The Astrophysical Journal Supplement Series, 269:29 (12pp), 2023 November https://doi.org/10.3847/1538-4365/acfef6

Accelerating Dedispersion Using Many-core Architectures

Jan Novotný[1,2], Karel Adámek[1], M. A. Clark[3], Mike Giles[4], and Wes 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] Research Centre for Theoretical Physics and Astrophysics, Institute of Physics, Silesian University in Opava, Czech Republic
[3] NVIDIA, 2788 San Tomas Expressway, Santa Clara, CA 95051, USA
[4] Mathematical Institute, University of Oxford, Oxford, OX2 6GG, UK
Received 2023 June 29; revised 2023 September 25; accepted 2023 September 26; published 2023 November 7

Abstract

Astrophysical radio signals are excellent probes of extreme physical processes that emit them. However, to reach Earth, electromagnetic radiation passes through the ionized interstellar medium, introducing a frequency-dependent time delay (dispersion) to the emitted signal. Removing dispersion enables searches for transient signals like fast radio bursts or repeating signals from isolated pulsars or those in orbit around other compact objects. The sheer volume and high resolution of data that next-generation radio telescopes will produce require high-performance computing solutions and algorithms to be used in time-domain data-processing pipelines to extract scientifically valuable results in real time. This paper presents a state-of-the-art implementation of brute force incoherent dedispersion on NVIDIA graphics-processing units and on Intel and AMD central-processing units. We show that our implementation is 4× faster (8-bit 8192 channels input) than other available solutions, and we demonstrate, using 11 existing telescopes, that our implementation is at least 20× faster than real time. This work is part of the AstroAccelerate package.

Unified Astronomy Thesaurus concepts: GPU computing (1969); Computational methods (1965); Computational astronomy (293); Pulsars (1306)

1. Introduction

An upcoming new generation of radio telescopes, such as the Square Kilometre Array[5] (SKA; Carilli & Rawlings 2004), will simultaneously observe many different regions of the radio sky. Each simultaneous observation will have high time resolution and fine channelization of the observed bandwidth, giving rise to large data volumes at high data rates. These data are expected to make storing all data for offline analysis impractical, necessitating faster-than-real-time data-processing software.

To extract the very faint signals present in the noisy data produced by these telescopes, many processing steps have to be performed on the data. One of the more computationally expensive steps is dedispersion. The dedispersion process increases the signal-to-noise ratio (S/N) of received signals from the emitting object being studied by shifting samples in different frequency channels in time, thus correcting for the time delay introduced by dispersion. Samples at the same time stamp are then summed over frequency channels, increasing the S/N and probability of detection.

The dispersion of the emitted pulse occurs due to the interaction between photons in the pulse and the ionized interstellar medium (ISM) through which they travel. Dispersion has the effect of causing a frequency-dependent time delay ($\Delta\tau$) in the photon's propagation. Specifically, lower-frequency photons within the pulse ($f_{\rm low}$) are observed later than their high-frequency ($f_{\rm high}$) counterparts (see Lorimer & Kramer 2004). This time delay is proportional to the inverse square of the frequency, given by the cold plasma dispersion law:

$$\Delta\tau = \mathrm{DM}\ C_{\rm DM}\left(\frac{1}{f_{\rm low}^2} - \frac{1}{f_{\rm high}^2}\right), \tag{1}$$

with the constant of proportionality $C_{\rm DM} = 4148.8 \times 10^3$ MHz^2 pc^{-1} cm^3 s. The parameter DM in Equation (1) is referred to as the dispersion measure and is defined as the integral of the electron column density ($n_{\rm e}$) along the line of sight (distance d) between source and observer, i.e.,

$$\mathrm{DM} = \int_0^d n_{\rm e}\, dl. \tag{2}$$

Given the quadratic relationship between time delay and frequency (see Equation (1)), dispersion is governed by a single free parameter, the dispersion measure. When searching for new events from unknown sources, the distance between the object and the observer is unknown, which means all possible dispersion measures must be calculated and searched. Detecting and studying such events in real time on the scale required for the next generation of radio telescopes, together with the computational complexity of dedispersion, necessitates a high-performance computing (HPC) solution for real-time observation and detection (Barsdell et al. 2010; Armour et al. 2012; Fluke 2012).

