeline written in Python and the right column represents our implementation written in C with its CUDA kernels. The data cube is processed in place and in parts on the host using device memory.

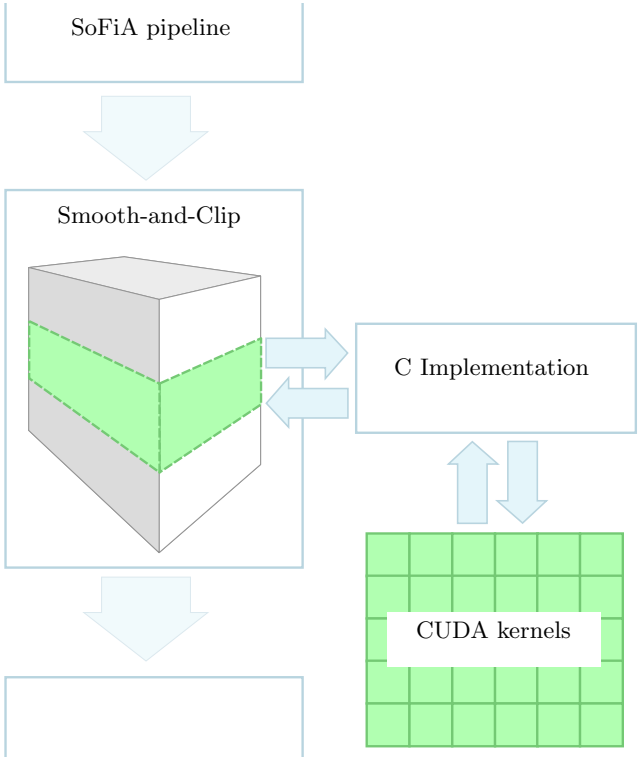


Figure 1: System Architecture displaying S+C as part of SoFiA's pipeline (left column). A pointer to a copy of the cube is passed to our C implementation (right column), data is transferred in parts onto the GPU for processing and returned in-place.

3.1.2 *Software methodology*

Memory is limited and it is important to minimise data redundancy when calling the C module from the existing Python logic. The C Foreign Functional Interface (CFFI) for Python provides a mechanism to call compiled C code from Python with a clean separation and ability to pass pointers to existing data structures which we then process in place to avoid duplicating data. This is achieved using GPU memory as an off-site buffer.

The FITS data cube is loaded into a 3D numpy array and filtered with scipy's image processing module. Convolution from filtering is expected to require the most work and the C module re-implements the execution flow from scipy. Using CFFI we call the implementations with a pointer to the numpy array. This cleanly integrates with the existing Python code and avoids the need to duplicate the data. Each filter is developed and tested in isolation before being integrated and tested with S+C.

3.1.3 *Software development*

We follow an iterative software development life cycle running in phases. A sequential C version is developed initially,

then multi-threaded with OpenMP and finally CUDA kernels. Speedups for the OpenMP and CUDA implementations are measured and the output is tested to ensure values do not change. The CUDA phase implements the APOD iteration loop as specified by CUDA's best practices handbook providing a cyclic and systematic process to achieving performance improvements [2].

3.2 Evaluation

The execution of S+C is analysed using algorithm 1 to measure strong scaling with *Amdahl's law* and weak scaling with *Gustafson's law*. The primary *for loop* on *Line 4* of Algorithm 1 duplicates the data cube on each iteration to be processed independently, which makes it ideal for parallellisation. However, the amount of data duplication would conflict with our limited memory constraint and it must run sequentially. Initialisation on *lines 1 & 2*, filtering on *lines 8 - 11* and clipping on *lines 12 - 16* are parallellisable. The default number of kernels k is 11. Applying *Amdahl's law* from equation 1 we get:

$$\begin{aligned} \mathrm{T}_{S+C} &= T_{init} + k.(T_{copy} + T_{filt} + T_{clip}) \\ \mathrm{B}_{S+C} &= B_{init} + k.(B_{copy} + B_{filter} + B_{clip}) \\ \mathrm{S}_{S+C} &= \frac{1}{B_{S+C} + \frac{1}{n}(1 - B_{S+C})} \end{aligned} \tag{5}$$

The following assumptions are made: Histogram binning execution in RMS takes 97% so $B_{init} = 0.03$. The sequential portions of Gaussian and Uniform filtering consume 2% of filtering time so $B_{filt} = 0.02$. By substituting this into equation 5 we have,

$$\begin{aligned} \mathrm{B}_{S+C} &= 0.03 + 0.02 \times 11 = 0.25 \\ \mathrm{S}_{S+C} &= \frac{1}{0.25 + \frac{0.75}{n}} \to 4 \end{aligned} \tag{6}$$

Equation 6 results in a strong scaling factor of 4. We conclude that it is worthwhile to increase the number of processors and maximise parallel portions of S+C code.

Next we consider weak scaling with *Gustafson's Law*. We assume that all parallellisable code executes sequentially on each processor. Then $f = \frac{s}{s+0} = 1$ for any fixed of execution time s. By substituting this into equation 2 we have,

$$\mathrm{S(n)} = \mathrm{n} - 1 \times (n-1) = n - n + 1 = 1 \tag{7}$$

Equation 7 results in a weak scaling factor of 1. In other words if 1 processor processes d bytes in 1 second then by scaling up to n processors we can process $n \times d$ bytes in the same amount of time.

3.2.1 Hardware

Computations were performed using facilities provided by the University of Cape Town's ICTS High Performance Computing team: http://hpc.uct.ac.za

Execution time is evaluated on hardware provided in table 1.

