Accelerating the Noise Removal Process for Astronomical Source Finding

Yaseen Hamdulay
University of Cape Town
yaseen@hamdulay.co.za

ABSTRACT

Next generation radio telescopes, such as the Meer Karoo Array Telescope, will perform the largest surveys of the sky ever completed. The results of which will allow us to gain a better understanding of how galaxies form. Source finders process the data from these surveys in order to identify previously unknown galaxies. Our current source finders are too slow and will be unable to cope with the size of upcoming surveys from next generation telescopes. We use General Purpose Graphics Processor Units to accelerate the source finding process in DUCHAMP in order to handle larger surveys. This resulted in a 11 times speedup proving the viability of this technique.

Keywords

Astronomy, GPU, Signal Processing, Source Finding

1. INTRODUCTION

Radio astronomy is the study of the universe in the radio spectrum of light. Identifying galaxies from background noise is a crucial process in radio astronomy. This process is called source finding and can improve our understanding of galaxy evolution over time [7]. Source finding is done by astronomers by hand. This is a tedious and slow process that has been augmented by automated source finders.

Background noise emitted by man-made objects such as cellphones and satellites as well as natural sources like the cosmic microwave background radiation dominate observations in the radio spectrum. Galaxies sparsely populate the universe making them difficult to differentiate from background noise. This noise needs to be removed from observations before attempting to identify sources. All automated source finders perform a noise removal step.

These galactic surveys produce three-dimensional outputs called data cubes that are typically stored in Flexible Image Transport System (FITS) files. FITS was designed by the International Astronomy Union for storing and transferring astronomy image data.

Modern radio telescopes such as the Karoo Array Telescope (KAT) are more detailed and cover a larger volume of the sky than any existing telescope. Existing automated source finders are currently unable to process such large observations. They either run too slowly or limit the size of input observations to the size of memory. Source finders need to be improved for these use cases.

DUCHAMP is an automated source finder created at the Australia National Telescope Facility that suffers from the these problems.

A general technique to decrease the execution time of algorithms is by splitting the problem into several parts and executing a few at a time simultaneously. Modern desktop CPUs can run up to eight threads of execution in parallel while graphics processing units (GPUs) can execute up to several thousand simultaneously.

We propose using the computational power of GPUs to speed up DUCHAMP.

CUDA is a proprietary API that exposes general purpose computing on NVidia GPUs. GPUs are massively parallel and provide high levels of arithmetic throughput in comparison to the CPU. The high performance of a GPU comes at the cost of high programming complexity [5].

OpenMP is a multi-platform API to write multi-threaded CPU code in the C family of languages [4]. It manages individual threads internally and exports a simple API for loop-level concurrency. The pthreads API is more common, powerful and gives the user more control but with a higher level of programming complexity.

In this paper we present performance improvements to the DUCHAMP source finder. We investigate performance bottlenecks in the source finder by running a profiler.

We first investigate bottlenecks in the source finder, find algorithmic improvements and then use parallelism to speed it up.

2. BACKGROUND

Observations from radio telescopes have two spatial and one frequency spectrum dimension.

We define the recession velocity of a galaxy as the velocity at which it is moving away from earth. The emission spectra of a galaxy is the range of frequencies emitted from it and their corresponding intensities. When an object moves away at high speeds its light gets "red shifted" which means that its frequency decreases or equivalently that its wavelength increases. We are able to calculate a galaxies recession velocity as a function of the difference between the emitted frequency and the frequency of the emission at rest.

$$v \approx c \cdot \frac{f_{emit} - f_{observed}}{f_{observed}}$$

Where v is the recessional velocity, c is the speed of light, f_{emit} is the frequency of light emitted and $f_{observed}$ is the frequency of light that has been observed. Due to the expansion of the universe objects that are further away have a greater recessional velocity than those that are closer by a constant multiple.

$$v = H_0 \cdot d$$

Where H_0 is Hubble's constant and d is the distance from earth.

$$d \approx \frac{c}{H_0} \cdot \frac{f_{emit}}{f_{observed}} - \frac{c}{H_0}$$

Rearranging this we find that d is a linear function of the observed frequency. The spectral dimension is then effectively a proxy for distance. We are then able to treat the spectral dimension as another spatial dimension.

Completeness and reliability are metrics that are commonly used to compare the performance of different automated source finders [10]. Completeness is defined as the ratio of sources found by a source finder in a given data cube to the number of sources that exist within the bounds of the data cube [10]. High completeness implies that a source finder would detect most of the sources that actually exist within a data cube. It is possible for a source finder to mistakenly identify what is in reality noise as a source. Reliability is the ratio of true positive source detections to the number of total detections [10]. Low reliability implies that most of the sources detected do not actually exist.

A source finder has to make a trade-off between its completeness and reliability. Consider the noise removal threshold. As it decreases what was previously considered noise is now signal and potentially detected as a source, so reliability decreases. Sources that were previously considered noise are now detected and completeness rises. The reverse occurs if we increase the noise removal threshold.

