The Astrophysical Journal Supplement Series, 239:28 (14pp), 2018 December https://doi.org/10.3847/1538-4365/aabe88

A GPU Implementation of the Correlation Technique for Real-time Fourier Domain Pulsar Acceleration Searches

Sofia Dimoudi[1,5], Karel Adamek[1], Prabu Thiagaraj[2], Scott M. Ransom[3], Aris Karastergiou[4], and Wesley Armour[1]
[1] Oxford e-Research Centre, Department of Engineering Science, University of Oxford, 7 Keble Road, Oxford OX1 3QG, UK; wes.armour@oerc.ox.ac.uk
[2] Jodrell Bank Centre for Astrophysics, The University of Manchester, Macclesfield, Cheshire, SK11 9DL, UK
[3] National Radio Astronomy Observatory, Charlottesville, VA, USA
[4] Department of Physics, University of Oxford, The Denys Wilkinson Building, Keble Road, Oxford OX1 3RH, UK
Received 2017 August 21; revised 2018 April 6; accepted 2018 April 12; published 2018 December 5

Abstract

The study of binary pulsars enables tests of general relativity. Orbital motion in binary systems causes the apparent pulsar spin frequency to drift, reducing the sensitivity of periodicity searches. Acceleration searches are methods that account for the effect of orbital acceleration. Existing methods are currently computationally expensive, and the vast amount of data that will be produced by next-generation instruments such as the Square Kilometre Array necessitates real-time acceleration searches, which in turn requires the use of high-performance computing (HPC) platforms. We present our implementation of the correlation technique for the Fourier Domain Acceleration Search (FDAS) algorithm on Graphics Processor Units (GPUs). The correlation technique is applied as a convolution with multiple finite impulse response (FIR) filters in the Fourier domain. Two approaches are compared: the first uses the NVIDIA cuFFT library for applying Fast Fourier transforms (FFTs) on the GPU, and the second contains a custom FFT implementation in GPU shared memory. We find that the FFT shared-memory implementation performs between 1.5 and 3.2 times faster than our cuFFT-based application for smaller but sufficient filter sizes. It is also 4–6 times faster than the existing GPU and OpenMP implementations of FDAS. This work is part of the AstroAccelerate project, a many-core accelerated time-domain signal-processing library for radio astronomy.

Key words: instrumentation: miscellaneous – methods: data analysis – methods: numerical – pulsars: general – telescopes

1. Introduction

The Square Kilometre Array (SKA) will transform the field of pulsar astrophysics (Kramer & Stappers 2015) by allowing for the discovery of a large fraction of the radio pulsar population beaming toward Earth. One particular area of opportunity for advances in fundamental physics comes from the potential discovery of a new population of pulsars with companion objects in orbits that allow more precision tests of theories of gravity (Demorest et al. 2010; Kramer 2014). Additional binary neutron star systems, where one or both objects are detected as radio pulsars, relativistic millisecond pulsars with white dwarf companions (Antoniadis et al. 2013; Ransom 2014), hierarchical triple systems (Ransom et al. 2014), and, of course, a pulsar orbiting a black hole, are prize targets of the SKA pulsar surveys.

The yield of the SKA surveys will be large due to a combination of the sensitivity of the telescope, a large number of simultaneously searched tied-array beams, and significant investment in computer hardware to search the data. Arguments pertaining to the large number of tied-array beams ($N \sim 1000$), the available spectral bandwidth, and the resolution in time and frequency that is required to discover new pulsars suggest that storage of the data will be extremely costly. Offline processing of raw pulsar data will give way to direct searches of the observed data streams, recording to disk only the raw data associated with carefully chosen pulsar candidates. There are two unavoidable consequences. First, searching for pulsars—and above all, the types of systems described above—must occur in real time. Second, the algorithms used to perform the search should not compromise on sensitivity.

Searching for periodic signals modulated by binary motion increases the complexity of simple periodicity searches. Among the compensation methods for binary motion that exist currently, linear acceleration searches are particularly favored for their simplicity and computational practicality while also delivering gains in signal recovery for a significant fraction of binary pulsars. Acceleration searches employ a linear, one-dimensional model for orbital motion related to a constant acceleration, which enables the signal recovery via the application of a number of trial acceleration values incorporated in the traditional search. There are currently two main methods used routinely for acceleration searches. The first method, which we refer to as the Time Domain Acceleration Search (TDAS), uses time resampling according to several trial acceleration values, followed by a Fast Fourier transform (FFT) for each trial. The second method is the Fourier Domain Acceleration Search (FDAS), which applies multiple filters corresponding to the trial acceleration values in the Fourier domain with the use of short-length FFTs. FDAS is considered generally faster and well suited for parallelism, compared to TDAS.

Employing acceleration searches on the vast amount of observation data that will be produced from the SKA, together with the requirement for real-time processing, poses a computational challenge that current CPU platforms may not be able to meet. That is especially true within the strict limitations on energy consumption that the operation of these facilities will have. Special-purpose parallel coprocessors such as Graphics Processor Units (GPUs) are ideal candidates for FDAS, as they offer massive parallelism through thousands of

[5] Currently at Oxford Centre for Clinical Magnetic Resonance Research, RDM Division of Cardiovascular Medicine, University of Oxford, John Radcliffe Hospital, Oxford OX3 9DU, UK.

computing cores on a single device with very high memory bandwidth and local caches. Their programming model is particularly suited to exploiting the memory locality that the FDAS technique exhibits with its short-length array operations. In addition, high-performance FFT libraries such as the cuFFT[6] are available for GPUs to easily perform very fast Fourier transforms, and the technology is evolving continuously toward energy efficiency with an increasing performance-per-Watt ratio.

Pulsar processing software with acceleration search capabilities can be currently found in a number of scientific software projects. The SIGPROC package (Lorimer 2011) and the GPU-enabled PEASOUP library (Barr 2014) use the TDAS method. The FDAS is implemented as part of the PRESTO (Ransom 2011) package. A tested GPU version of the PRESTO acceleration search currently exists.[7,8] This existing PRESTO GPU implementation is not optimized specifically for the real-time pulsar processing scenarios mentioned above, and to this extent, we believe that the community would benefit from a tunable library with a GPU FDAS component aimed specifically at performing real-time processing of radio telescope data. In this work, we propose a GPU implementation of the FDAS algorithm as part of AstroAccelerate (Armour et al. 2012), a GPU-enabled processing library for time-domain radio astronomy data.[9]

In the following, we give details of the techniques required to perform the aforementioned searches and show how we have tackled this problem using modern high-performance computing hardware and software techniques. The remainder of this document is organized as follows. Section 2 introduces the basic theoretical background for the two acceleration search methods mentioned above. In Section 3, we describe the application of the FDAS in more detail, with a focus on the functional workflow and computational considerations. Our core work of the GPU implementation of the FDAS algorithm is explained in detail in Section 4, and Section 5 presents and discusses experimental results from performance and signal-recovery measurements in comparison to existing software. Section 6 demonstrates the speed of the algorithm on the latest GPU hardware based on real-time parameters. Finally, we summarize and draw our conclusions in Section 7.

2. Effects of Orbital Motion in Pulsar Binaries and Corrective Methods

Pulsar searches typically detect the peak signal power at the pulsar spin frequency by applying the FFT to a series of input samples that are obtained during an observation interval. It should be noted that there exists an alternative to the FFT method for periodicity searches: the Fast Folding Algorithm (FFA; Staelin 1969), which has seen renewed interest in recent years and is being increasingly used in wide area surveys (see, e.g., Cameron et al. 2017), but its use is currently still limited. This work is based on FFT methods, which are the most commonly used.

Orbital motion in short-period binary pulsars causes the apparent pulsar spin frequency to change during the observation time as a result of its varying line-of-sight velocity. The effect of this is to spread the signal power over a number of neighboring Fourier bins, reducing the sensitivity of the FFT to these objects. Acceleration searches (Middleditch & Kristian 1984; Anderson et al. 1990; Johnston & Kulkarni 1991; Wood et al. 1991; Middleditch et al. 1993; Ransom et al. 2001, 2002; Eatough et al. 2013) are methods to partly correct for this effect by assuming a constant acceleration over a fraction of the binary orbit. For a circular orbit with period $P_{\rm orb}$, the line-of-sight velocity and acceleration are sinusoidal in time, as the orbiting pulsar moves away and toward the observer. For a small fraction of the orbit, the acceleration can be assumed to be constant. If the observation time $T_{\rm obs} \lesssim P_{\rm orb}/10$ (see, e.g., Johnston & Kulkarni 1991; Jouteux et al. 2002; Ransom et al. 2003), then, using the constant acceleration assumption, the frequency change can be approximated with a linear relationship to the initial frequency for the specified integration time, according to the Doppler effect.

If $v(t)$ is the observed radial velocity of a pulsar along the line of sight, then, using the Doppler formula, a time interval τ in the pulsar frame can be related to a corresponding interval t in the observed frame with the transformation

$$\tau(t) = \tau_0[1 + v(t)/c], \tag{1}$$

where c refers to the speed of light, τ_0 is a constant that is used to maintain the correct sampling during a transformation (e.g., Camilo et al. 2000), and terms higher than first order in (v/c) are neglected. To search for objects with unknown orbital parameters using Kepler's laws to calculate $v(t)$, one would need to search in a five-dimensional parameter space, which would be computationally impractical. Assuming a constant acceleration α during an observation interval $T_{\rm obs} \lesssim P_{\rm orb}/10$, we can approximate $v(t) = \alpha t$ and thus greatly reduce the search requirements. The received signal can then be resampled by running a linear interpolation over the original time series. This time-resampling method is used in TDAS.