To remove the effects of dispersion, two different approaches can be used: coherent and incoherent dedispersion. The coherent approach uses information about the observed phase of the pulse to reconstruct the pulse profile as it was emitted (within the limits of the inhomogeneous scattering of the ISM). The incoherent method applies the appropriate time delay to each independent frequency channel in channelized intensity data. Although the coherent method is more accurate and has higher sensitivity, its computational requirements are

[5] https://www.skao.int/

far more demanding than those for the incoherent method. As such, when performing surveys of the radio sky, it is common to employ incoherent dedispersion.

Several different codes exist with differing implementations of dedispersion specifically for graphics-processing units (GPUs; Magro et al. 2011; Barsdell et al. 2012; Sclocco et al. 2016; Bassa et al. 2017; Zackay & Ofek 2017; Kong et al. 2021). There are other implementations of the dedispersion transform—for example, using fast Fourier transforms (Bassa et al. 2022). However, that approach is not capable of detecting FRBs or accelerated pulsars, due to there being weak or no Fourier response to these signals. In this article, we present our implementations of dedispersion for different computer architectures, NVIDIA GPUs, and Intel and AMD central-processing units (CPUs). We compare the performance of these implementations with the state-of-the-art packages, specifically looking at data-processing rates and sensitivity.

All of the implementations that we present in this article have been developed for AstroAccelerate[6] (Dimoudi et al. 2018; Adámek & Armour 2020; Adámek et al. 2020, 2022; Armour et al. 2020; Novotný et al. 2022; White et al. 2023), a many-core accelerated software package for processing time-domain radio astronomy data. AstroAccelerate is actively used as part of scientific pipelines like MeerTRAP (Rajwade et al. 2020; Morello et al. 2022) and Greenburst (Agarwal et al. 2020).

Incoherent dedispersion is described in Section 2. The implementation for the CPU is presented in Section 3, and that for the GPU is in Section 4. In Section 5, we discuss performance results of AstroAccelerate for different scenarios on selected many-core platforms, and compare these results with the performance of other software packages like Heimdall. Real-time performance on selected telescopes is presented in Section 5.5, and our conclusions are summarized in Section 6.

2. The Direct Dispersion Transform

Incoherent dedispersion is the process of shifting detected power data in time inside each individual frequency channel, which collectively make up the total telescope bandwidth. Shifts are applied to counter the effect of interstellar (or intergalactic) dispersion before integrating the data over the frequency bandwidth of the telescope, to increase the S/N of astrophysical signals detected by the telescope.

Here, we present the direct (sometimes called brute force) approach to dedispersing measured power data. As well as being the simplest approach for performing the task of dedispersion, it has two significant advantages. The first is that the algorithm is exact; by this, we mean that the errors associated with this approach are at the discretization level of the instrument. The second is that the algorithm can be written in such a way that is particularly suited to execution on many-core devices.

Incoherent dedispersion can be algebraically expressed as

$$S(\mathrm{DM}, t) = \sum_{f=0}^{N_f} x(f, t + \Delta t(\mathrm{DM}, t, f)), \tag{3}$$

where a frequency-dependent shift in time $\Delta t(\mathrm{DM}, t, f)$ is calculated for each digitized frequency channel. By substituting $f_{\mathrm{low}} = f_c - \Delta f/2$ and $f_{\mathrm{high}} = f_c + \Delta f/2$ into Equation (1), we can express the time shift in the form

$$\Delta t(\mathrm{DM}, t, f) \approx 8297.4 \times 10^3 \left(\frac{\mathrm{DM}\Delta f}{f_c^3} \right), \tag{4}$$

where f_c corresponds to the central frequency of the band and Δf is a finite bandwidth that is $\Delta f \ll f_c$. Applying the correct time shifts to each frequency channel results in a shifted signal that appears as though it has arrived at the same instant in time. The process of incoherent dedispersion is shown pictorially in Figure 1, in which dotted lines (red and black) correspond to different DM trials 1, 2,..., N. From Equation (3), we can derive a simple pseudocode (see Algorithm 1) that outlines the direct dedispersion approach.