A source finders completeness and reliability is measured using a data cube where all sources contained within it are known. The results of the source finder are compared to the known results and the completeness and reliability are calculated. A data cube with known sources can be created by manually counting the true sources of an existing data cube or by generating a synthetic data cube.

Not all algorithms or programs parallelise well. There are algorithms that achieve a small speedup compared to the number of threads it is run on. This may be due to many factors such as data dependencies or lock contention. Amdahl's law provides an upper bound on the speedup we are able to achieve when only some subsections of a program are able to be parallelised [6]. Let $P \in [0, 1]$ be the portion of the program that can be parallelised. $1 - P$ is then the portion that cannot be parallelised. Amdahl's law say that the maximum speedup that can be achieved with N processors is:

$$S(N) = \frac{1}{(1 - P) + \frac{P}{N}}$$

This tells us that the speedup will be at most the number of processors but probably smaller.

An analysis of DUCHAMP found that 65-95% of the source finding process is spent in the noise removal phase [1]. We focus our acceleration attemp on the 3D A' Trous wavelet reconstruction algorithm for noise reduction within DUCHAMP. It takes a data cube, an intensity threshold and a filter kernel as input and outputs a noise reduced data cube. The intensity threshold is chosen as some multiple of the median average deviation from the median (MADFM) which is approximately equal to a constant multiple of the standard deviation.

The A' Trous algorithm is described in Algorithm 1. DUCHAMP runs A' Trous many times until the average noise level is below a user defined limit.

Algorithm 1 Removing noise from a Data cube

```
Precondition: input is the noisy data cube with dimension (x, y, z)
Precondition: filter is the wavelet we reduce by
 1: function A' TROUS(input)
 2:     initial median ← median(input)
 3:     output ← zero matrix with dimension (x, y, z)
 4:     coefficients ← output - coefficients
 5:     for $scale = 1$ do scale< $\log_2(\min(x, y, z))$
 6:         threshold ← median(wavelets) ·initial median · sigmaFactor[i]
 7:         wavelet ← coefficients - (filter*coefficients)    ▷ Where * is the convolution operator
 8:         coefficients ← coefficients - wavelet thresholdOutput(output, wavelet, threshold)
 9:         filter ← double filter size
10:     end for
11:     output ← output + coefficients
12: end function
Precondition: output data cube
Precondition: input data cube to be thresholded
Precondition: threshold real number
    function THRESHOLDOUTPUT(output, input, threshold)
 2:     for i=0 doi<dimension of input data cube
            if $abs(input[i]) \geq threshold$ then
 4:             $output[i] \leftarrow output[i] + input[i]$
            end if
 6:     end for
    end function
```

The Graphics Processor Unit was developed for gaming due to the limitations of CPU's for high throughput graphics processing. It has since been generalised to perform general purpose high throughput computation on what is now known as a General Purpose Graphics Processing Unit (GPGPU). The GPU falls into the Single Instruction Multiple Data (SIMD) category of parallel computing. Algorithms that perform the same operation on many data points parallelise well on SIMD hardware. GPUs have thousands more floating-point units and memory bandwidth than a high-end CPU. For example the Tesla K80 has a maximum memory bandwidth of 480 GB/sec while the latest Intel Haswell CPU architecture has a maximum reported bandwidth of 102 GB/sec. It can perform these tasks orders of magnitude faster and at lower power than a CPU. Unfortunately this is only possible for suitable algorithms.

NVidia released the Compute Unified Device Architecture (CUDA) API in 2007 to simplify general purpose computing on their GPUs. Prior to the release of CUDA, graphics programming shader APIs were repurposed into performing general computing. Despite CUDA's simplifying approach to GPGPU programming it is still fraught with complexity. Synchronization between the GPUs hundreds of cores, moving data through the memory hierarchy and chunking complicate CUDA coding.

GPGPU code has the potential to run faster due to its high level of parallelism. It achieves this by having hundreds to thousands of threads running concurrently. A block is a group of threads run concurrently by a single streaming multiprocessor of which there can be many within the GPU.

CUDA enabled GPUs have several layers of memory that trade off size with memory access latency. The type of memory used by a CUDA program greatly influences its performance. Global memory is the slowest and largest section of memory. It is the only memory that persists after the GPU program has completed and has to be used to store input and the results of computation. Shared memory is accessible between all threads within a block and is many times faster than global memory as it is on the streaming multiprocessor. Registers are the fastest and most limited piece of memory but are not random access and can therefore not be used to store arrays of data.

When optimising GPU code we try to move as much of the memory accesses onto registers and shared memory. The number of threads per block and the number of blocks run concurrently on a single GPU is limited by the amount of resources the GPU has available. Decreasing the number of registers or shared memory used by a thread allows more to be run concurrently. Occupancy is the ratio of active threads to maximum number of threads the GPU supports. Maximising occupancy is an important optimization step as higher occupancy generally means better performance. Occupancy is affected by block size, register count and shared memory usage.