A constant acceleration α corresponds to a constant frequency derivative, $\dot{f}$, which is related to the number of Fourier bins, z, that the signal frequency has drifted during the observation time T, such that

$$\alpha = \frac{\dot{f}}{f_0}c = \frac{zc}{f_0 T^2}. \tag{2}$$

The initial signal response can then be recovered coherently in the Fourier domain by correcting the Fourier response over a frequency–frequency derivative plane (f–$\dot{f}$) using only local Fourier amplitudes of the FFT of the signal, according to the assumed number of bins the frequency has drifted. Pulsar acceleration searches in the Fourier domain are based on the correlation technique (Ransom et al. 2002), which works by correlating a predicted Fourier response, or template, that corresponds to a frequency derivative, or z number, with the local Fourier amplitudes along the input FFT. The process is described mathematically by Equation (3). The corrected Fourier response A_{r_0} of the signal at frequency bin r_0 is recovered by correlating the m bins around r_0 with the frequency reversed and complex conjugated template $A^*_{r_0-k}$ of

[6] For the latest description of the cuFFT library, including accuracy and performance information, see the NVIDIA CUDA documentation website at http://docs.nvidia.com/cuda/cufft/index.html.

[7] Developed by Jintao Luo and available at https://github.com/jintaoluo/presto2_on_gpu.

[8] A second GPU version also exists that was under development at the time of this study. Developed by Chris Laidler and available at https://github.com/ChrisLaidler/presto.

[9] The AstroAccelerate source code is an open-source package licensed under GNU General Public License v3.0 and is publicly available on Github at https://github.com/AstroAccelerateOrg/astro-accelerate. Zenodo DOI:10.5281/zenodo.1212488.

the predicted normalized Fourier response to a given orbital acceleration:

$$A_{r_0} \simeq \sum_{k=[r_0]-m/2}^{[r_0]+m/2} A_k A^*_{r_0-k}. \quad (3)$$

This is effectively a matched filtering process, which can be efficiently computed in parallel for the whole signal over a number of templates by using Fourier techniques. The correlation technique has certain advantages over the time-domain resampling method (Ransom et al. 2001), in particular, the ability to allow independent and memory-local calculations, which are beneficial to the application of parallelism.

3. Fourier Domain Acceleration Search Method

The Fourier Domain Acceleration Search method is implemented as a matched filtering process, where the Fourier response of a long time series is convolved in the Fourier domain with a number of short Finite Impulse Response (FIR) templates. Using the convolution theorem, convolution then becomes a complex multiplication, which is a local operation and is highly parallelizable. For pulsar searches, the resulting correlated response must be searched for candidates. This is done by calculating the power spectrum and comparing the resulting Fourier power to a precalculated threshold. The power spectrum that results from the correlated output forms a two-dimensional plane of frequency and frequency derivative. This plane is referred to as the f–$\dot{f}$ plane.

The Fourier response of the signal must be Fourier transformed prior to the complex multiplication and inverse transformed after. However, the convolution theorem expects that the input signal and template are of equal length, which is not generally the case. Furthermore, the FFT assumes that the signal is periodic with a period equal to the length of the input, $\boldsymbol{N}$, and the convolution performed via the FFT is cyclic. The cyclic nature of the convolution causes contamination of $\boldsymbol{M}$ bins in the output signal, where $\boldsymbol{M}$ is the width of the template. These issues can be treated by zero-padding of the input and template to the size of their linear convolution, $\boldsymbol{N} + \boldsymbol{M} - \boldsymbol{1}$. This would result in performing a long FFT for each template. To avoid this, an overlap and save method is used (Press et al. 1992) that works on small blocks of the signal independently and accounts for the contamination in each block, as well as for the continuity of the convolution along the signal. A schematic representation of the process is shown in Figure 1.

As mentioned in Section 2, the acceleration search templates are complex, and the phases need to be preserved to enable coherent recovery, so the filters are centered at bin zero. In order to perform the Fourier domain convolution, zeros are added in the middle of the L-length filter array between $M/2$ and L–$M/2$+1, where each half of the template is placed, as illustrated in the h filter block of Figure 1. After an initial zero-padding at the beginning of the input array, which is done to avoid the wraparound pollution, equally sized blocks of the signal are picked for FFT convolution, each one starting from a region that overlaps the previous block by half the filter points. This ensures the continuity of the operation along the signal and provides the necessary padding for the aliasing effect. After the convolutions are performed, the contaminated edges of each block can be discarded, and the blocks with the remaining useful output points are concatenated. An additional advantage

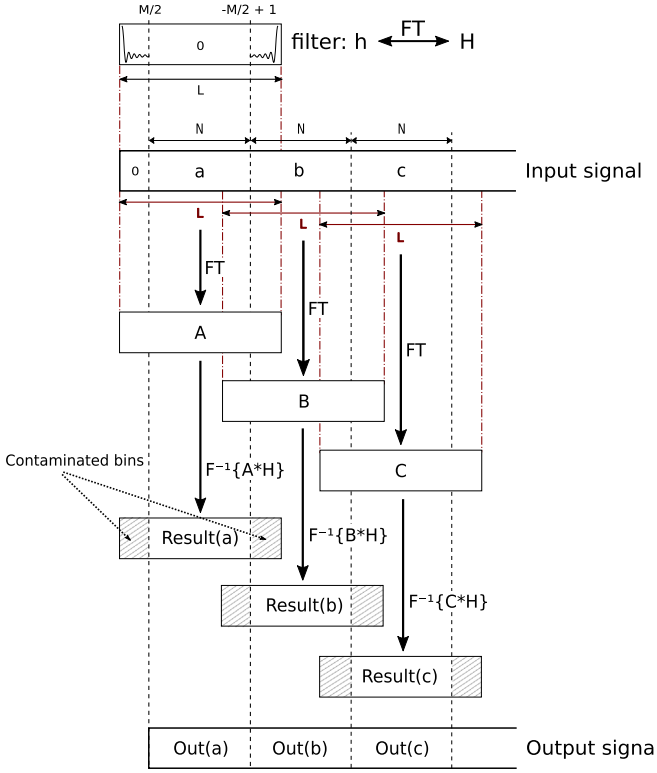

Figure 1. Overlap and save method in the FDAS. The complex filter h is placed at the edges of an array of length L so as to be centered at bin 0, and its Fourier transform is represented by H. Blocks of length L (indicated by the red double arrows and dash-dotted lines) are picked from the signal, overlapping continuous-length N segments (a, b, and c, separated by dashed lines) by $M/2$ bins at each end. After Fourier transform (FT) of the blocks to form A, B, and C, complex multiplication with H (indicated by *), and inverse Fourier transform, the edges in the results are discarded, and the remaining blocks, Out (a), Out(b), and Out(c), are concatenated to form the output signal.

of this method is that it allows one to choose the optimal block size for maximum performance of the FFT.

Computationally, the acceleration search process is broken into a number of individual steps, shown in Figure 2. First, a real-to-complex FFT is performed on the input time series, and low-frequency (red) noise is removed with the use of a local median filter (see, e.g., van Heerden et al. 2017 for a description). Next, the signal size is divided into blocks, and individual blocks are picked according to the overlap and save method. After a short step of normalization on the data of each block, the matched filtering is performed by applying a forward complex-to-complex FFT, complex multiplication with a number of precalculated templates, and inverse FFT of all the blocks that result from the block–filter pairs. The last step is to calculate the power spectrum and search for peaks of Fourier power. Pulsar signals have the form of pulsed waveforms with an often very narrow duty cycle, which results in the distribution of their Fourier power across a number of harmonics. In order to increase the probability of detection, a harmonic-summing step is typically performed before the search by adding the power that is distributed in the harmonics to that of the fundamental. The work described in this document deals only with the correlations and power calculations and does not include the steps of normalization, harmonic summing, and candidate selection.

In order to compensate for the loss of sensitivity that occurs for frequencies that fall outside Fourier bin centers (known as

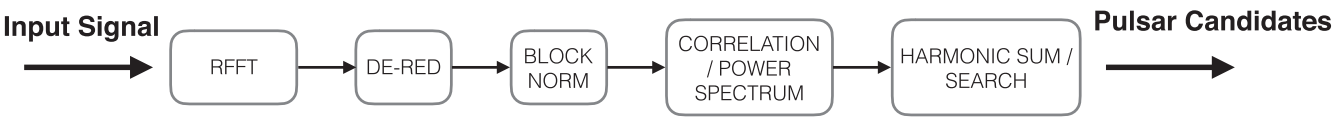

Figure 2. The acceleration search process.

"scalloping loss"), a method known as "interbinning" is commonly used in the Fourier analysis of pulsar observations (Middleditch et al. 1993; Lorimer & Kramer 2004). This involves the computation of the Fourier response at half-bin frequencies, which is approximated from the interpolation of the amplitudes of the two neighboring bins and given by

$$\mathcal{F}_{k+\frac{1}{2}} \simeq \frac{\pi}{4}(\mathcal{F}_k - \mathcal{F}_{k+1}). \quad (4)$$

Interbinning reduces the maximum loss in Fourier amplitude to ∼7% (from a potential 36% loss) for a relatively low computational cost, and for this reason, it is usually incorporated in periodicity searches. For the Fourier domain acceleration search, a 2-bin Fourier interpolation (Ransom et al. 2002) can be used by doubling the template and signal resolution using interleaving zeros prior to the correlations, or interbinning can be explicitly implemented by applying Equation (4) to the correlated Fourier amplitudes. The latter has the advantage of performing the Fourier domain computations on the initial signal resolution, while the former is more accurate but operates on double the number of points throughout the process. Our GPU application uses the interbinning technique on the Fourier amplitudes after the correlations.