Table 1: Hardware used to obtain results

Home Desktop	
CPU	Quadcore Intel i3-2120 @ 3.30 GHz
RAM	4GB DDR3 1333 MHz
GFX	GeForce GT 430, 1GB DDR3
UCT's HPC Cluster (HEX)	
CPU	16x Xeon E5-2650 @ 2.00GHz
RAM	62GB
GFX	Tesla M2090, 6GB GDDR5

3.2.2 Profiling Smooth-and-Clip

The pipeline is reduced to only perform global RMS and S+C. The initial evaluation of the original Python S+C execution time is performed on HEX (see table 1). Routines are timed with Python's *time.time()* method and SoFiA's default input and kernels (see table 2) are used for filtering.

Table 2: Default kernel sizes for S+C

Default kernel sizes (px)												
	1	2	3	4	5	6	7	8	9	10	11	12
x	0	0	0	0	3	3	3	3	6	6	6	6
y	0	0	0	0	3	3	3	3	6	6	6	6
z	0	3	7	15	0	3	7	15	0	3	7	15

Figure 2 illustrates execution time of the pipeline for a single run of each FITS file. S+C execution appears linear with respect to cube size and global RMS calculation adds nearly 75% onto the overall execution time. The kernel sizes increase every 4th kernel in X and Y to 0, 3 and 6 respectively which explains the stepped effect in Figure 2 (b). Kernel sizes in Z cycle between 0, 3, 7, 15 yet appear constant in 3, 7 and 15 because the Uniform implementation is independent of kernel size.

3.2.3 Data

Three FITS files are used. The first two are synthetically generated, realistic data cubes with a known number of sources. The synthetic FITS files are generated by Ed Elson from the Astronomy department at the University of Cape Town. The smallest is 327MB in size with 360x360x660 pixels containing 32-bit floating point numbers. Similarly, the larger is 2.3GB in size with 900x900x758 pixels. The third is 724MB at 360x360x1464 provided by Paola Serra and used in *Serra, Jurek Floer (2012)*.

3.2.4 RMS

Global RMS is responsible for 99% of total RMS time on the 2.3GB cube and contributes 38.8% to the combined execution time (Figure 6 (a)). RMS during filtering contributes 0.05% towards filtering time and is unaffected by kernel size. We conclude that accelerating RMS is worthwhile given global RMS execution time.

3.2.5 Filtering

Filtering contributes 61.1% of the reduced pipeline's execution time (Figure 6 (a)), with the Gaussian and Uniform filters contributing almost entirely towards filtering. Gaussian filtering, applied to both X and Y axes, increases with kernel size. Uniform filtering, applied to the Z axis, is unaffected by kernel size. In total filtering consumes 327 seconds, with 150 seconds for Gaussian filtering and 174 seconds for Uniform filtering for the 2.3GB file.

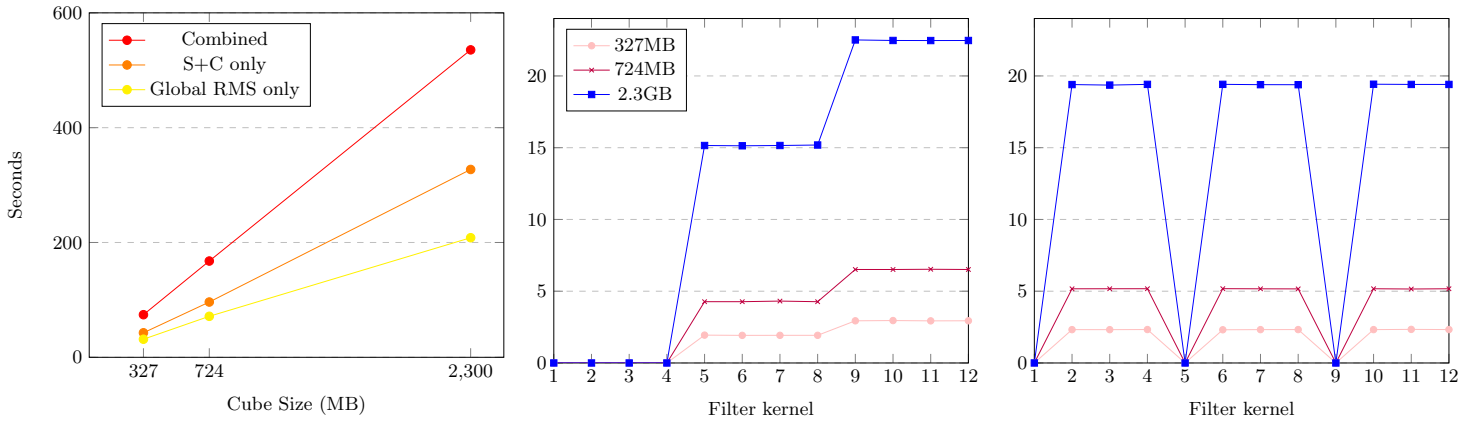


Figure 2: (a) Execution is linear with respect to cube size. Global RMS (yellow) execution time is significant compared to S+C (orange) and together form the reduced pipeline (red). (b) is Gaussian filter speedups on each filter kernel, which increase in X-Y from 0, 3, 6 every 4th kernel. (c) is Uniform filter speedups on each filter kernel, which cycle in Z through 0, 3, 7, 15.