Westerlund et al found that using GPUs sped up their source finder by 3.2x over their multi-core multi-threaded source finder [12].

Noise on the radio spectrum is of high-frequency. Removing the high frequency parts of the data cube removes most of the noise at the expense of some detail. We do this by transforming our data cube from the time domain into the frequency domain, removing the high frequency elements and reconstructing back into the time domain. This is known as a high-pass filter in electronics and image processing. The wavelet transform is an instance of a time to frequency domain transformation. Fourier transforms are not suited to noise removal on data cubes as removing or reducing coefficients leaves artifacts or ripples throughout the data cube. Wavelet transforms do not have this problem as each wavelet has a local effect.

The A' Trous algorithm performs a discrete wavelet transform on the input signal, in this case a data cube.

A common way to apply a linear filter to a signal is to convolve the signal with the filter. Convolution, commonly denoted as $*$, is defined as

$$(f * g)(t) \overset{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$$

where f, g are continuous functions. Since our observations are broken into discrete voxels, the three-dimensional equivalent of a pixel with approximate floating-point intensities we use discrete convolution.

$$(f * g)[n] \overset{\text{def}}{=} \sum_{m=-\infty}^{\infty} f[m]\, g[n-m]$$

Let f be a filter (represented as a matrix) and g the signal (in this case our data cube) that we are filtering. A filter, f, is called separable if it can be broken into parts f_1, f_2 such that $f = f_1 * f_2$. Applying the filter to the signal we have

$$g * f = g * (f_1 * f_2)$$

By the associativity property of convolution we can rewrite it as

$$g * f = (g * f_1) * f_2$$

That is to say that convolving the signal with the filter is the same as convolving it with its separable parts in order.

There is a computational advantage to convolving the separable parts instead of the entire filter. Let the dimension of f be

$$|f| = P \times Q, |f_1| = P, |f_2| = Q, |g| = M \times N$$

where f and g are represented by matrices. The number of operations required to convolve $g * f$ is $\Theta(PQMN)$. However, if we instead convolve the separable parts individually we would first convolve $g * f_1 = p$ which takes $\Theta(PQM)$ operations. And then subsequently convolve this partial result p with $p * f_2$ which takes $\Theta(PQN)$ operations. In total we take $PQM + PQN = PQ(M + N)$ operations and $PQ(M + N) \lll PQMN$ operations.

Extending this to three dimensions we can see that the speedup of an $N \times N \times N$ filter would be $N^3/(N + N + N)$. For a filter of width five, we have a theoretical $125/15 = 8$ times speedup.

Duchamp is a three dimensional source finder written by Matthew Whiting at the Australia Telescope National Facility. The Duchamp source finding strategy is the most reliable and complete of all source finding strategies that were tested by Popping et al [10]. This makes it an ideal candidate for acceleration, as it would be the most useful for use by future large hydrogen surveys.

Duchamp currently runs in a single thread on a single CPU core. There have been multiple attempts to run parts of the Duchamp pipeline over multiple CPU cores [1]. Badenhorst et al successfully sped up the A' Trous noise removal algorithm, which was the greatest contributor to the execution time, by 13x with eight threads on a quad-core CPU. This speed up came with a 6x memory usage penalty. They found that with the speed up in the noise removal the execution time is now dominated by the statistics section of the pipeline. Finally, they note that GPU acceleration has the potential to dramatically increase performance.

Selavy [13] is a distributed version of Duchamp that runs across multiple CPU cores across multiple hosts using the Messaging Passing Interface to communicate between hosts. It too accelerated the noise removal algorithm with a small loss in precision due to using approximate statistics methods and converting from computation with doubles to floats. It was designed to run on a cluster of nodes each with several CPU cores where the entire data cube is unable to fit onto a single node.

Duchamp has a configurable pipeline (see Figure 1), we provide an overview of the important parts below.

1. **Noise Removal** smooths the data cube via convolution with a kernel or wavelet reconstruction.

2. **Searching** considers the plane formed by fixing each channel (the spectral or frequency dimension of the data cube) individually and runs the two-dimensional Lutz algorithm over this plane [9]. The Lutz algorithm scans each horizontal row and merges objects within