4. GPU Implementation

We have developed and tested two GPU implementations for the FDAS algorithm. The first, which we use as a reference, makes use of the cuFFT library for the Fourier operations, along with custom functions that perform the convolution and power spectrum operations. The second implementation incorporates a short-length custom FFT that helps perform all the correlation and power spectrum processes using fast on-chip resources. The details of each implementation are described in this section. Our application is targeted to time-domain processing for radio telescopes and applies to a limited input size that does not exceed the GPU onboard memory resources. Current NVIDIA technology provides enough memory to enable the processing of signals with up to 2^{24} input samples and 300 correlation templates or close, which is within the specifications of the SKA for acceleration searches.

4.1. Using cuFFT

At the core of the matched filtering process described in Section 3 is a set of forward and inverse Fourier transforms. As a first step, we used the NVIDIA cuFFT library, which provides a highly optimized implementation of FFT routines written in CUDA for NVIDIA GPUs with an easy to use host interface. The rest of the steps were also performed on the GPU using the CUDA device code. The GPU workflow is shown in the schematic diagram of Figure 3.

The input signal that represents a time series is a one-dimensional array of n32-bit floating point elements. This array is transformed to the Fourier domain using a real-to-complex

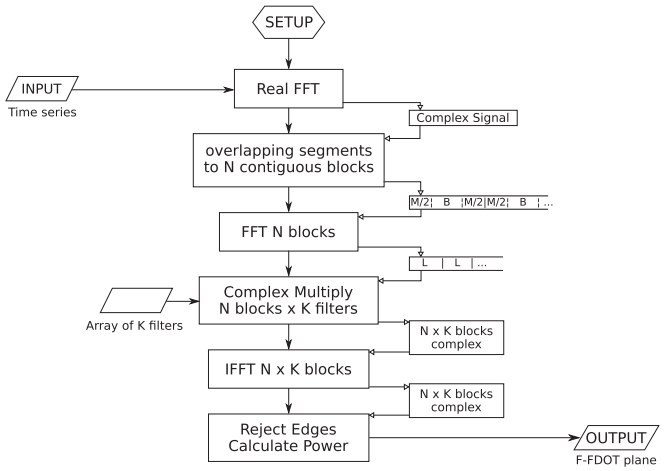

Figure 3. Workflow of Fourier domain matched filtering on the GPU using the cuFFT library. After an initial setup, the real FFT is applied on the input series, producing the complex signal array. Segments overlapping by half the filter length ($M/2$) are picked from this array according to the overlap and save method and placed contiguously into a separate array as N blocks of length $L = B + M$, where M is the length of our template and B is the number of useful output points to recover from each block. An array of K filters stored in memory is loaded to perform element-wise complex multiplication on a total of $N \times K$ L-length blocks, with the result written to an $N \times K \times L$ complex array. An inverse FFT is performed on this array, and finally, the edges are rejected and the Fourier powers are calculated, resulting in the real f-fdot plane, an array of total length $N \times K \times B$.

cuFFT routine. Overlap and save is implemented by way of copying N overlapping segments from the Fourier domain input to a new array in contiguous positions. The segments are of length $L = B + M$, where B is the size of the useful output points from each segment and M is the length of the filter. The remaining computations can then take place in independent memory regions. Each segment is transformed with a forward complex-to-complex FFT using a batched cuFFT routine, where multiple segments are transformed on the GPU with one call in parallel. Each transformed segment then is element-wise multiplied with each of $\boldsymbol{K}$ acceleration templates, producing an f–$\dot{f}$ of complex numbers that includes the contaminated regions. Next, the segments are inverse Fourier transformed with a batched routine, and this time the number of batches is multiplied by $\boldsymbol{K}$. Finally, the result is processed by a device kernel that computes the Fourier power for the whole f–$\dot{f}$ plane and stores the useful B elements of every segment to a floating point array contiguously.

The complex multiplication GPU kernel is structured in two ways. In the first scheme (Figure 4), a one-dimensional GPU grid of threads the size of a single segment loads all templates, with each thread loading a "column" of the stacked templates on device registers. It then loops along the signal and across all templates to perform the complex multiplications. For each signal segment, the algorithm loads the input, and then each thread iterates over its array of template elements, performing one complex multiplication per element and writing the result back to the device global memory. The process is repeated for

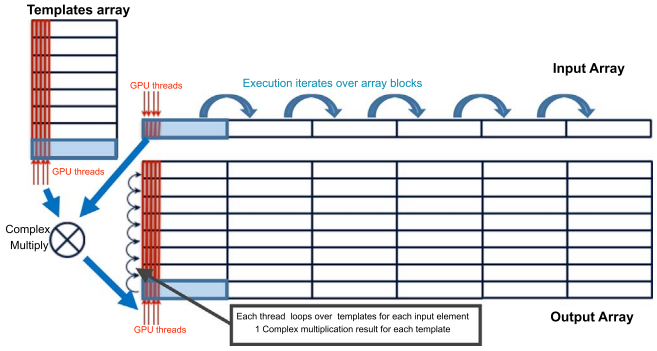

Figure 4. Schematic representation of complex multiplication GPU kernel using arrays of registers and iteration over templates.

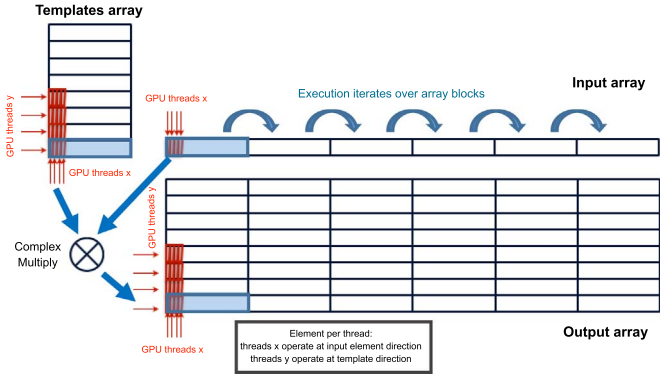

Figure 5. Schematic representation of complex multiplication GPU kernel using a two-dimensional GPU compute grid.

every signal segment. This scheme allows for maximum memory reuse with on-chip resources but suffers from register and shared-memory pressure, meaning that each thread block consumes a significant amount of GPU multiprocessor resources that limits the amount of blocks that can be run simultaneously on the device. There can also be register spillage to the global memory that induces a significant overhead to the execution. Optimizations were done for this scheme to avoid such spillage, as well as to reduce memory dependencies using loop unrolling and instruction-level parallelism techniques. To increase the memory read speed, we made use of the read-only data cache, which is available with NVIDIA devices of the *Kepler* generation and above and is known as the texture cache for previous generations. Data alignment to 32-bit floats was also used with shared-memory transactions to improve bandwidth.

The second scheme is shown in Figure 5. Instead of looping through the templates, this scheme utilizes a two-dimensional grid, where the vertical direction of the grid represents the template number. The grid then operates on a thread-per- f–$\dot{f}$ element basis. This scheme has very high multiprocessor occupancy by utilizing a very high percentage of the computing cores in parallel.

Both schemes are memory bandwidth–limited and have similar execution speeds in general, but the first shows relative gains over the second when a GPU architecture with a higher number of available registers is used. Because of the use of the cuFFT library and the low arithmetic intensity in this kernel, the memory bandwidth limitation cannot be improved significantly, but the first scheme was preferred because it proved to have more potential for improvement in performance with increases in device on-chip resources. This step, using currently available hardware, consumes approximately 20% of the total execution time.

The final step is to compute the power spectrum of the complex f–$\dot{f}$ plane. This is also a bandwidth-limited operation that only involves two multiplications and one addition per element read/write. It accounts for about 35% of the execution time.

This implementation of the FDAS is dominated by the cuFFT operations, which cost around 42% of the execution time, but is also limited by the memory bandwidth required to read and write data at intermediate stages, which is imposed by the use of cuFFT. Under these conditions, the template length has been chosen to produce the best cuFFT performance, and other GPU parameters, such as grid and thread block size for GPU kernel execution, as well as the kernel structure, were tuned to produce optimal overall performance, which also aligns with the cuFFT requirements.

4.2. Using a Custom FFT on the GPU Shared Memory

To overcome the limitations of the first implementation, we have developed a custom GPU FFT code and incorporated it into the algorithm. In order to avoid transferring data to and from GPU global memory, we aimed at performing the FFTs for a full segment entirely on the chip's shared memory, so that the result can be immediately used in the other steps, which are performed within the same kernel. For this reason, the template size in the current implementation is limited to 1024 points. The matched filtering algorithm was applied using the first complex multiplication kernel scheme, but it was found that there was no benefit in prestoring the templates. A complex conjugate symmetry that had been observed previously between the positive and negative acceleration templates was exploited so that half of the templates need not be transferred from the global memory but could be applied on the fly as the complex conjugate of their symmetric template contained in the other half. The power calculations were also done on shared-memory data, and the clean, concatenated result is stored in the global memory at the end. The grid of thread blocks with this method extends to the length of the Fourier domain input, and the number of threads in a block is equal to the FFT size; hence, there is no need to loop over the signal segments. The remainder of this section describes the FFT algorithms that were investigated for this implementation.