3.2.6 *Ensuring correctness*

RMS, Gaussian and Uniform filters are implemented in isolation, each with their own test cases. Tests compare filtered arrays from the original methods for equality against our implementations. Equality is verified with *assert_array_almost_equal* from the *numpy.testing* module, using double floating point numbers for accuracy. Arrays are generated using *numpy.random.random* to various sizes containing random values between 0 and 1.

RMS is implemented in SoFiA by the GetRMS method found in *functions.py* of the *sofia* module. Gaussian and Uniform filtering is used from scipy's ndimage filtering module *scipy.ndimage.filters.gaussian_filter* and *scipy.ndimage.filters.uniform_filter1d* respectively and we therefore test against these methods for filtering.

3.2.7 *CUDA evaluation*

S+C is evaluated with the APOD development methodology provided by the CUDA Best Practices Handbook [2]. Our primary focus is CUDA acceleration and we do not optimise for the serial-C and OpenMP implementations. Assessments to CUDA implementations are incremental and calculate the speedup against the original python execution time. S+C is implemented by the *SCfinder_mem* method from *pyfind.py* in the *sofia* module. We evaluate *SCfinder_mem*'s execution time as the ultimate acceleration metric achieved for the project.

4. IMPLEMENTATION

The implementation, although focused on acceleration, is designed to be compatible with the SoFiA package. This is done to maximise the likelihood of integrating the changes back into SoFiA. We make no adjustments to the SoFiA pipeline itself, instead we alter the *SCfinder_mem* sub routine by replacing the filtering processes described in *Algorithm* 1 on lines *9-11* with a single call to our own python method *C_SCfinder_mem* which takes the copy of the data cube and a kernel as input. This is a wrapper method which uses CFFI to call to our C implementation and performs the same filtering logic as the replaced lines but instead using CUDA for filtering.

Algorithm 2 illustrates the simplicity of the *C_SCfinder_mem* wrapper function:

Algorithm 2 C_SCfinder_mem wrapper

```
1: procedure C_SCFINDER_MEM(cube, kernel)
2:     ffi = FFI()
3:     cube_ptr = ffi.cast("double *", cube.ctypes.data)
4:     kernel_ptr = ffi.cast("int *", kernel.ctypes.data)
5:     C.SCfinder_mem(
6:     cube_ptr, cube.shape, kernel_ptr
7:     )
8: end procedure
```

The *scipy* filtering methods are highly optimised for speed at the expense of memory with up to 4 times the memory usage of the input data cube [8]. Our implementation utilises the memory on the CUDA enabled device to perform filtering in-place on the host, requiring no additional host memory.

We assume data cube size exceeds memory available on the device. Both the Gaussian and Uniform filters allocate and process the cube in parts that fit into available device memory. Memory strides for the axes depicts how the cube is padded, and differs for the Gaussian and Uniform filters.

4.1 Gaussian filter implementation

Gaussian performs separable convolution along X and Y axes, which have the least stride in memory alignment. Our Gaussian implementation therefore subdivides the cube along the Z-axis, ensuring X-Y planes of data fit onto the device in their entirety. This has three advantages: A simpler kernel which does not need to handle padding within the planes themselves, but only at their borders with zeros; both X and Y filtering is performed per plane with a single copy to the device; and this favors the typical cube structure which are

longer along the spectral Z-axis. However, it is unable to split X-Y planes and cannot process cubes where a single X-Y plane is too large to fit onto the device. We utilise shared memory to minimise latency from memory access since each pixel samples multiple neighbouring pixels. The gaussian weight values are calculated once on the CPU and cached in constant memory on the device for low latency access. We tested a range of CUDA kernels to find an optimal implementation.

4.2 Uniform filter implementation

The Uniform filter performs a 1D convolution along the Z axis, which has the biggest stride in memory alignment. Fortunately we take advantage of a sliding average implementation which amortises the memory accesses required in the initial average calculation and remaining accesses to 1. This implementation starts at the first Z-index and processes pixels along Z until the end pixel is reached. Generally the matrix is transposed beforehand to coalesce pixels along the Z-axis which results in a significant performance improvement. We did not implement this as it requires a duplicate cube and this would break our limited memory constraint and we did not attempt an in-place matrix transposition with the GPU. Data is subdivided into sections by the Z-axis in the same manner as the Gaussian filter except in this case pixel values are required for padding. Sliding averages are calculated at the beginning of each Z-index per section. Each Z-index is assigned to a single thread. A disadvantage of this is the number of averages calculated per Z-index increases by the number of sections. Each pixel is accessed only once so we would not benefit from shared memory and the implementation just uses global memory.

We note that the CUDA implementation crashes abruptly when there is insufficient device memory and it is difficult to predict how this will happen. We could not allocate all available memory without experiencing a crash and as a result shave off 15MB from the reported available memory, but we do not know of an optimal value.

5. RESULTS AND DISCUSSION

We first detail speedups for the Gaussian and Uniform filters and RMS, then list final speedups on S+C execution time. The Gaussian code ran on data cubes with a fixed height of 320px and varied square, planar width. The Uniform code ran on data cubes with a fixed square, planar width of 320px and varying height. The S+C results ran on the 327MB, 724MB and 2.3GB FITS files. CUDA results are discussed in their order of implementation using the APOD approach.

5.1 Gaussian filter

The original CPU version runs in quadratic time with respect to cube size (Figure 3 (a)) and increases with respect to increasing filter size. The spike in Figure 3 (b) is a result of paged memory alignment. Here the data cube with square width of 512 results in memory accesses aligned to the same line in cache. This causes frequent L1 and L2 cache invalidations from paged memory alignments [1]. Our CUDA implementation did not experience the same effect and the result is a dramatically increased execution time on the CPU and comparatively high speedups that obscure results. Cube sizes therefore avoid powers of 2 but are detailed separately.