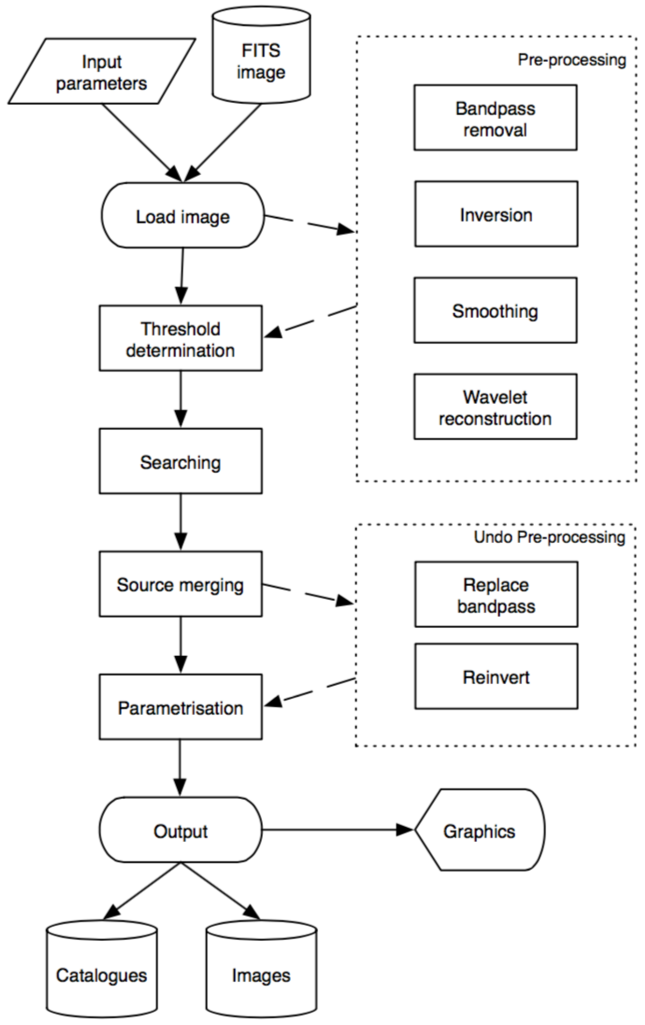


Figure 1: The Duchamp source finder configurable pipeline. Dashed borders contain optional steps.

a small distance threshold. Each objects position and flux values are calculated and added to the list of detected sources [14].

3. **Merging** joins sources that are within a small distance from each other in all three dimensions.

4. **Parameterisation** calculates important astronomical values such as integrated flux, ascension and declination for each source.

The majority of existing source finders use intensity thresholding. An intensity thresholding source finder identifies a voxel, the smallest subdivision of a data cube, as part of a source if it exceeds a preset and often user-defined threshold. Sources that have an intensity close to the noise level are often overlooked by this type of source finder. Since the observable energy for a given telescope surface is fixed as we increase the telescope resolution the observed energy for a single voxel decreases and so does the ability to identify sources. This could cause large or distant sources to be missed. The Characterised Noise H_1 (CNHI) source finder attempts to remove this limitation with its innovative conceptual framework [8].

CNHI makes the assumption that data cubes are dominated by noise and sources span at least three voxels. CNHI applies the Kuiper statistical test to distinguish a test region from noise. The Kuiper test is used to tell whether a set of samples come from a certain distribution. CNHI selects all possible regions as a test region and applies the Kuiper test against the remaining region which is assumed to be noise [11]. If the Kuiper test passes that means the test region is distributed the same as noise and is therefore noise. If it fails it is marked as a source.

After source regions are identified adjacent regions are merged using the Lutz one-pass algorithm. Sources that are smaller than a given number of voxels are removed.

Despite CNHI's innovative conceptual framework the performance of CHNI is poor across all resolutions. CNHI has many false positives, low reliability and low completeness [10]. It was therefore not considered for acceleration.

3. DESIGN

3.1 Aims

We aim to accelerate DUCHAMP's noise removal with algorithmic improvements and parallelism to reduce the execution time of the source finder with large data volumes expected from next generation radio telescopes. We aim for a simple design that is simple to reintegrate into DUCHAMP.

3.2 Approach

DUCHAMP's current implementation is single threaded and uses an inefficient version of A' Trous reconstruction.

We accelerate the A' Trous wavelet reconstruction noise removal algorithm with algorithmic improvements and parallelism. CUDA and OpenMP will be used to parallelise our implementations on the GPU and CPU respectively.

All improvements will be performed on our own code base separate from DUCHAMP to simplify testing and development. We implement only the code related to reading and writing FITS files and noise reduction.

Our accelerated version of the A' Trous reconstruction is implemented in several phases with a working implementation at the end of every phase. This reduces the risk of failure due to high levels of complexity. We verified the correctness of each implementation at the end of each phase.

3.3 Constraints

The proprietary CUDA API requires us to use NVidia GPUs in order to run the accelerated DUCHAMP pipeline. We will not distribute execution over multiple hosts or run data cubes bigger RAM available on a single host due to time constraints. DUCHAMP and CUDA are written in C++ and our implementation will be written in C++ as well.

Any optimisations that reduce the accuracy of our results will not be attempted to conform to the output of DUCHAMP. Maintaining backwards compatibility is important to allow our changes to be integrated to DUCHAMP in the future.

3.4 Evaluation

3.4.1 Validation

We will evaluate the correctness of our implementations using synthetic data cubes created by Dr Ed Nelson of the UCT Astrophysics department. Testing with synthetic data cubes is advantageous due to the fact that exact source information is available and can be used to accurately measure completeness and reliability. Correctness is simply whether the accelerated algorithm produces equivalent outputs to the original DUCHAMP implementation. A small error is allowed for each value in the data cube due to floating-point approximations.