The discrete Fourier transformation (DFT) of a signal x is given by

$$X_m = \sum_{n=0}^{N-1} x_n \mathrm{e}^{-i2\pi nm/N} = \sum_{n=0}^{N-1} x_n \, \omega_N^{nm}, \tag{5}$$

where X_m is a signal in the frequency domain, x_n is a signal in the time domain, N is the signal length or DFT length, and the exponential factors $\omega_N^{nm} = \mathrm{e}^{-\mathrm{i}2\pi nm/N}$ are called twiddle factors. The inverse discrete Fourier transformation (IDFT) is

$$x_m = \sum_{n=0}^{N-1} X_n \mathrm{e}^{i2\pi nm/N} = \sum_{n=0}^{N-1} X_n \, \omega_N^{-nm}. \tag{6}$$

We can also express the DFT as a matrix multiplication,

$$X = \boldsymbol{F}x, \tag{7}$$

where $\boldsymbol{F}$ is a matrix of twiddle factors called the Fourier matrix $\boldsymbol{F} \in \mathbb{C}^{N\times N}$, and X and x are vectors X, $x \in \mathbb{C}^N$ in the frequency and time domains, respectively.

The DFT is an operation of complexity $O(N^2)$. The FFT allows us to calculate the DFT with a lower number of operations and higher precision. The original FFT algorithm was published by Cooley & Tukey (James & Cooley 1965), and since then, many different FFT variants have been published (Van Loan 1992).

The Fourier matrix in Equation (7) could be expressed as a multiplication of factorization matrices. In the case of the Cooley–Tukey algorithm, the Fourier matrix $\boldsymbol{F}_N$ can be written as

$$\boldsymbol{F}_N = \boldsymbol{A}_t \boldsymbol{A}_{t-1} \ \ldots \ \boldsymbol{A}_1 \boldsymbol{P}_N, \tag{8}$$

where $\boldsymbol{A}_q$ are factorization matrices, and $\boldsymbol{P}_N$ is a permutation matrix responsible for the reordering of time-domain vector x. The index t is related to the DFT length N, and if we restrict ourselves to radix-2 algorithms, it is given by $N = 2^t$. The factorization matrices $\boldsymbol{A}_q$ depend on the chosen FFT algorithm, as well as on the presence of the perturbation matrix $\boldsymbol{P}_N$.[10] For the exact definition of the factorization matrices and more details about matrix representation of FFT algorithms, we refer the reader to Van Loan (1992).

One more distinction has to be made with FFT algorithms. Each FFT algorithm has two possible variants, namely, decimation in time (DIT) and decimation in frequency (DIF). These variants reflect the manner in which we divide the input data. The relevance of these two variants for convolution lies in the position of the perturbation matrix $\boldsymbol{P}_N$. The expression for the DIT FFT algorithm is given in Equation (8). To obtain the DIF, we only need to transpose Equation (8). This will give us

$$\boldsymbol{F}_N = \boldsymbol{A}_t \boldsymbol{A}_{t-1} \ \ldots \ \boldsymbol{A}_1 \boldsymbol{P}_N, \tag{9}$$

$$\boldsymbol{F}_N^T = \boldsymbol{F}_N = \boldsymbol{P}_N \boldsymbol{A}_1^T \ \ldots \ \boldsymbol{A}_{t-1}^T \boldsymbol{A}_t^T. \tag{10}$$

Using these in Equation (7) results in

$$X = \bar{\boldsymbol{A}} \boldsymbol{P}_N x, \tag{11}$$

$$X = \boldsymbol{P}_N \bar{\boldsymbol{A}}^T x, \tag{12}$$

where we have replaced the factorization matrices with $\bar{\boldsymbol{A}}$. We can see that in the case of the DIT algorithm (Equation (11)), the FFT algorithm represented by $\bar{\boldsymbol{A}}$ requires the time-domain vector x to be reordered by the permutation matrix $\boldsymbol{P}_N$ and produces the correctly ordered frequency-domain vector X. The DIF algorithm, on the other hand, requires the in-order time-domain vector x and produces vector X, which needs to be reordered by the permutation matrix. The same holds for the inverse DFT. Since the convolution in the frequency domain is represented by point-wise multiplication, the result does not depend on the order of the elements within vectors that are being convolved, as long as both vectors are ordered in the same way. Thus, we can eliminate the permutation matrix $\boldsymbol{P}_N$ from our FFT implementation by choosing the DIF FFT variant for the forward FFT and the DIT variant for the inverse FFT algorithm. This enables us to save memory transactions and decrease execution time.

[10] Autosort algorithms like the Stockham algorithm do not require a perturbation matrix.

We have examined and implemented three radix-2 FFT algorithms. These are the Cooley–Tukey algorithm (James & Cooley 1965), the Pease algorithm (Pease 1968), and the Stockham algorithm (Cochran et al. 1967). We have considered DFTs with lengths of power of two only. To perform the DFT, we use two different algorithms. For the forward DFT, we use the DIF variant of the Pease algorithm, and for the inverse DFT, we use the DIT variant of the Cooley–Tukey algorithm.

For the calculation of the twiddle factors, we use CUDA fast-math intrinsics.[11] We reuse the twiddle factors by calculating two elements of the DFT transformation per thread. In the case of the inverse FFT, we are also performing two transformations simultaneously when exploiting the symmetry of the templates, as mentioned earlier in this section, and this further increases the reuse of twiddle factors, which are calculated once.

The Cooley–Tukey FFT algorithm is perhaps the best-known FFT algorithm. It requires reordering in order to produce the correct result. The memory access pattern for the algorithm is given by a butterfly diagram. During this operation, results of smaller DFTs are combined into larger DFTs. For smaller butterflies, this induces shared memory bank conflicts. However, by using thread shuffle instructions, we can remove these bank conflicts and reduce the synchronization requirements between threads.

We have also implemented the Pease FFT algorithm in the DIF variant, which requires reordering of the output (frequency-domain) vector. The Pease algorithm has the interesting feature that its data access pattern remains constant across all iterations. For the radix-2 variant, this memory access pattern is favorable for a shared memory implementation of the algorithm.

Our final arrangement used for convolutions is a combination of the Pease DIF FFT algorithm for forward FFT transformation and the Cooley–Tukey DIT FFT algorithm for inverse transformation. A combination of the DIF and DIT algorithms allowed us to eliminate the reordering step, which is otherwise necessary to produce correct results, and thus to reduce the number of operations performed and the execution time. When using the FFT without the reordering step, we have to use the same algorithm for the template transformations. We have used the Pease DIF FFT algorithm instead of the Cooley–Tukey DIF FFT algorithm because the Cooley–Tukey algorithm had higher register usage.

Using the shared memory FFTs in the matched filtering step eliminated the device memory dependence of the algorithm, and the execution performance is now limited by the efficiency of the use of on-chip resources and by synchronization.

5. Experimental Results

We have run a series of tests on our custom code in order to establish its signal recovery capability, as well as to assess computational performance. As described in Section 3, the full acceleration search process in PRESTO consists of several functional parts, which are shown in Figure 2. The FDAS GPU implementation optimizes only the matched filtering part, and this is the subject of the comparisons to follow.

[11] More in Appendix D of the NVIDIA CUDA programming guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide.

Table 1
Hardware Specifications

	K80	M60	M40	P100
No. of GPUs on card[a]	2	2	1	1
No. of Streaming Multiprocessors (SMs)	2 × 13	2 × 16	24	56
Total no. of cores	2 × 2496	2 × 2048	3072	3584
Base/Max core clock (MHz)	560/875	899/1178	948/1114	1189/1328
Main memory bandwidth (GB s^{-1})	2 × 240	2 × 160	288	732
Main memory size (GB)	2 × 12	2 × 8	12	16
L2 cache size (MiB)	1.5	2.0	3.0	4
Peak Shared memory bandwidth (est.)[b] (GB s^{-1})	2 × 2712	2 × 4494	6374	9519

Notes.
[a] Where there are two GPUs, only one is used in the experiments.
[b] The shared memory bandwidth was estimated using the formula: BW (bytes s^{-1}) = (bank bandwidth (bytes)) × (clock frequency (Hz)) × (32 banks) × (no. of multiprocessors).

5.1. Computational Performance

We compared computational speed between PRESTO, PRESTO GPU, and our FDAS GPU implementation with cuFFT and our custom FFT for varying input signal sizes and a varying number of correlation templates. Only the real FFT, correlations, and power spectrum calculations were timed in all cases. The tests were performed both with and without the use of interbinning. PRESTO was run using 20 OpenMP threads on the 10 CPU cores (20 Hyperthreading cores) of an Intel Xeon E5-2650. Our GPU codes were run on the NVIDIA Maxwell architecture M40 card, as well as on the dual-GPU NVIDIA K80 (*Kepler* architecture) and M60 (Maxwell) cards using only one of the two GPUs on board (when using each GPU of the dual cards to process a different time series, the execution speed of the total number of independent time series is doubled). Preliminary measurements on the latest NVIDIA architecture, Pascal, with the Tesla P100 card are also presented.

We first compare our basic GPU implementation to PRESTO and PRESTO GPU. We then examine the performance of our custom GPU code against our basic GPU version and, finally, compare this to PRESTO and PRESTO GPU. The comparison with PRESTO GPU is only meant to give an indication of the difference in execution time, and for this reason, we only used the M40 card, which is a representative high-end single-GPU card of the Maxwell architecture. Table 1 lists the main specifications of the GPU hardware.