Our serial-C implementation is much slower than the original scipy version, which is heavily optimised for speed. However, our implementation uses less memory at 2 times the original cube size whereas the SoFiA implementation peaks between 3 and 4 times the cube size [8]. The OpenMP implementation performed up to 1.5x faster than the original scipy version on the quad-core i3 CPU (Figure 3 (c)).

Our naive, unoptimised CUDA implementation on a 430 GT device using Doubles, shared memory and a block size of 1024 threads (32x32) shows constant speedup or around 1.5x on all cube sizes (Figure 4 (a)). The dip in performance for small cubes is likely due to overhead in copying data between device and host. For Doubles, filter sizes of 15 could not be computed on the 430 GT device due to limited memory. A kernel of size 12 is used instead.

The naive implementation is first improved for for memory access and second for compute time, since memory latency is the biggest contributing factor to GPU performance [2]. Due to time constraints we did not attempt low priority optimisations specified in the *CUDA Best Practices Handbook*. The first improvement aligns Y-axis pixels in shared memory for processing on the Y-axis. The improvement (Figure 4 (b)) is minimal and constant, with larger gains achieved by larger filters due to the increased coherent accesses.

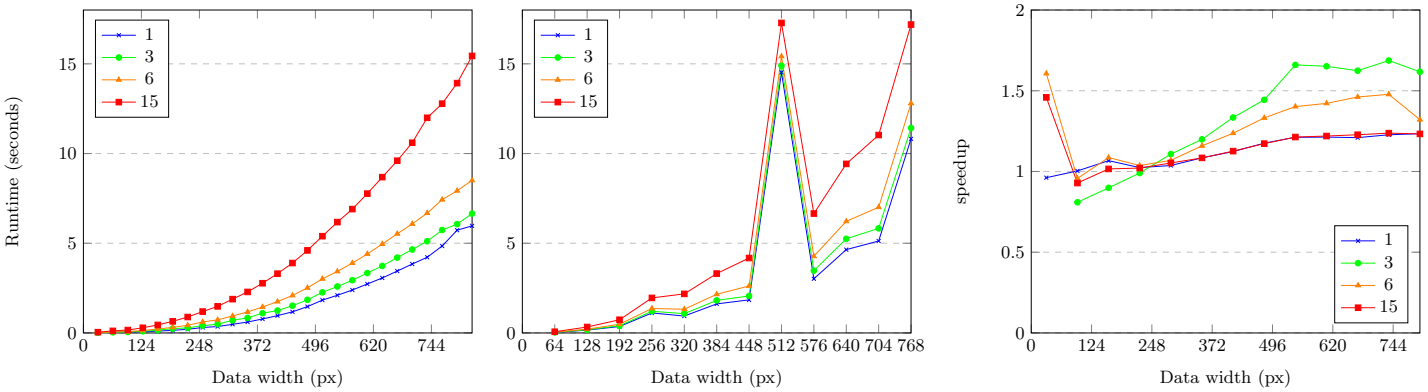
The implementation is then optimised to use Floats instead of Doubles. This affects both memory access and compute time as the memory footprint is halved, allowing twice the number of pixels to be processed per interval by the GPU. The GPU has more floating point arithmetic units than double floating point arithmetic units and the extra usage results in higher occupancy. The improvement is more pronounced at larger cube sizes (Figure 4 (c)) that exceed device memory and benefit from half the number of data transfers, but is difficult to see at the given scale.

Next we optimise for compute time. The Nvidia Graphical Profiler reported that reducing the block size from 1024 to 256 threads increases the number of concurrently executing warps from 32 to 48, which increased the speedup from 2.5x to 3.5x (Figure 4 (d)).

The graphical profiler also reported under utilisation of shared memory, which can increase from 4KB to around 8KB and maintain 48 concurrent warps. The number of pixels loaded into shared memory and convolved per thread is increased to 4. This amortises the cost of calculating pixel indexes and reduces the proportion of padding in shared memory, reducing redundant loads from global memory and increasing occupancy to nearly 100%. However, increases in occupancy values above 50% do not necessarily improve performance [2].

The spike at 512px on Figure 4 (f) is due to frequent cache invalidation caused by memory alignments. No optimisations were implemented for (f) and the speedups are purely the result of cube widths a multiple of 64.

Asynchronous memory copies and kernel calls execute concurrently on devices with multiple compute and copy engines. The 430 GT device has 1 copy and compute engine and did not increase in speedup. The visual profiler reported that kernels begin executing nicely after the first copy task completes but do not begin to copy back to the host until the last kernel has completed. Increasing the number of asynchronous memory transfers and kernel calls to 8 on the Tesla M2090 device (Figure 4 (g)) utilises its 2 copy engines to hide memory latency even further and improve speedups



(a) Original execution (b) Powers of 2 data size (c) OMP Speedup

Figure 3: The execution time of the Gaussian filter on data cubes with square, increasing width and fixed height of 320px. Each filter size is represented by a colour. (a) is the original, sequential execution time while (b) is the same except on powers of 2 data cubes. (c) is the speedup obtained with OpenMP on (a) using the quad-core i3.

to 20x on larger filters.

The last two optimisations use the improved hardware on the Tesla M2090. The blocksize is increased from 256 to 1024 threads (Figure 4 (h)) which achieves a slight performance gain over (g) that is more pronounced on larger cube sizes. Finally, the number of consecutive pixels processed per thread is increased from 4 to 10 (Figure 4 (i)). This is the maximum we could achieve and may not be optimal, only improving the 15px filter.