To check correctness we run both the original and accelerated implementations with the same inputs and log the outputs of each. The outputs are then compared for floating-point equality and the floating-point error between the two implementations is logged.

3.4.2 Execution time

On completion of each phase the execution time of our accelerated implementation is measured.

The size of the input data cube and the filter size are the only factors that affect the execution time.

To determine the relationship between the size of the input data cube and execution time we test with data cubes of various sizes. We create data cubes of various sizes by taking subsections of an existing large data cube. Each implementation is run on all test data cubes three times and the results are averaged over all three executions. This is done to minimize the affect of background processes on performance.

Our measured execution time is compared to the original DUCHAMP implementation and we measure the speedup.

We benchmarked the result of each phase with data cubes of various sizes. The benchmarks were run on a desktop with an Intel i7-4790 clocked at 3.6GHz, 8GB of RAM and a NVidia GTX 970 graphics card as well as the Hex computing cluster run by ICTS. The Hex cluster has ten NVidia Tesla M2090 GPU's that are designed for general purpose computing and a CPU compute cluster with 18 hosts each with 64 cores and 128 GB RAM. We use the CPU compute cluster for testing our OpenMP implementation. We ran the OpenMP accelerated implementation on a varying number of cores to determine its relationship with execution time.

3.5 Software Development Methodology

An Agile methodology was used over the course of this project. Our tasks are divided into small phases. Each sprint completes at least one phase including running all correctness checking. The agile methodology is appropriate for this project as it allows us to ensure a continuously working product and prevents us from having a monolithic "acceleration" step to our project.

NVidia has released a guide on what they consider the best work-flow to accelerate your code with GPUs [3]. They suggest a four stage acceleration cycle as follows:

1. **Assess** the program, find bottlenecks and small sections of code that dominate execution time by profiling. Use Amdahl's and Gustafson's law make a prediction of the theoretical speedup that can be attained by paralellising this bottleneck and use this information to decide the best section to accelerate.
2. **Parallelise** the bottleneck found in the previous step. The code may be paralellised using OpenMP or CUDA. This may require some factorising if the code is not written in a way that easily allows paralellism.
3. **Optimise** the code as much as possible.
4. **Deploy** and analyse the results. If this was production software we would deploy to consumers at this stage.

4. IMPLEMENTATION

In this section we present the details of our accelerated implementations.

We use the cfitsio library to read and write FITS files and the CUDA toolkit version 7.5 for our GPU code. To simplify the evaluation the accelerated A' Trous algorithm is implemented independently from DUCHAMP. Our application accepts a three-dimensional telescope observation in a FITS file as input and outputs the noise reduced cube.

4.1 Phase 1: Naive Single Threaded

This phase is a reimplementation of the single threaded A' Trous algorithm in DUCHAMP.

Our rewrite is easier to understand than the original DUCHAMP implementation as we factored out many common methods and refactored common sections. The original version is written in one monolithic function, we split it up into easier to understand methods.

Identifying the next acceleration step follows from profiling our implementation as per the assess step of the APOD process. Finding where our program is the slowest helps us identify what part to accelerate. A profiler is used to find slow parts of the program. Figure 3 a show us the profiling information of this implementation. It can be seen that 91% of total execution time is spent in convolving data cubes and that this is the logical section to accelerate.

4.2 Phase 2: OpenMP

We accelerated our naive A' Trous algorithm with OpenMP on all the threads available to the CPU.

This was done by applying parallel directives around the convolution loop (step 7 of Algorithm 1). These directives let subsections of the data cube be convolved in parallel. This is possible since convolution does not have any intermediate data dependencies.

4.3 Phase 3

This phase ported convolution to the GPU. The partially de-noised data cube is copied to the GPU before convolution and back to the host after the convolution for further processing. In later phases we can avoid this copy by only modifying the data cube on the GPU.

Each thread on the GPU calculates the partial convolution of a single voxel with a single filter element. A block of threads calculates the complete convolution of a single voxel in the data cube. The partial results within a block are added using parallel reduction in $O(\log N)$ addition operations. The input and output matrices are stored in global memory and the filter in constant memory. Constant memory is a type of read-only global memory that caches lookups within the streaming multiprocessor for faster subsequent accesses.

4.4 Phase 4

This phase incorporates the benefits of separable filtering into the single threaded naive implementation. The three-dimensional filter is broken up into three one-dimensional filters and applied individually. Additional memory is required to store the intermediate results until all filters are applied.

4.5 Phase 5

We then port the separable implementation to the GPU. Each thread within a block calculates the result for a single voxel instead of splitting the computation over an entire block as done previously. Three kernels are execution, one for each one-dimensional filter.

C++ templates are used to specialise kernels for each filter used in the convolution. Using templates prevents branch divergence and unnecessary comparisons when the filter dimension is known at compile time.