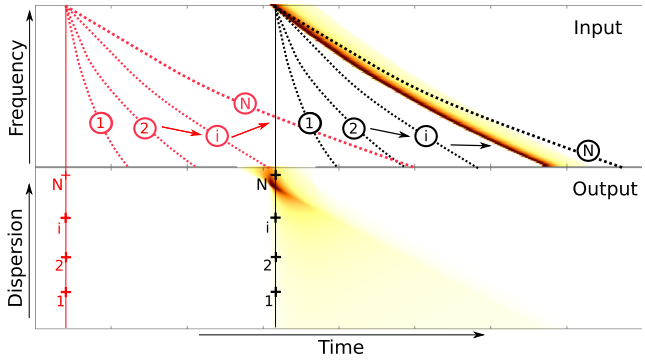

Figure 1. Representation of the incoherent dedispersion approach. The top panel represents the input data, and the bottom shows the results (output). For simplicity, we present a clear single signal. The dotted lines correspond to individual DM trials (the summation of data points along the line) computed for a given time sample (showed for DM trials 1, 2, i and N). The sum of the line maps to one point (cross) in the output field.

Algorithm 1. Pseudocode for the direct dispersion transform, where N_t is the number of time samples, N_f is the number of dispersion measures searched, and N_f is the number of frequency channels.

Data: $x(f, t)$
Result: $DM(dm, t)$
for $t = 0,\ldots, N_t - 1$ **do**
 for $dm = 0,\ldots, N_{DM} - 1$ **do**
 $S(dm, t) = 0.0$;
 for $f = 0,\ldots, N_f - 1$ **do**
 $h = \Delta t(dm, t, f)$
 $S(dm, t) \leftarrow S(dm, t) + x(f, t + h)$
 end
 $DM(dm, t) \leftarrow S(dm, t)$
 end
end
return $DM(dm, t)$

From Algorithm 1, we see that, for N_{DM} trial dedispersion searches over power data that have N_t time samples and N_f frequency samples, the computational complexity of the algorithm is $\mathcal{O}(N_{\mathrm{DM}} N_t N_f)$.

The arithmetic intensity I is another important characteristic of an algorithm. The roofline model of Williams et al. (2009) defines I as a ratio of the number of floating point operations performed by the algorithm per amount of data required in bytes read or written by the algorithm to RAM—or GPU main memory, in the

[6] https://github.com/AstroAccelerateOrg/

Table 1
Hardware Specifications and Compiler Specifications on the Tested GPUs and CPUs

	V100 SXM2	A100 SXM4		Xeon Gold 6230	Xeon Phi 7290	AMD EPYC 7601
CUDA cores	5120	6912	No. of cores/threads	20/40	72/288	32/64
No. of SMs	80	108	Base clock frequency	2.1 GHz	1.5 GHz	2.2 GHz
Boost core clock	1455 MHz	1410 MHz	Frequency for AVX-512	2.0 GHz	1.3 GHz	⋯
Memory clock	877 MHz	1215 MHz	Mem. bandwidth	140.8 GB s^{-1}	115.2 (400+) GB s^{-1}	276.573 GB s^{-1}
Dv. m. bandwidth	900 GB s^{-1}	1555 GB s^{-1}	Cache size	27.5 MB L3	36 MB L2	64 MB L3
Shared m. bandwidth	14,899 GB s^{-1}	19492 GB s^{-1}	DP compute	⋯	⋯	1126 GFLOPS
Memory size	32,768 MiB	40,960 MiB	DP compute (AVX-512)	1280 GFLOPS	2995.2 GFLOPS	⋯
TDP	300 W	400 W	TDP	125 W	245 W	180 W
Critical arithmetic intensity	15.4	11.6	Critical arithmetic intensity	9	7.5	4
Other specifications						
NVIDIA driver						495.29.05
CUDA version						11.5.119
ICC version						18.0.3
OpenCL version (CPU)						Intel 2019.3.208
Compiler flags						
AMD EPYC 7601						-std=c99 -O2 -Wall -Wextra -qopenmp -march=core-avx2 -qopt-prefetch -fma -ftz -fomit-frame-pointer -finline-functions -qopt-streaming-stores=never
Xeon Phi 7290						-qopenmp -fp-model fast=2 -std=c99 -O2 -fma -xMIC-AVX512 -align -finline-functions -no-prec-div -ipo -DOPEMP_SPEC -qopt-streaming-stores=never
Xeon Gold 6230						same as Xeon 7290 except: -march=core-avx2

Note. The value of the memory bandwidth in brackets on Xeon Phi 7290 corresponds to the case of using the 16 GB Multi-Channel DRAM (MCDRAM).

case of GPUs. The value of I can help us to identify what will limit the performance of an algorithm. When the algorithm is limited by the number of floating point operations it needs to perform, it is called a compute-bound algorithm. If the memory bandwidth limits the algorithm by not providing enough data per second, we have a memory-bound problem.