It should be noted that, because the cuFFT requires 1.5× the amount of GPU global memory storage as the custom code, due to the need to store the result of the inverse complex FFTs, some trials with signal size of 2^{23} and a high number of templates could not be executed with this version, because they exceeded the available GPU memory capacity. These can be seen as missing parts of lines in the results figures. This is a limitation of the cuFFT version. The custom code allows larger f–$\dot{f}$ plane sizes to be processed. As the GPU architecture evolves, it is expected that memory storage will not be a limiting factor in future generations for our problem sizes.

5.1.1. Comparisons of FDAS-cuFFT to PRESTO

In this section, we report on the comparison in computational performance of our FDAS with the cuFFT code against both the CPU OpenMP and GPU version of PRESTO.

PRESTO was run using 20 OpenMP threads on a 10 core Intel Xeon E5-2650 (Haswell), and only the correlations were timed (including the long real FFT) for input signal sizes of 2^{21}, 2^{22}, and 2^{23} (the closest powers of 2 for 2, 4, and 8 million samples), with a varying number of templates up to 256. The PRESTO GPU and FDAS-cuFFT run the same test on an NVIDIA M40 GPU card.

Figure 6 shows the execution time of each code along with the number of filter templates and the gain in speed of the FDAS-cuFFT from each of the PRESTO codes. The execution time of all three codes appears to be approximately linear with the number of templates, as a result of the similar blocked algorithm employed, which iterates over FFT blocks. This imposes approximately the same number of memory transfers but on different memory subsystems. The method is inherently memory bandwidth-dependent; therefore, the difference in execution time relies heavily on memory bandwidth. The CPU version appears to have some significant variations from linearity at points, which are mainly the effect of the OpenMP parallelization across filters. The speedup graphs show a characteristic drop in speedup. The steepness of the decrease is due to an approximately constant difference in execution time of the long real-to-complex FFT, which is run on the CPU in both PRESTO codes and is at the order of 30, 75, and 220 ms for the signal lengths of 2^{21}, 2^{22}, and 2^{23}, respectively.

Examining the difference in execution time between the three codes without including the real FFT, we can see (Figure 7) that the difference in performance between the GPU codes without interbinning shows a relatively small variation compared to its average, which indicates that the two codes have many similarities. However, the PRESTO GPU has an overhead compared to the FDAS-cuFFT, possibly due to the synchronous PCIe memory transfers and some additional indexing and other preparatory operations. We also observe that the FDAS correlation kernel provides a small performance increase between 32 and 96 templates. This difference is at the range of 50 ms with the longest input, although the speedup becomes very small for large numbers of templates. When interbinning is used, the FDAS implementation has a significant gain, maintaining twice the execution speed of the PRESTO GPU in the worst case, with an approximately linear increase in execution time difference. When comparing the GPU versions to the CPU, we can also identify a proportional increase in the difference with the number of templates, which shows that the method improves in performance with size on the GPU.

Overall, if we exclude the real FFT calculations on the CPU, the basic FDAS-cuFFT implementation without interbinning does not provide a significant improvement over the PRESTO GPU version, although it does demonstrate an advantage for the case of real-time processing by removing various overheads associated with PRESTO indexing calculations and PCIe memory transfers and optimizing the correlation GPU kernel. The implementation of interbinning in the FDAS offers a significant improvement over the PRESTO GPU, with a minimum of double the execution speed.

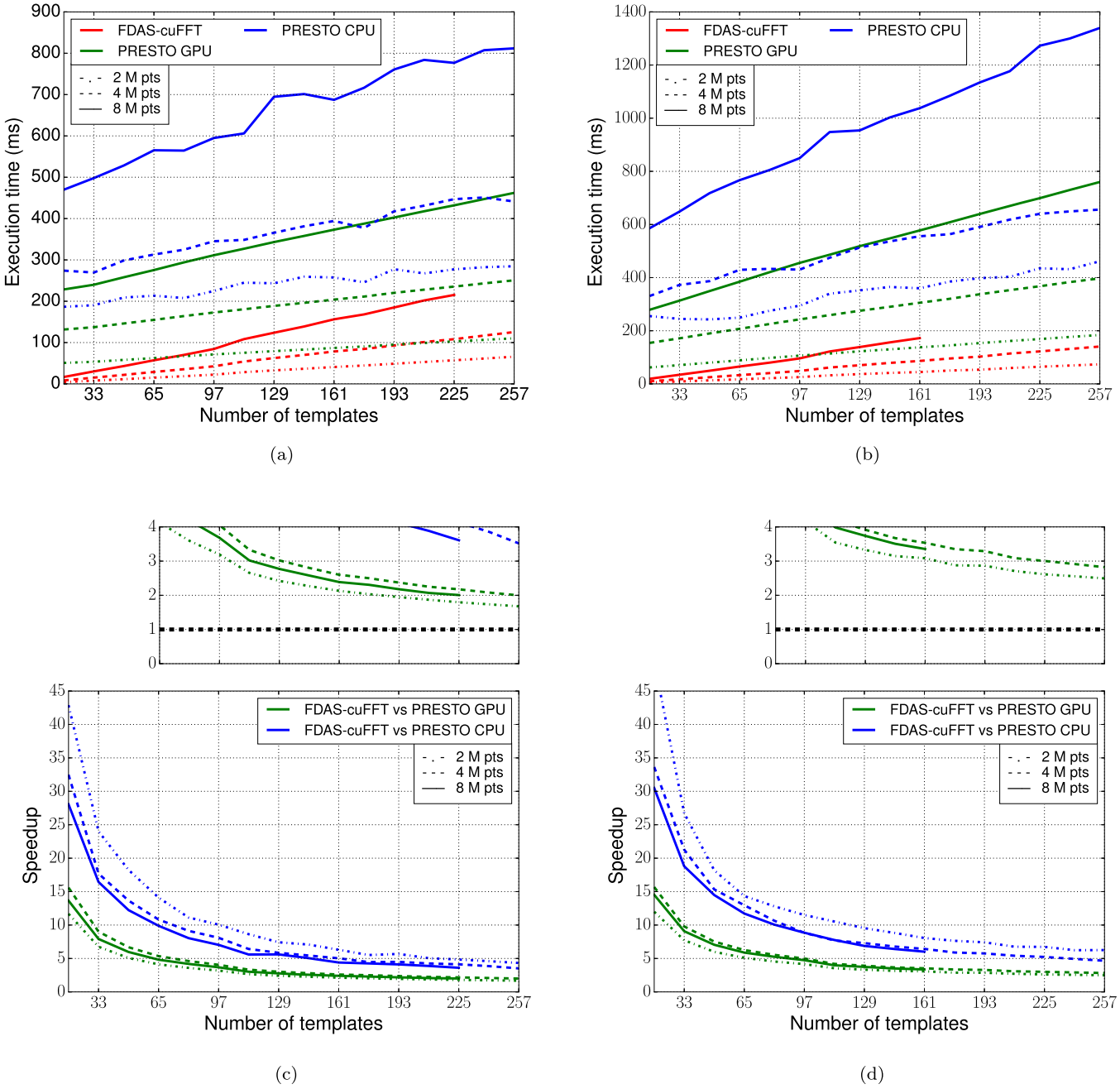

Figure 6. Comparison of the FDAS-cuFFT to the PRESTO CPU and GPU versions on the NVIDIA M40. (a) and (b): measured execution time with increasing number of filter templates. (c) and (d): speedup of the FDAS-cuFFT compared to PRESTO CPU and GPU. (c) and (d), top: zoomed y-axis to show lower speedups in more detail, with the black dashed horizontal line at the point where speedup $= 1$. The graphs on the left are for runs without interbinning, and those on the right have interbinning.

5.1.2. Comparisons between FDAS GPU Versions with cuFFT and Custom FFT

The cuFFT version is dominated by the Fourier transforms, whose speed is dependent on the filter length, while the custom FFT version is restricted to a filter length of up to 1024 points. On the other hand, because of the overlap and save scheme, as the filter size decreases, the number of blocks to be convolved along the signal increases, and so does the number of FFTs performed. For this reason, we have chosen to run each code with its optimal filter sizes. Tests of both codes with varying filter sizes have shown that, with the available testing hardware, the best size for the cuFFT version is 8192 points, while the custom FFT code has shown varying performance in different cases for sizes of 512 and 1024 points, and we chose to use both of these in the comparison. Figure 8 shows the speedup of the custom FFT code with respect to the reference GPU code (FDAS-cuFFT) against the number of filter templates processed. Results are plotted separately for the cases of single bins and interbinning and for the different GPU devices (see caption of Figure 8 for details). Solid lines indicate filter sizes of 512 points and dotted lines of 1024 points. The code has been tuned on each card for