With large kernel sizes, this best implementation of the Gaussian kernel achieves a maximum of 23x speedup.

5.2 Uniform Filter

Execution of the Uniform filter is linear with respect to cube size and independent of kernel size (Figure 5 (a)). This is due to the sliding window implementation, which amortises memory accesses to 1 per pixel.

The OpenMP implementation performed poorly, with a speedup of about 1.1 on the desktop CPU for every kernel size except 1. The kernel with size 1 yielded a constant speedup of 20 for Doubles and 40 for Floats. We attribute the otherwise low speedup to large strides in memory access along the Z-axis.

Our initial naive CUDA kernel uses Doubles and achieves a relatively constant speedup of around 6 on the 430 GT device (Figure 5 (c)). This implementation reaches a performance plateau at a cube height of 320px, which is likely the point at which global device memory becomes fully allocated and occupancy is maximised.

As before, we first optimise for memory accesses and second for compute time. Using Floats instead of Doubles halves cube size and memory strides. The adjustment is trivial and doubles the speedup to around 12 (Figure 5 (d)). Smaller filter sizes have slightly larger speedups than larger filter sizes, which is likely due to fewer strided memory accesses. The opposite effect is seen in Gaussian filter sizes.

Speedups on cube sizes which are powers of 2 increase linearly (Figure 5 (e)). This is similar to (d) but not as obvious given the cube range. We do not experience spikes in performance as we do with the Gaussian filter but instead experience overall increased speedup of up to 20x (Figure 4 (f)). This is achieved with square widths of 256 pixels and varying height by increments of 64.

The optimal Uniform filter kernel achieved similar speedups on the desktop machine and HEX, with a maximum speedup of 13.78x on data cube sizes which are not a power of 2.

5.3 RMS

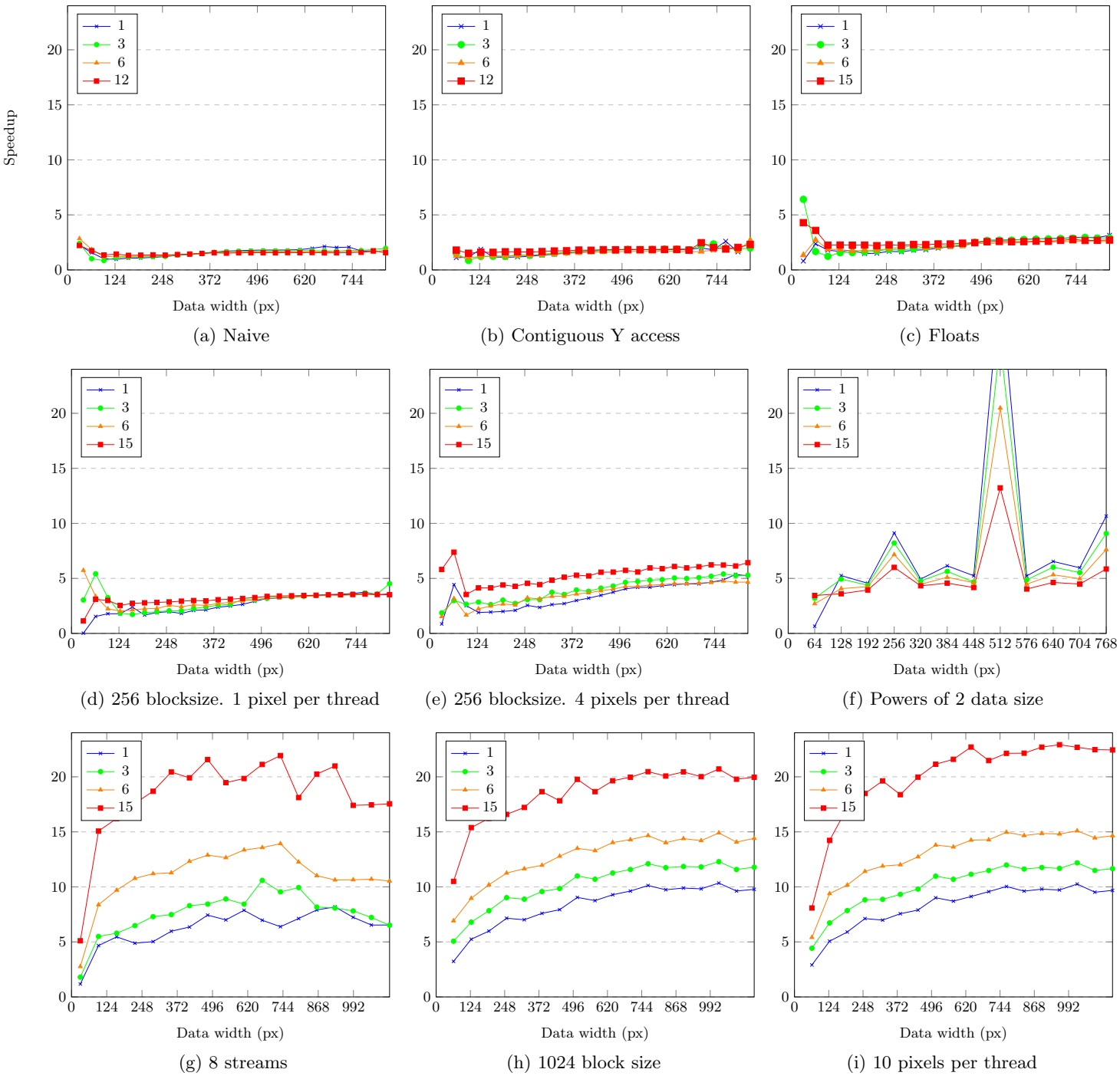
As of v0.5.0 of SoFiA the RMS method has been improved and our efforts to accelerate it are no longer appropriate. However, we briefly report on the results achieved for this method. We found that 94% of the total RMS execution time is in the histogram binning process and a naive serial-C implementation was written to replace it, which reduced the execution time from 208 to 32 seconds on the 2.3GB cube, a 6.4x speedup. Further reductions are expected using OpenMP but we did not explore this as filtering remained the primary bottleneck.