In order to decide whether an algorithm on a given hardware platform (CPU, GPU) is compute-bound or memory-bound, we need to look at the critical arithmetic intensity I_{crit} that represents a turning point from an algorithm being memory-bound to being compute-bound and vice versa. For a given hardware platform, the critical arithmetic intensity I_{crit} is a ratio of computational performance in FLOPS and the memory bandwidth in bytes. If I_{alg} of an algorithm is $I_{\text{alg}} < I_{\text{crit}}$, that algorithm is memory-bound. For $I_{\text{alg}} > I_{\text{crit}}$, the algorithm will be compute-bound. On the modern hardware $I_{\text{crit}} > 1$, see Table 1 for values of I_{crit}.

The dedispersion's arithmetic intensity for a single DM trial is given as

$$I_d = \frac{n_{\text{o}}}{n_{\text{b}}} = \frac{N_{\text{f}} - 1}{N_{\text{f}} + 4} \rightarrow 1, \tag{5}$$

where n_{o} is the number of floating point operations performed, N_{f} is the number of frequency channels, and n_{b} is the number of bytes required. We have assumed that incoming data are 8-bit and the output data from the dedispersion is FP32 (4 bytes). Thus, the dedispersion transform will be memory-bound on the most modern hardware platforms. Therefore, data reuse in an available cache must be utilized to increase dedispersion performance.

3. CPU Implementation

Our CPU implementation of the incoherent dedispersion algorithm is written in the C programming language with OpenMP and Cilk Plus, where OpenMP is used for parallelization across cores on multi-core CPUs and Cilk Plus is used to express fine-grained parallelism and allows the compiler to effectively vectorize parts of the code.

Algorithm 2. Pseudocode of the parallel CPU direct dedispersion transform.

Data: $x(f, t)$
Result: DM
#pragma omp parallel for collapse(2)
for $i = 0$ to N_{t}; by D_{t} **do**
 for $jj = 0$ to N_{DM} by D_{dm} **do**
 for $kk = 0$ to N_{f} by D_{f} **do**
 for $j = jj$ to $(jj + \text{D}_{dm})$; j++ **do**
 int $S[\text{D}_{\text{t}}]$;
 if kk $= 0$ **then**
 $S[:] = 0$; **else**
 $S[:] = DM[j \times N_{\text{t}} + i: \text{D}_{\text{t}}]$;
 end
 for $k = kk$ to $(kk + \text{D}_{\text{f}})$; k++ **do**
 $h{=}\Delta t(j, i, k)$; $S[:] \mathrel{+}= x[k \times N_t + i + h: \text{D}_{\text{t}}]$; **end**
 $DM[j \times N_{\text{t}} + i: \text{D}_{\text{t}}] = S[:]$;
 end
 end
 end
end $DM[0: N_{DM} \times N_{\text{t}}]/=$ channels; **return** DM

As discussed in the previous section, dedispersion is a memory bandwidth bound algorithm. Hence, to achieve good performance, careful use of CPU cache is required. As such, our CPU algorithm implements a very well-known optimization technique called loop tiling (see Figure 2), also known as loop blocking, or strip mine and interchange (Wolf &

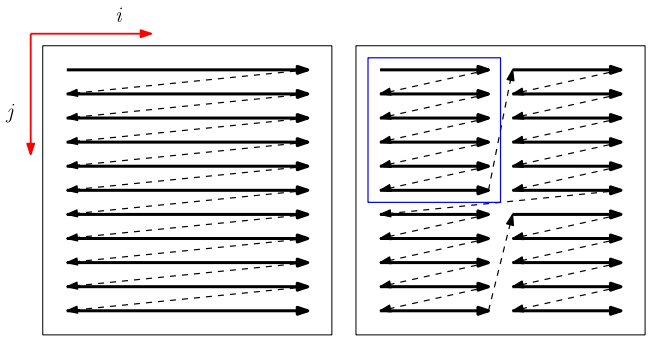

Figure 2. Schematic example of tiling optimization technique in the case of two dimensions (i and j). On the left and on the right, we see the situation when loop blocking is not used and the case when it is, respectively. The original large array is partitioned into smaller blocks (blue rectangle), which can fit into cache size.