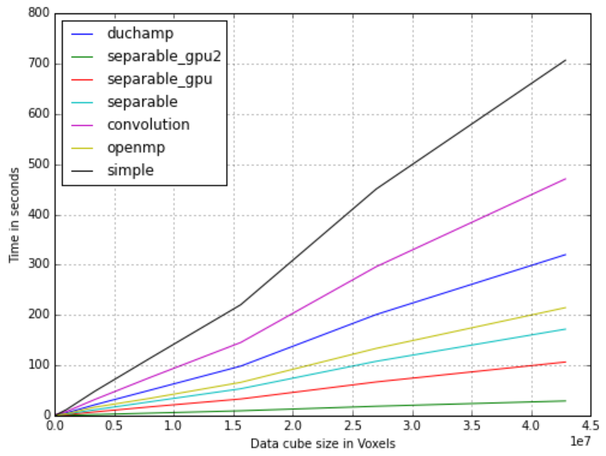
5. RESULTS AND DISCUSSION

Benchmarking shows that the separable GPU implementation (shown in Figure 2 a) is the fastest over all data cube sizes. Our reimplementation of the single-threaded algorithm (labeled “simple” on the graph) does not perform as well as the original DUCHAMP implementation. Not much effort was put into optimising this version and this is to be expected.

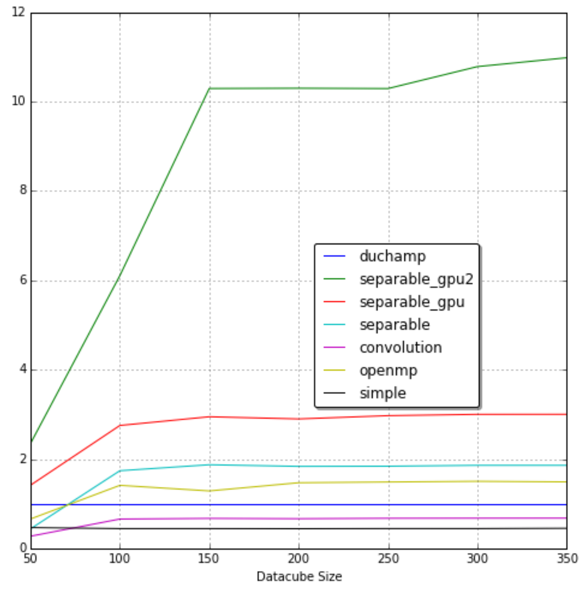
Profiling the simple implementation shows that 91% of the execution time is spent performing convolution. This is consistent with the analysis by Badenhorst et al.

The speedup graph on Figure 2 b shows the speedup of each implementation compared to DUCHAMP’s execution time. Each implementation shows a horizontal speedup for medium to large data cubes. This tells us that the speedup does not change based on data cube size with the exception of small data cubes. The increasing speedup for small data cubes can be attributed to constant time startup times that dominate when noise removal takes a short amount of time.

The single-threaded separable algorithm performed better than OpenMP acceleration running on four cores. The speedup relative to our simple implementation (on which the separable implementation is based) is 4.1x. The theoretical speedup of 8x discussed earlier is of the convolution algorithm in isolation. Our measured speedup is of the A’ Trous algorithm in its entirety not just the convolution algorithm. Badenhorst et al had a speedup of 3.7x for their

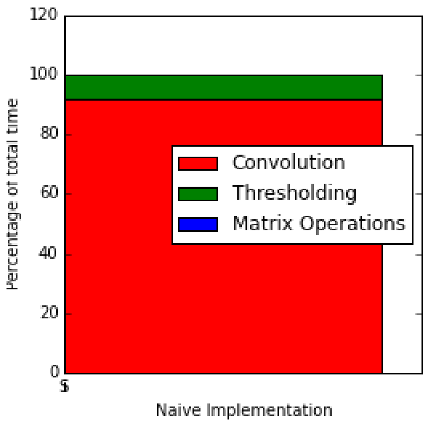


(a) Comparison of execution time of implementations with varying data cube size

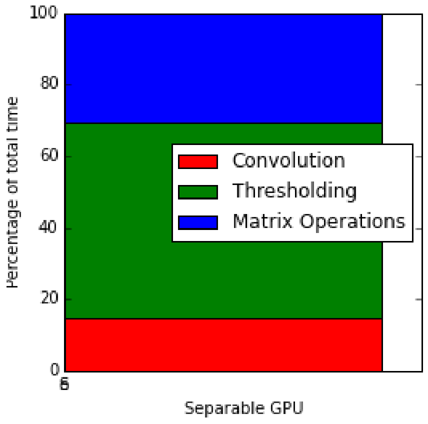


(b) Comparison of speedup of implementations

Figure 2: Duchamp is the unmodified original program. ”Convolution” is the first GPU accelerated convolution algorithm. Separable is the single-threaded separable convolution. ”Separable GPU” is the GPU accelerated separable implementation.