5.4 S+C speedup

S+C results are achieved after integrating the Gaussian and Uniform CUDA kernels and the serial-C histogram method. We ran S+C on the three FITS files and the results (Figure 6) show a linear relationship between execution time and cube size.

Speedups achieved by each filter kernel is illustrated in Figure 6 (b) and (c). Kernel sizes only increase in X and Y every 4th kernel which results in the stepped behaviour in (b). However, kernel sizes in Z cycle between 0, 3, 6 and 15 which results in the jittered behaviour clearly apparent in (c). Gaussian filtering execution time reduced from 150 seconds down to 16, a speedup of 9x. Uniform filtering reduced from 174 seconds to 20, a speedup of 8.3x. Local RMS as mentioned previously reduced from 1.89 to 0.225 seconds, a speedup of 8.4x.

Overall execution time is reduced by a factor of 7.6x from 535 seconds to 70 on a 2.3GB data cube. The original implementation has constant throughput of 4.3GB/s on all three data cubes, while throughput for the accelerated implementation is 16GB/s, 23GB/s and 32GB/s for the 327MB, 724MB and 2.3GB files respectively.

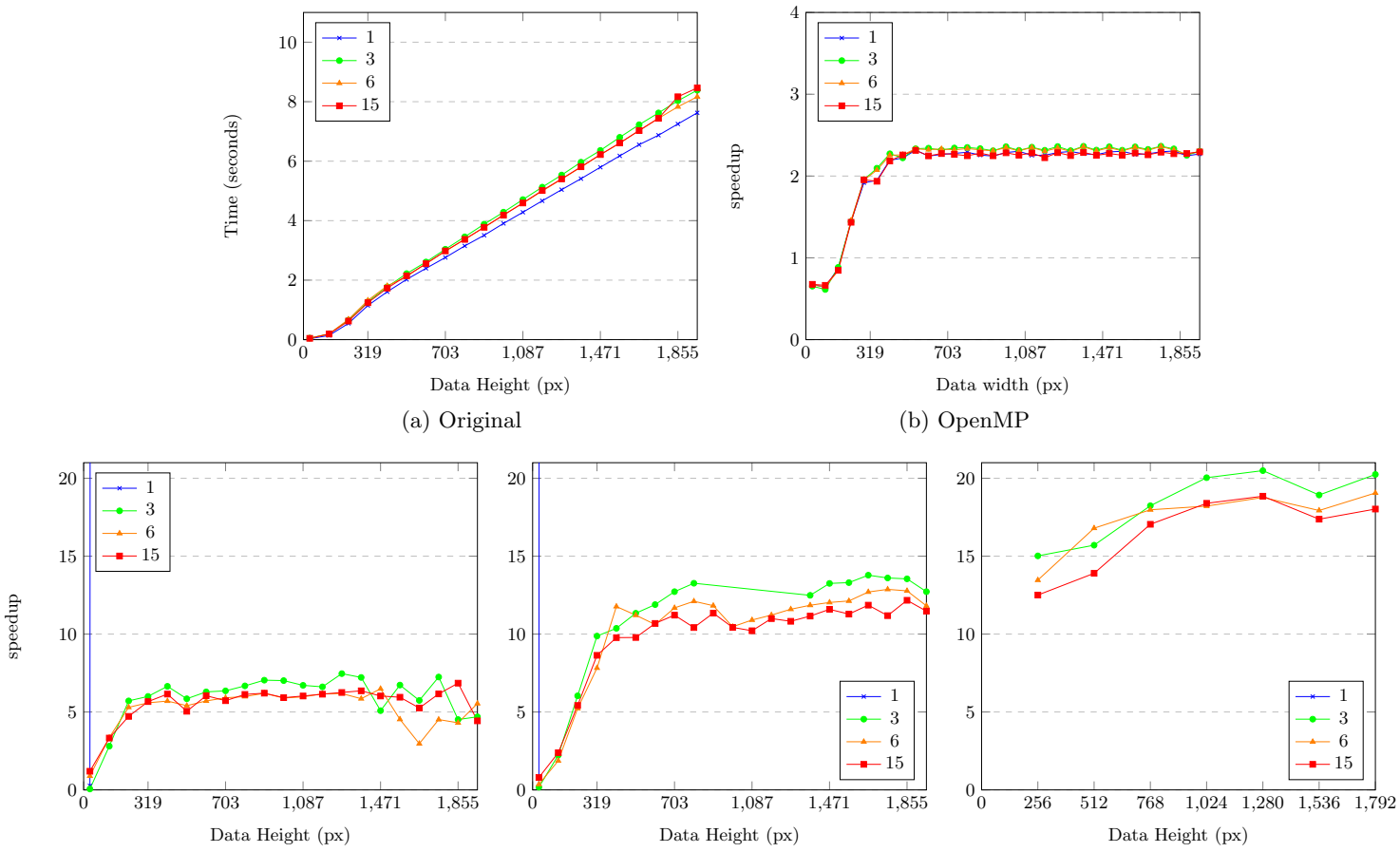


(a) Naive (b) Contiguous Y access (c) Floats

(d) 256 blocksize. 1 pixel per thread (e) 256 blocksize. 4 pixels per thread (f) Powers of 2 data size