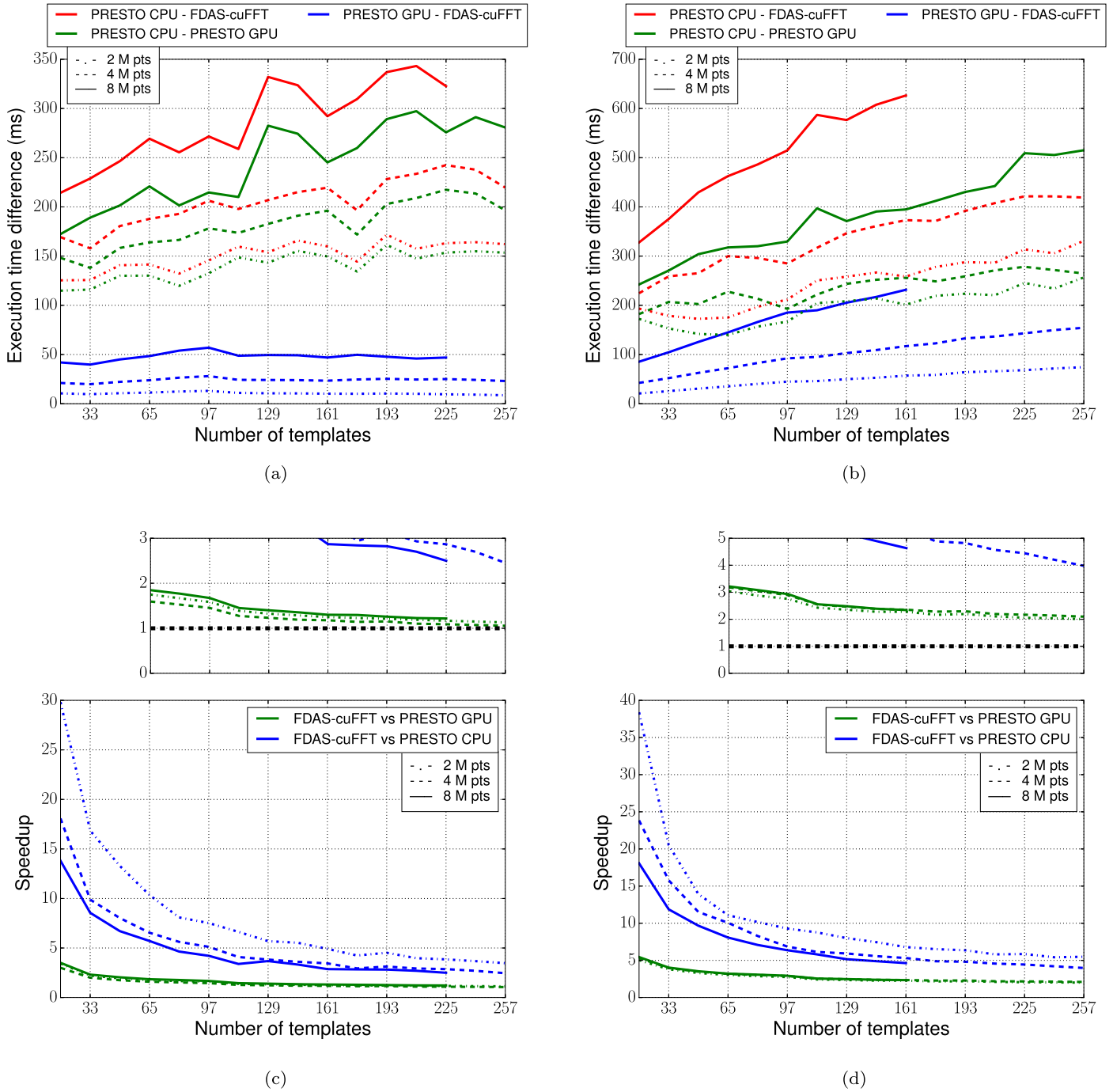

Figure 7. Comparisons between the FDAS-cuFFT and the PRESTO CPU and GPU on the NVIDIA M40, excluding the real FFT on the CPU. (a) and (b): difference in execution time between codes. (c) and (d): speedup of the FDAS-cuFFT compared to the PRESTO CPU and GPU, with the top zoomed to the *y*-axis at lower speedups and the black dashed line indicating where speedup = 1. The left panel is without interbinning, and the right panel is with interbinning.

optimal performance. Preliminary measurements without any specific optimization or tuning are shown for the P100 card in Figure 9.

The speedup varies according to the number of templates and the filter size, but by choosing the optimal filter length, a speedup between 1.5 and 3.2 can be achieved on Maxwell architectures, depending on the number of templates.

We observe that in all cases, the speedup of the FDAS with custom FFT against the cuFFT follows a similar pattern. The lines for the 512 and 1024 long filters cross in a particular region of the number of templates for each card. For numbers of templates below this region, filter sizes of 512 points perform better, while for higher template numbers, there is a continuous drop in speedup, which is worse for the 512-point filters. This indicates that with the increasing number of templates, the custom FFT code is becoming compute-bound, and the effect of the increased number of convolution blocks to be processed with the shorter filter lengths becomes evident. The worst-case speedup is seen with the single GPU on the *Kepler* generation K80 card; however, performance gains are improved with interbinning. In contrast, with the Maxwell generation cards, there is an overall improvement in speedup in

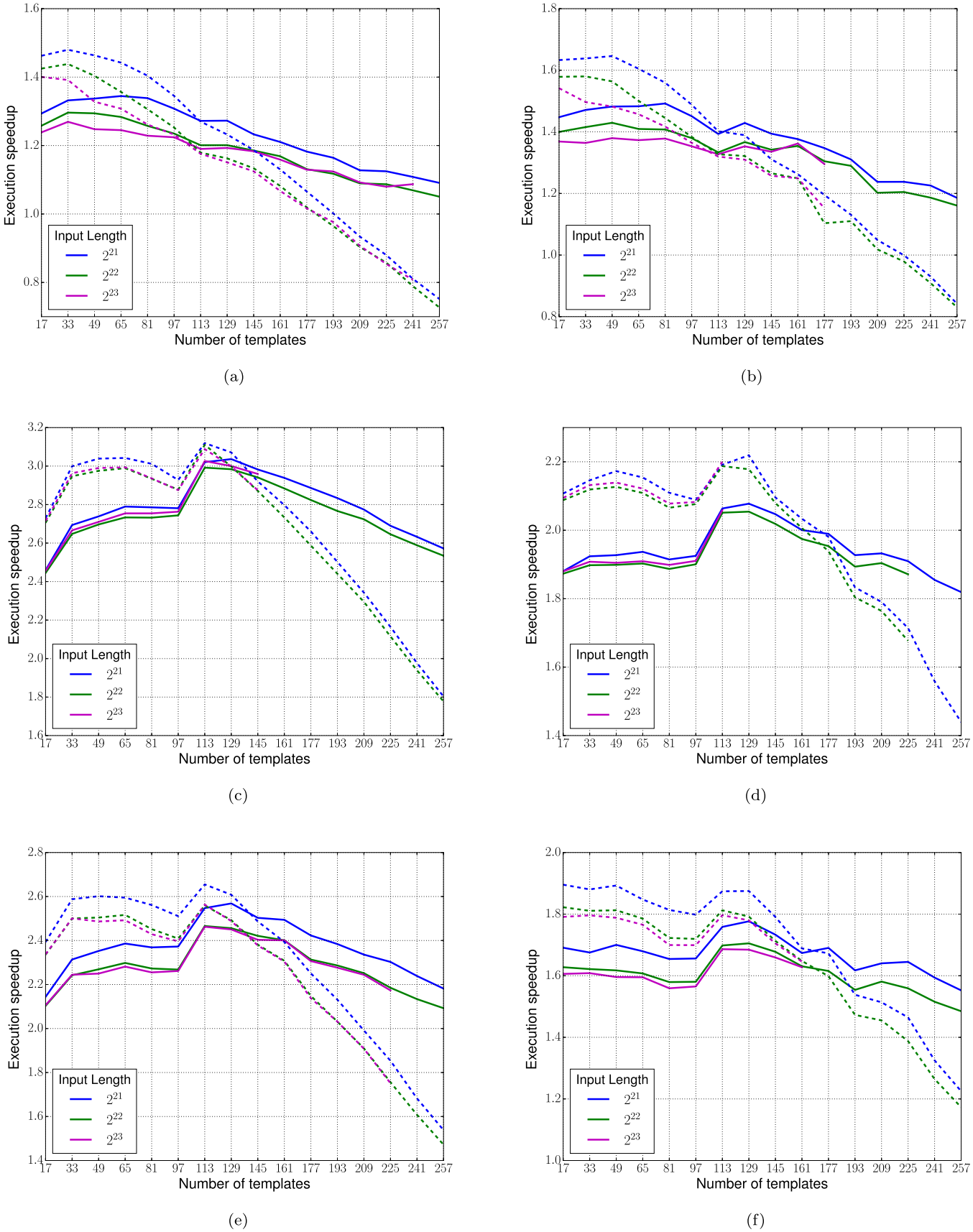

Figure 8. Comparison of the FDAS with cuFFT and custom FFT on the NVIDIA Tesla K80 single GPU (panels (a) and (b)), M60 single GPU (panels (c) and (d)), and M40 (panels (e) and (f)). Graphs on the left are without interbinning, and graphs on the right are with interbinning. The solid lines are for a template length of 1024 points, and the dashed lines are 512 points. Speed gains drop with an increasing number of templates but are maintained to a better level with a template length of 1024.