Lam 1991), which transforms the input domain (memory) to smaller chunks able to fit into cache, thereby improving the locality of data accesses in loops. This technique also helps reduce the number of cache misses.[7] Specifically, we use "tiling" in frequency channels as well as in dedispersion trials (DMs), where each thread calculates: (1) several DM trials for neighboring time samples (achieving good spatial locality in cache) and (2) several neighboring DM trials (achieving good temporal locality in cache,[8] as a cacheline can be used multiple times to update multiple DM trials). A schematic overview of our partitioning of the data space is shown in Figure 3.

In the pseudocode (see Algorithm 2), the sizes of the tiles are represented by D_f for frequency channels and D_{dm} in the case of the DM. For the code to achieve the best performance, optimal values of D_f, D_{dm} need to be found, as well as the optimal number of time samples per thread (D_t). These optimal values are dependent not only on the CPU used, but also on the input telescope data parameters (like central frequency, number of frequency channels, etc.) and the DM survey plan.

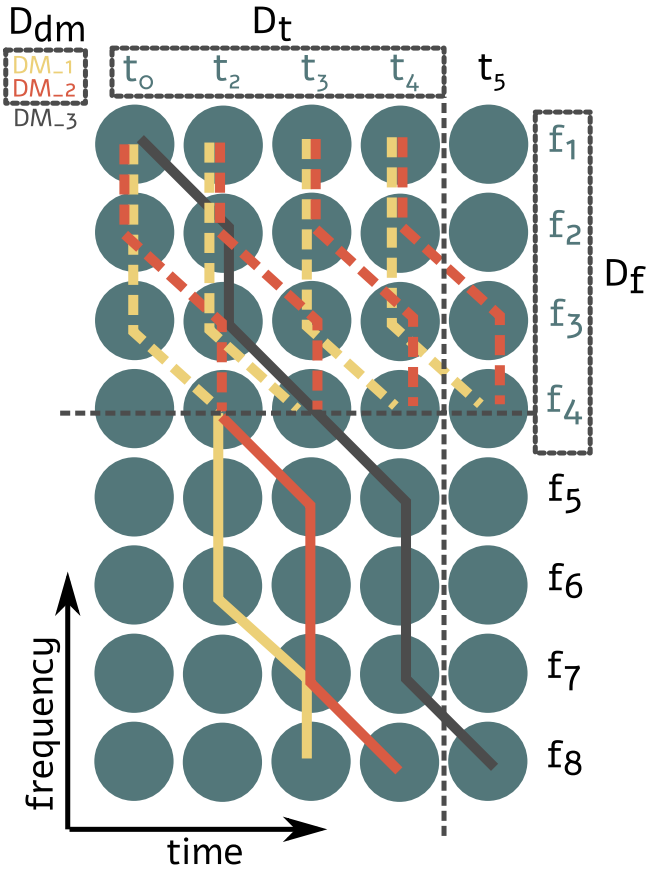

Figure 3. Example of the loop tiling in the case of the CPU dedispersion algorithm. One thread computes a partial sum of D_f frequency channels (f_1–f_4) of D_{dm} for a number of DM trials (DM_1 and DM_2) and for D_t time samples with the same DM trials (t_0–t_4); all of these are represented by the colored dashed lines, where different colors correspond to different DM trials, the x-axis to the time samples, and the y-axis to the frequency channels.

4. GPU Implementation

For a GPU code limited by the memory bandwidth to the GPU main memory, it is essential to reuse data and effectively and efficiently use the L1/L2 cache or the user-managed cache called the shared memory. For peak performance, we need to ensure three things. First, the accumulator that stores the integrated value of the frequency channels S_{loc} must be stored in the fastest area of memory available. Second, the data for each frequency channel that will undergo the dedispersion transform must be readily available to the GPU's streaming multiprocessors; a compute unit analogous to CPU cores. Finally, to avoid costly evaluation of the power law by each thread, the value of the dedispersion shift should be calculated using as few operations as possible.

The advantage of the shared memory over an L1/L2 cache is that the user can control data locality. The shortfall of the shared memory is its size. Where the L1 cache can defer to the larger but slower L2 cache, the shared memory has no such option. Any implementation that uses shared memory and relies on a custom data structure will be limited by its size. This limitation gives rise to the two different algorithms outlined below. In short, the shared-memory version of the direct dedispersion transform can process most shift values with high performance; a cache version, while slower, can handle even large shifts, which are often present at lower central frequencies.