(g) 8 streams (h) 1024 block size (i) 10 pixels per thread

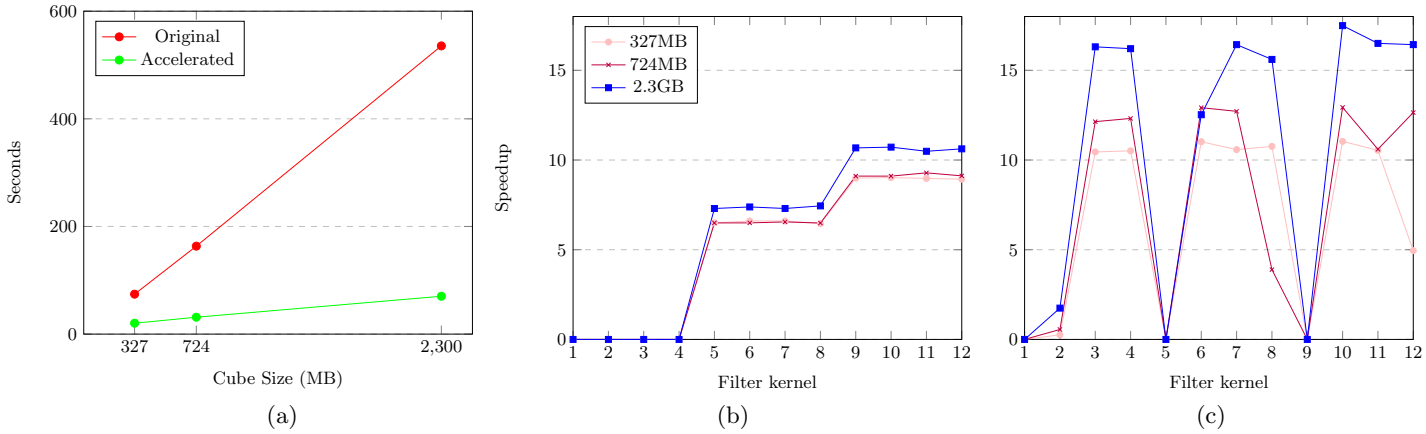
Figure 4: Incremental speedups obtained from the Gaussian filter using the APOD approach. Diagrams (a) - (f) are performed on the 430 GT while (g) - (i) are performed on the Tesla M2090. (a) Naive implementation using Doubles with no optimisations, we observe speedups similar to OpenMP. (b) First optimisation implementing contiguous shared memory accesses along Y for Y-axis filtering. The improvement benefits larger kernels and cube sizes with their increased sampling. (c) Second optimisation, Doubles are replaced by Floats. Memory use is halved, reducing the number of transfers onto the GPU and increasing occupancy with added utilisation of floating point arithmetic units. (d) Third optimisation, the block size is reduced from 1024 threads to 256. This increases the number of concurrent warps from 32 to 48. (e) Fourth optimisation, filtering 4 consecutive pixels per thread instead of 1 amortizes the index calculation time and reduces the proportion of padding in shared memory, almost quadrupling shared memory use. (f) No optimisation, speedup with power of 2 cube sizes. Large spikes such as at 512px are caused by degraded CPU performance from frequent cache invalidations which is not a factor on the GPU. (g) Fifth optimisation, run on the Tesla M2090 instead of the 430 GT in order to utilise streaming with 2 copy engines. Speedup is a result of both this optimisation and the stronger GPU. (h) Sixth optimisation, block size is increased from 256 threads back to 1024. Larger cube sizes benefit the most. (i) Seventh optimisation, threads each filter 10 consecutive pixels, the maximum we could achieve, instead of 4.



(a) Original (b) OpenMP

(c) Doubles (d) Floats (e) Powers of 2

Figure 5: (a) Original execution time which is linear with respect to cube size and independent of kernel size, as expected from the sliding average implementation. (b) OpenMP speedup on the quad-core home desktop is constant at just around 2.2. The low speedup can be attributed to the slow serial-C execution time and strided memory access along Z, while the constant behaviour can be attributed to on average 1 memory access per pixel. (a) is the naive CUDA implementation using Doubles which achieves a constant speedup of about 6. (b) optimises for memory access by using Floats instead of Doubles which halves memory use, stride and the number of *cudaMemCpy* and kernel calls, achieving twice the speedup of (b). (c) is the speedup achieved with cube sizes a power of 2. Cubes contain square widths of 256 pixels with heights that are multiples of 256.



(a) (b) (c)

Figure 6: Execution of the reduced S+C pipeline. (a) shows the linear relationship between execution time and cube size for the original (red) and accelerated (green) implementation. (b) is the Gaussian filter speedups achieved with the default filter kernels (table 2). The stepped behaviour is due to the filter sizes increasing in X and Y every 4th filter kernel with values 0, 3, 6. (c) is the Uniform filter speedups achieved with the default filter kernels (table 2). The jitter-ed behaviour is due to the Z filter sizes cycling between 0, 3, 7, 15.