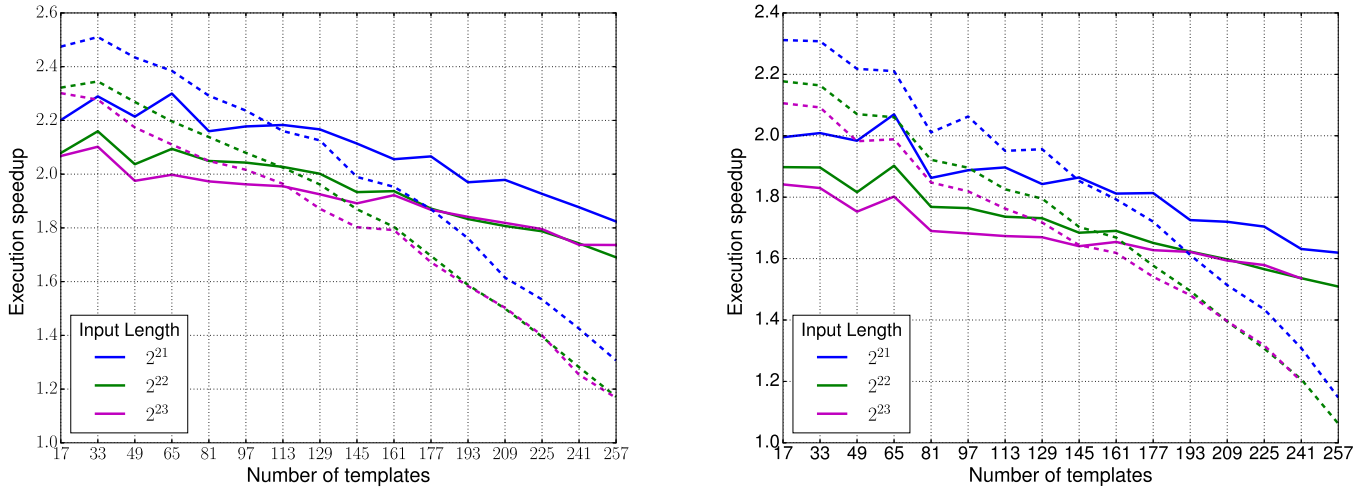

Figure 9. Speedup of the FDAS with custom FFT to the FDAS with cuFFT on the NVIDIA P100 (a) without interbinning and (b) with interbinning. Without any optimizations, we obtain better gains for a smaller number of templates and similar gains with a higher number of templates to that of the Maxwell architecture.

favor of the custom FFT, but this is reduced when interbinning is used. Interbinning costs in execution time compared to noninterbinning on the K80 GPU are $\sim\times1.5$ for the cuFFT code, $\times1.2$ for the custom FFT, and $\sim\times1.1$ and $\times1.6$ on the Maxwell cards. When using the single-GPU card M40, the speedup behavior is very similar to the M60 but displays a lower level of speedup overall, which is consistent with the device memory bandwidth. In particular, the lowered memory bandwidth per GPU in the dual-GPU M60 card worsens the performance of the cuFFT code to 61% and 79% of that of the K80 and 57% and 58% of that of the M40, as it is largely memory bandwidth–limited. However, this dual GPU architecture is desirable because it can increase speed when running on both GPUs while keeping energy consumption lower than using a single GPU card.

The custom FFT code clearly benefits from the Maxwell architecture. A key improvement in Maxwell from the *Kepler* architecture that enables this to happen is the shared memory capacity and bank size, which enable higher shared memory efficiency. Code profiling has shown that the custom FFT code is sensitive to shared memory efficiency, while the cuFFT code is almost entirely dependent on global memory bandwidth. In addition, the Maxwell processor has a dedicated shared memory of 96 KB per multiprocessor, while the cache on the K80 is divided between the L1 cache and shared memory. Much higher shared memory throughput is achieved on Maxwell (see Table 1), due to the fact that the K80 chip optimizes this bandwidth only with the utilization of an optional 8-byte bank size, and as a result, it performs worse for 32-bit floating points and integers. In particular, the M60 card has a much lower peak memory bandwidth than the K80, and consequently, the relative performance of the cuFFT code is reduced.

Finally, on a P100 without any specific tuning, the execution time seems to scale with an approximately 75% decrease, while the speedup between the FDAS custom FFT and cuFFT, shown in Figure 9, is maintained at a level of 1.5, similar to the M40 for the highest number of templates. There is also considerable improvement with a filter size of 1024 for lower numbers of templates.

5.2. Signal Recovery

We assess the quality of the signal recovery performance with our custom FFT algorithm, which has achieved the fastest results. We have run a set of trials using the PRESTO acceleration search code as a reference for signal recovery. Acceleration searches are known to cover a limited parameter space (Johnston & Kulkarni 1991; Camilo et al. 2000; Ransom et al. 2003; Knispel et al. 2013; Ng et al. 2015), and to ensure our comparison is applied within a space where PRESTO is known to be effective, we used simulated pulsars with the typical orbital parameters of millisecond pulsars in binary systems.

The input signals were produced using the program "fake" from the SIGPROC pulsar processing package. Using pulsar periods of 1.0, 3.0, 5.0, 6.0, 7.0, and 10.0 ms and a short orbital period of 2.0 hr in a circular orbit, we have modified the companion mass between 0.01 and 0.6 $M_{\odot}$ to represent low-to-medium-mass companions and obtain a range of acceleration values. The orbital phase was also varied to obtain negative accelerations. To ensure sufficient recovery of the first signal harmonic, the input pulse level was kept relatively high, at 0.1 for a single pulse, and the signal was produced with a wide duty cycle of 45%. The observation length is a constant 536 s with a sampling interval of 64 μs. The upper-edge channel frequency was set at 1550 MHz with 4096 channels of width 0.075 MHz each, which is consistent with the SKA design parameters. The peak signal-to-noise ratio (S/N) produced by our custom GPU code was compared to that produced by PRESTO, and the Fourier amplitudes were interpolated during the correlations in PRESTO and interbinned after the correlations in the FDAS custom FFT. As mentioned in Section 3, due to a lack of a harmonic summing module, the S/N for both PRESTO and the FDAS custom FFT is measured only on the fundamental frequency without harmonic summing. Isolated pulsars were also produced with the same telescope parameters at each of the periods used for the accelerated pulsars. The peak S/N of the isolated and accelerated pulsars was measured. The measurements taken are the S/N peaks calculated immediately after the correlations in both codes. The S/N calculations in the

Table 2
Comparative Search Results from PRESTO with Fourier Interpolation (PR) and FDAS Custom FFT with Interbinning (FD) for 45% Duty Cycle Pulsars

Period (ms)	Accel. (PR) (m s^{-2})	Δz (bins)	Δr (bins)	S/N (PR)	S/N (FD)	S/N Isol. (PR)
1	12.48	2	0	51.13	54.25	92.68
	−10.40	0	−0.5	47.01	52.26	⋯
	22.89	0	0	41.88	53.66	⋯
	−25.00.	−2	0	36.75	52.74	⋯
3	31.21	0	0	69.61	78.08	81.95
	−31.20	0	0	62.59	61.17	⋯
	43.69	0	0	76.37	78.97	⋯
	−49.94	−2	0	58.59	59.65	⋯
5	62.43	0	0	58.68	55.70	65.70
	−72.83	0	−0.5	61.60	61.13	⋯
	83.24	0	0	61.31	60.93	⋯
6	87.41	0	0	35.92	40.48	65.55
	−87.33	0	0	37.26	37.74	⋯
	112.40	0	0	37.08	48.98	⋯
	−112.27	0	0	39.37	44.20	⋯
7	101.98	0	0	51.78	59.00	68.66
	−101.88	0	0	53.62	58.03	⋯
	116.56	0	0	59.48	62.48	⋯
	−116.43	0	0	54.53	60.92	⋯
10	83.25	0	0	49.74	56.25	86.67

FDAS were done using the PRESTO S/N functions, which were embedded in the FDAS custom FFT. Table 2 lists the search results. The differences quoted are subtractions of the FDAS custom FFT results from the PRESTO results, and a negative sign indicates that the respective PRESTO value is lower. Here Δz indicates the difference in the frequency derivative bin where the peak was detected, and Δr is the difference in frequency bins of the detected frequencies along the time series. As a duty cycle of 45% is very high, we have also performed a limited number of searches in simulated pulsars with a 10% duty cycle for a more realistic example. The observation files were created for a small subset of the existing data set described above using the same specifications, except for the duty cycle, which was modified to 10%. The search results are displayed in Table 3.

There are very few discrepancies in the frequency and frequency derivative bins where the signal peaks were detected, each corresponding to a single f and z step, i.e., one half bin and one template in the computations. The discrepancies in S/N recovery vary, with the largest variations being due to a higher value detected by the FDAS custom FFT. The cause of this result is unclear, and it does not appear to relate directly to the differences in the detected frequency and acceleration. Possible causes are the method of interbinning, which is an approximation of a 2-bin Fourier interpolation and is applied after the correlations, as well as the different FFT algorithms used, which are well known to vary in accuracy[12] (Schatzman 1996; Johnson & Frigo 2008). Nevertheless, the FDAS custom FFT is succeeding in signal detection sufficiently close to the frequency and acceleration of interest, which we believe provides a proof of concept. More accurate results and further metrics can be obtained in postprocessing by fine-binning small parts of the f–$\dot{f}$ plane around the area of detection (see, for example, Ransom et al. 2002). Finally, it should be noted that the single-bin, interbinned, and Fourier-interpolated data have different statistical properties that affect the S/N calculations. The FDAS custom FFT at this stage does not have its own methods for signal significance calculations, and the subject is beyond the scope of this work. Future work, however, should address this issue by investigating the properties and effects of this interbinning scheme and developing and implementing the appropriate methods for calculating the S/N.

[12] Also see the FFTW website page titled "FFT Accuracy Benchmark Comments" at http://www.fftw.org/accuracy/comments.html.