(a) Profiling information of simple naive implementation of various subroutines as a percentage of total execution time. It is clear that convolution dominates the execution time.



(b) Profiling information of separable GPU implementation of various subroutines as a percentage of total execution time. The majority of execution is spent thresholding the data cube with various manipulations of the data cube coming a close second.

Figure 3: Profiling information

original separable implementation. They went on to optimise the separable implementation further by transposing the data cube to improve memory accesses and use vector CPU instructions to achieve a 4.5x speedup.

This shows that parallelism is not always the best first step for acceleration. Algorithmic optimisation improves execution time without requiring specialised or more expensive hardware. This is beneficial to existing users of Duchamp as they do not need to purchase hardware and can simply update in order to benefit from this speedup. Integrating this change into Duchamp would be rather simple as all changes are contained within a single module and would not require software architectural changes.

The OpenMP benchmark was run on a desktop machine as well as the Hex computing cluster. The desktop machine has four physical cores each with Hyper-Threading enabled. Hyper-Threading lets a single physical core appear as two logical cores [2]. Figure 4 c plots the speedup of the implementation versus the number of threads running on the desktop. Speedup grows with the number of cores until we reach the number of physical cores available. Increasing the number of cores still gives us an increase in speedup but at a much lower rate. Adding four more virtual core only gives us a speedup from 3.5x to 4.5x.

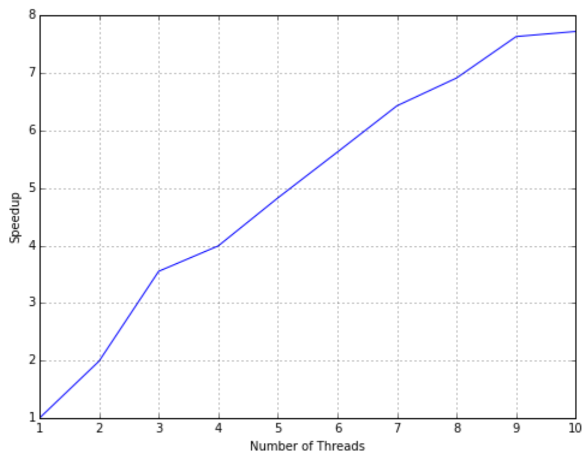
Running on the Hex computing cluster (Figure 4 a and b) showed that increasing the number of cores gives a linear increase in speedup. The speedup is less than the number of cores it is run on and appears to grow linearly.

The initial GPU port of the convolution algorithm split the calculation of calculating a single voxels result over a block of threads. It was quickly determined by profiling that this was not making efficient use of the available computing resources. We made each thread compute a single voxels result instead of only a part of it. This improved the speedup from 3x to 11x by a simple reallocation of work. The speedup of each separable GPU implementation is shown in Figure 2 b as separable_gpu and separable_gpu2 respectively.

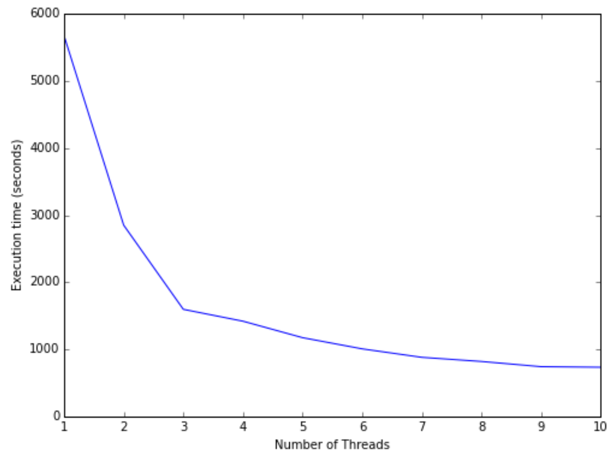
Porting the separable implementation the GPU gave a 5.8x speedup over the single-threaded version and 11x overall against the benchmark. Profiling this accelerated implementation shows that convolution no longer dominates the execution time of the algorithm and only contributes to 14% of total execution time (see Figure 3 b). 54% of execution time is now spent in the thresholding operation. Further acceleration of convolution would now show diminishing returns.

Thresholding involves removing all values of a matrix that are above some multiple of the median of the matrix. Finding the median is performed with the average case linear time QuickSelect algorithm. Parallel merge sort has a time complexity of $O(\log N)$ with N parallel cores which is better than the complexity of QuickSelect. Replacing QuickSelect with parallel merge sort on the GPU was more than twice as slow as the highly optimized QuickSelect from the C++ standard template library. Accelerating the thresholding proved to be difficult. Selavy suffered from this problem and solved it by calculating the median of a subset of the data which can be done faster at the expense of precision. We are constrained to having output equivalent to that of duchamp and was unable to perform this optimisation.