6. CONCLUSIONS

We have accelerated the S+C implementation in the SoFiA source finding package as follows: The filtering logic using scipy's Gaussian and Uniform filters (Algorithm 1 Lines 9-11) were replaced with a C method executing similar Gaussian and Uniform CUDA kernels. GPU device memory is used as a convenient buffer to avoid redundant memory use on the host. Gaussian filtering is performed on two axes, X and Y, while Uniform filtering is performed on Z. In addition the Uniform filter implementation performs fewer memory accesses independent of filter radius.

Despite this the original Uniform filter took longer on the host, at 174 seconds, with the Gaussian filter taking 150 seconds on a 2.3GB data cube. Similarly, the Uniform CUDA implementation ran slower at 20 seconds as opposed to 16 seconds for the Gaussian CUDA implementation. The poorer performance is a result of the Z axis having the largest stride in memory access but may benefit from in-place matrix transformation to coalesce Z-axis pixels in memory. Progressive, incremental improvements in the Gaussian kernel allowed us to achieve speedups of over 20x for larger kernel sizes.

The GPU is highly beneficial for data cubes whose size is a power of 2. This is a result of degraded performance on the CPU due to frequent cache in-validations from aligned memory accesses. The GPU does not experience this, and a data cube of 512x512x256 achieved a 26x speedup with a filter radius of 3px, while similar cube sizes only achieved a 5x speedup (Figure 4 (f)). Our implementation uses less memory at 2 times the original cube size whereas the SoFiA implementation peaks between 3 and 4 times the cube size [8].

Overall, S+C acceleration increased with cube size from 2.88x on a 327MB data cube to 8.6x on a 2.3GB data cube. Execution time reduced from 327 seconds to 39 seconds using 2.3GB data. Combined with global RMS total execution time was reduced from 535 seconds to 70 seconds (Figure 6 (a)), a 7.6x speedup.

Results show CUDA is favorable over the CPU for filtering processes in source finding algorithms such as S+C in SoFiA. However, S+C is a single source finding algorithm and individual effort is required per algorithm to accelerate entire general purpose source finding pipelines such as SoFiA, as is also the case with the SSoFF framework.

7. FUTURE WORK

The CUDA Best Practices Handbook lists several libraries which are optimised for CUDA processing such as cuBLAS and cuFFT. These could be used to mitigate the manual effort required to implement CUDA kernels in general pipelines such as SoFiA. Additionally, pyCUDA could be tested instead of C or C++ CUDA implementations, as this would avoid the complexity of maintaining C/C++ code in addition to Python.

S+C supports both Uniform and Gaussian filtering along the Z-axis (spectral axis). Our implementation only supports Uniform filtering along Z, and work needs to be done to modify the Gaussian filter to the filter Z in addition to X and Y.

The Gaussian kernel requires that entire X-Y planes of data are copied onto the device. This limits the maximum width of the data cube that can be processed to the available memory on the device. In order to process these larger cubes they would have to be subdivided on the host or elsewhere before being processed. The kernel can be adjusted with some effort to process subsections of X-Y planes and accommodate the necessary padding this incurs. However, this may not be necessary as data cubes are longer along the spectral Z axis and not X-Y.

The code and additional dependency on CFFI can be integrated back into the SoFiA package. This project uses v0.4.0 of SoFiA but at the time of writing v0.5.0 has been released and further versions are under development. As of v0.5.0 the SoFiA developers have been focusing on C++ re-factors to integrate OpenMP into the pipeline and our CUDA efforts should not conflict with theirs. OpenMP refactoring on their part will conveniently clear a pathway for CUDA integration, which is yet to be undertaken.

8. REFERENCES

[1] S. J. Badenhorst. Acceleration of the noise suppression component of the duchamp source-finder. Master's thesis, University of Cape Town, 2014.

[2] C. Cuda. Best practices guide. *Nvidia Corporation*, 2012.

[3] B. W. Holwerda and S.-L. Blyth. Trumpeting the vuvuzela: Ultradeep hi observations with meerkat. *arXiv preprint arXiv:1007.4101*, 2010.

[4] J. L. Jonas. MeerkatâĂŤthe south african array with composite dishes and wide-band single pixel feeds. *Proceedings of the IEEE*, 97(8):1522–1530, 2009.

[5] R. Jurek. The characterised noise hi source finder: Detecting hi galaxies using a novel implementation of matched filtering. *Publications of the Astronomical Society of Australia*, 29(3):251–261, 2012.

[6] K. Karimi, N. G. Dickson, and F. Hamze. A performance comparison of cuda and opencl. *arXiv preprint arXiv:1005.2581*, 2010.

[7] A. Popping, R. Jurek, T. Westmeier, P. Serra, L. Flöer, M. Meyer, and B. Koribalski. Comparison of potential askap hi survey source finders. *Publications of the Astronomical Society of Australia*, 29(03):318–339, 2012.

[8] P. Serra, T. Westmeier, N. Giese, R. Jurek, L. Flöer, A. Popping, B. Winkel, T. van der Hulst, M. Meyer, B. S. Koribalski, et al. Sofia: a flexible source finder for 3d spectral line data. *arXiv preprint arXiv:1501.03906*, 2015.

[9] I.-J. Sung, J. Gómez-Luna, J. M. González-Linares, N. Guil, and W.-M. W. Hwu. In-place transposition of rectangular matrices on accelerators. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–218. ACM, 2014.

[10] S. Westerlund and C. Harris. A framework for hi spectral source finding using distributed-memory supercomputing. *Publications of the Astronomical Society of Australia*, 31:e023, 2014.

[11] R. Wu, B. Zhang, and M. Hsu. Clustering billions of data points using gpus. In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 1–6. ACM, 2009.