Table 3
Comparative Search Results from PRESTO with Fourier Interpolation (PR) and FDAS Custom FFT with Interbinning (FD) for 10% Duty Cycle Pulsars

Period (ms)	Accel. (PR) (m s^{-2})	Δz (bins)	Δr (bins)	S/N (PR)	S/N (FD)	S/N Isol. (PR)
1	10.40	0	0	31.39	33.65	43.73
3	31.21	0	0	30.67	32.17	43.04
5	62.43	0	0	33.70	31.35	29.87
6	87.41	0	0	23.01	24.52	30.63
7	87.41	0	0.5	25.07	29.11	30.42
10	83.24	0	0	25.00	28.55	35.60

6. Operational Scenario

To examine the potential of our algorithm for execution within a streaming pipeline with the latest GPU hardware, we run a set of acceleration searches on the Tesla P100 for input signal lengths of $N = [2^{21}, 2^{22}, 2^{23}, 2^{24}]$ points. The signals were processed with $m = [64, 96, 128, 196, 256]$ templates each, except for the longest signal, which could fit up to 242 and 254 templates on a single GPU. Using a sampling of 64 μs, as proposed for the SKA, we derive the observation period for each time series and define this as the real-time limit for the corresponding signal. We then calculate the number of independent time series that can be processed for each (N, m) combination. Table 4 lists the results.

The number of time series achieved is affected much more by the number of templates than the signal size, which is expected, since the signal size is directly related to the real-time limit, i.e., when the signal size is increased, the time limit is

Table 4
Number of Independent Time Series of Various Numbers of Samples Processed with the FDAS with Custom FFT in Real Time for a Constant Sampling Interval of 64 μs

No. of Samples in Tseries	No. of Templates	No. of Tseries Processed	
		No Interbins	With Interbins
2^{20}	64	41427	33893
	96	24900	19775
	128	18847	15277
	192	12436	9898
	256	8039	6620
2^{21}	64	41645	34086
	96	26801	21123
	128	20147	15568
	192	12216	9686
	256	8338	6757
2^{22}	64	41850	34322
	96	27614	21756
	128	20455	16079
	192	12629	10043
	256	8477	6867
2^{23}	64	42099	34196
	96	27826	21744
	128	20627	16199
	192	12807	10157
	256	8600	6954
2^{24}	64	42454	34393
	96	28040	21832
	128	20708	16275
	192	12828	10203
	254/242[a]	8741	7537

Note.
[a] Number of templates limited by the GPU memory capacity. Quoted numbers represent templates for no interbins/with interbins.

also increased. The lowest number achieved is over 6500. Although this scenario does not include other important and computationally intense parts of the processing, such as harmonic sums, the high numbers listed in Table 4 indicate that the matched filtering step has been accelerated to such an extent that there is enough execution overhead to allow for the rest of the processing.

7. Conclusions

We have presented two GPU implementations of the correlation technique for FDAS, one based on the use of the cuFFT library and the other on a custom FFT computed entirely using on-chip GPU resources. Our custom FFT implementation provides significant execution speed gains over a wide range of template numbers compared to our cuFFT code, as well as to the other existing GPU and OpenMP implementations. Overall, we found that we are able to achieve a speed increase of at least 6 times from the existing CPU OpenMP version of the correlations in the PRESTO acceleration search and at least 4 times that of the PRESTO GPU version.

We have seen the performance advancing from the *Kepler* to the Maxwell GPU architecture, with a particular benefit in performance per energy consumption from dual GPU cards. The recently released Pascal architecture also features improvements in chip design that favor on-chip resources and shared memory bandwidth, all of which allow for the scalability of the algorithm to future architectures, something that was strongly indicated from our initial tests on the P100 card.

Our algorithm also appears to be capable of sufficiently accurate recovery of accelerated signals in comparison to the PRESTO acceleration search code. The use of interbinning on the correlated Fourier amplitudes demonstrates a benefit in the algorithm's execution speed compared to Fourier interpolation, but the effect it has on detection is not yet well understood. The particular scheme has not been characterized appropriately yet, and future analysis and investigations would be needed in order to determine and improve the accuracy of detection.

The utilization of shared memory and registers for intermediate steps instead of global memory can expand in the future to accommodate a harmonic summing, search, and candidate selection algorithm on the GPU, with the potential for big reductions in execution time and energy consumption for the acceleration search process, which would be particularly useful for real-time time-domain processing on data-intensive next-generation instruments such as the SKA.

This work is supported by a Leverhulme Trust Project Grant (ARTEMIS: Real-time discovery in Radio Astronomy). It has also received support from the members of the Oxford pulsar group, Christopher Williams and Jayanth Chennamangalam, as well as support from Prof. Ben Stappers and the Time Domain Team, a collaboration between Oxford, Manchester, and MPIfR Bonn, to design and build the SKA pulsar search capabilities. Scott M. Ransom is a Senior Fellow of the Canadian Institute for Advanced Research and is partially funded by the National Science Foundation's Physics Frontiers Center award 1430284.

Software: PRESTO (Ransom 2011), PRESTO 2 on GPU (https://github.com/jintaoluo/presto2_on_gpu), SIGPROC (Lorimer 2011).

ORCID iDs

Scott M. Ransom https://orcid.org/0000-0001-5799-9714

References

Anderson, S. B., Gorham, P. W., Kulkarni, S. R., Prince, T. A., & Wolszczan, A. 1990, Natur, 346, 42
Antoniadis, J., Freire, P. C. C., Wex, N., et al. 2013, Sci, 340, 448
Armour, W., Karastergiou, A., Giles, M., et al. 2012, in ASP Conf. Ser. 461, Astronomical Data Analysis Software and Systems XXI, ed. P. Ballester, D. Egret, & N. P. F. Lorente (San Francisco, CA: ASP), 33
Barr, E. 2014, Peasoup, v1.0, Zenodo, doi:10.5281/zenodo.10178
Cameron, A. D., Barr, E. D., Champion, D. J., Kramer, M., & Zhu, W. W. 2017, MNRAS, 468, 1994
Camilo, F., Lorimer, D. R., Freire, P., Lyne, A. G., & Manchester, R. N. 2000, ApJ, 535, 975
Cochran, W. T., Cooley, J. W., Favin, D. L., et al. 1967, in Proceedings of the IEEE 55, 1664
Demorest, P. B., Pennucci, T., Ransom, S. M., Roberts, M. S. E., & Hessels, J. W. T. 2010, Natur, 467, 1081
Eatough, R. P., Kramer, M., Lyne, A. G., & Keith, M. J. 2013, MNRAS, 431, 292
James, W., & Cooley, J. W. T. 1965, MaCom, 19, 297
Johnson, S. G., & Frigo, M. 2008, Implementing FFTs in Practice, Rice University, Houston TX: Connexions, http://cnx.org/content/m16336/
Johnston, H. M., & Kulkarni, S. R. 1991, ApJ, 368, 504

Jouteux, S., Ramachandran, R., Stappers, B. W., Jonker, P. G., & van der Klis, M. 2002, A&A, 384, 532
Knispel, B., Eatough, R. P., Kim, H., et al. 2013, ApJ, 774, 93
Kramer, M. 2014, IJMPD, 23, 1430004
Kramer, M., & Stappers, B. 2015, in Advancing Astrophysics with the Square Kilometre Array (AASKA14), Fundamental Physics with Pulsars (Trieste: SISSA), 36
Lorimer, D. R. 2011, SIGPROC: Pulsar Signal Processing Programs, Astrophysics Source Code Library, ascl:1107.016
Lorimer, D. R., & Kramer, M. 2004, Handbook of Pulsar Astronomy (Cambridge: Cambridge Univ. Press)
Middleditch, J., Deich, W., & Kulkarni, S. 1993, in Isolated Pulsars, ed. K. A. van Riper, R. I. Epstein, & C. Ho (Cambridge: Cambridge Univ. Press), 372
Middleditch, J., & Kristian, J. 1984, ApJ, 279, 157
Ng, C., Champion, D. J., Bailes, M., et al. 2015, MNRAS, 450, 2922
Pease, M. C. 1968, J. ACM, 15, 252
Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. 1992, Numerical Recipes in C, The Art of Scientific Computing (2nd ed.; New York: Cambridge Univ. Press)
Ransom, S. 2011, PRESTO: PulsaR Exploration and Search TOolkit, Astrophysics Source Code Library, ascl:1107.017
Ransom, S. 2014, APS April Meeting Abstracts
Ransom, S. M., Cordes, J. M., & Eikenberry, S. S. 2003, ApJ, 589, 911
Ransom, S. M., Eikenberry, S. S., & Middleditch, J. 2002, AJ, 124, 1788
Ransom, S. M., Greenhill, L. J., Herrnstein, J. R., et al. 2001, ApJL, 546, L25
Ransom, S. M., Stairs, I. H., Archibald, A. M., et al. 2014, Natur, 505, 520
Schatzman, J. C. 1996, SIAM J. Sci. Comput., 17, 1150
Staelin, D. H. 1969, Proceedings of the IEEE, 57, 724
van Heerden, E., Karastergiou, A., & Roberts, S. J. 2017, MNRAS, 467, 1661
Van Loan, C. 1992, Computational Frameworks for the Fast Fourier Transform, Society for Industrial and Applied Mathematics (Philadelphia: Society for Industrial and Applied Mathematics)
Wood, K. S., Norris, J. P., Hertz, P., et al. 1991, ApJ, 379, 295