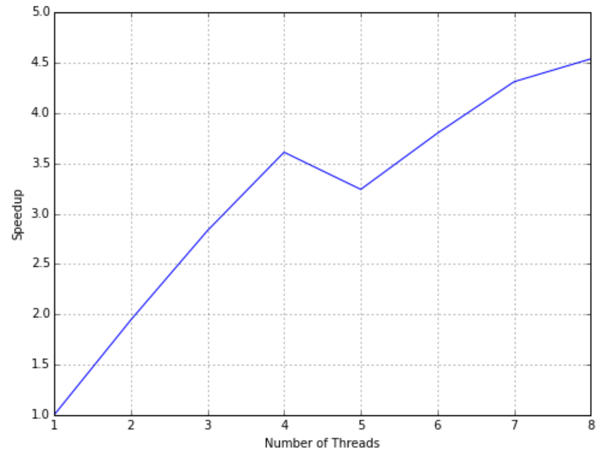
The various matrix addition and multiplication operations are the next largest contributers to execution time. Porting



(a) Speedup of OpenMP implementation versus number of threads. This experiment ran on the Hex computing cluster.



(b) Execution time of OpenMP implementation versus number of threads. This experiment ran on the Hex computing cluster.



(c) Speedup of OpenMP implementation versus number of threads. This experiment ran on a desktop PC with hyper-threading. The effects of hyper-threading can be seen by the decrease in the rate of speedup after reaching four threads.

Figure 4: OpenMP experiments.

this to the GPU further slowed down execution time and this was reverted. This was due to the overhead of copying to and from GPU memory.

We measured the accuracy of our implementations by calculating the absolute difference between our implementations results and DUCHAMP. We found that the average absolute difference was $2 \cdot 10^{-6}$. This is small enough that it does not affect the remaining

6. CONCLUSIONS

In this paper we presented a series of performance improvements to the Duchamp source finder. The algorithmic and hardware changes are orthogonal and can be applied independently of each other. In the case that GPU hardware is not available the algorithmic improvements still apply.

Separable filtering is an algorithmic improvement that improved our execution time by 4.1x.

We found that the execution time of wavelet reconstruction noise removal is largely dominated by massively paralellisable convolution operations. The next largest contributor to execution time are statistics operations which we did not benefit from GPU acceleration.

We conclude that GPU acceleration is a viable and useful way to speed up the Duchamp source finder. The speedup achieved by OpenMP on ten cores is 7.8x which is comparable to that of our GPU accelerated implementation. OpenMP and the use of separable filtering is a lower cost way to achieve very good speedups without purchasing additional GPU hardware.

Further improvements to noise reduction would require us to reduce the accuracy of our thresholding operation.

Overall we achieved a speedup of 11x over the Duchamp benchmark.

7. FUTURE WORK

This implementation limited the size of the data cube to the size of GPU memory. This can be subverted in future implementations by streaming the data cube into the GPU, processing and then streaming it off. Hopefully this can provide a way to deal with arbitrarily large data cubes.

The strides in separable filtering prevent it from using the cache effectively. There are optimisations that first rotate the input before applying a filter that has been shown to improve performance.

Duchamp's source merging phase compares every detected source with every other source. This can be improved by a constant factor using space partitioning techniques.

Duchamp has a long serial pipeline and improvements need to be made at every stage to completely improve performance.

8. REFERENCES

[1] Badenhorst. Acceleration of the noise suppression component of the duchamp source-finder.

[2] Chen, Y.-K., Tian, X., Ge, S., and Girkar, M. Towards efficient multi-level threading of h. 264 encoder on intel hyper-threading architectures. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International* (2004), IEEE, p. 63.

[3] Cuda, C. Best practices guide. *Nvidia Corporation* (2012).

[4] Dagum, L., and Enon, R. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE 5*, 1 (1998), 46–55.

[5] Hassan, A., Fluke, C. J., and Barnes, D. G. Unleashing the power of distributed cpu/gpu architectures: Massive astronomical data analysis and visualization case study. *arXiv preprint arXiv:1111.6661* (2011).

[6] Hill, M. D., and Marty, M. R. Amdahl's law in the multicore era. *Computer*, 7 (2008), 33–38.

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

[8] Jurek, R. 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 (2012), 251–261.

[9] Lutz, R. An algorithm for the real time analysis of digitised images. *The Computer Journal 23*, 3 (1980), 262–269.

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

[11] Walsh, A. J., Purcell, C., Longmore, S., Jordan, C. H., and Lowe, V. Maser source-finding methods in hops. *Publications of the Astronomical Society of Australia 29*, 03 (2012), 262–268.

[12] Westerlund, S., and Harris, C. Performance analysis of gpu-accelerated filter-based source finding for hi spectral line image data. *Experimental Astronomy 39*, 1 (2015), 95–117.

[13] Whiting, M., and Humphreys, B. Source-finding for the australian square kilometre array pathfinder. *Publications of the Astronomical Society of Australia 29*, 3 (2012), 371–381.

[14] Whiting, M. T. duchamp: a 3d source finder for spectral-line data. *Monthly Notices of the Royal Astronomical Society 421*, 4 (2012), 3242–3256.