Both of our GPU algorithms are written in the CUDA C/C++ programming language and use the following methodology: One GPU thread processes several time elements for a fixed value of dispersion. A thread stores accumulated values into registers.[9] Nearby values of time and DM in the output DM(dm, t) space are grouped together into thread blocks (Figure 4) such that a single thread block calculates D_t time samples and D_{dm} dispersion steps. The size of the area processed by a single thread block is tunable. Data from the input $x(f, t)$ are read in a coalesced manner, ensuring the best possible performance.

The higher-performing shared-memory version is described below. The cache version will not be described further. In the results section, the cache version is only used where we cannot use the shared-memory version of our code.

[7] When working with caches, it is important to optimize for cache hits. In principle, a cache hit occurs when the cache contains data needed by a thread; otherwise, a cache miss is generated, meaning that data must be reloaded from main memory, thus reducing performance.

[8] The amount of reuse is dependent on the closeness of neighboring DM trials.

[9] Registers are the fastest area of GPU memory. However, increasing the number of registers that a GPU kernel uses reduces the number of resident thread blocks that can occupy each SM.

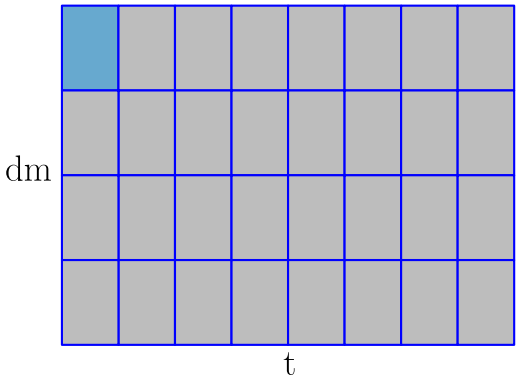

Figure 4. The output (*dm, t*) plane is partitioned into sections of size ($D_{\rm dm}$, $D_{\rm t}$) that are each processed by a single thread block, shown in blue.

4.1. Shared-memory Algorithm

The pseudocode for the GPU kernel is presented in Algorithm 3. In the shared-memory GPU implementation of the dedispersion algorithm, each thread block calculates a different part of the output DM(*dm*, *t*) space, as shown in Figure 4. A single thread from a thread block loops over frequency channels with steps of $D_{\rm ch}$ channels, where it loads a single element of the $x(f, t)$ data into a local buffer $B_{\rm loc}$ using the shift

$$\Delta t_{\rm bl} = T_{\rm bl}\,\Delta t_{\rm pc}, \tag{6}$$

where the coefficient $T_{\rm bl}$ in Equation (6) represents the lowest DM calculated by the thread block, and $\Delta t_{\rm pc}$ are the coefficients of the cold plasma dispersion law for each frequency channel. All threads in the thread block thus form a contiguous (in time) block of $x(f, t)$ data in shared memory. This is shown in Figure 5.

When local buffer $B_{\rm loc}$ is loaded, each thread sums appropriate elements in shared memory using the differential shift $\Delta t_{\rm diff} = T_{\rm diff}\Delta t_{\rm pc}$ and updates their value of the partial sum $S_{\rm loc}$, held in registers, that in the end will result in the dedispersed value expressed by Equation (3). The coefficient $T_{\rm diff}$ represents DM values calculated by different threads within the range of DM values calculated by a thread block ($D_{\rm dm}$). After this, threads integrate the loop over frequency channels and a new block of $x(f, t)$ data is loaded.

To avoid costly evaluation of the power law by each thread, the calculation of the dispersion shift Δt was split into two parts. The first part is an array of precalculated coefficients of the cold plasma dispersion law $\Delta t_{\rm pc}$, which are evaluated as

$$\Delta t_{\rm pc}(i) = 4148.74\left(\frac{1}{(f_{\rm high} - \Delta f \cdot i)^2} - \frac{1}{f_{\rm high}^2}\right), \tag{7}$$

where $\Delta t_{\rm pc}(i)$ represents the time shift for the ith frequency channel, $f_{\rm high}$ is the highest frequency of the telescope bandwidth, and Δf is the bandwidth of a single frequency channel. This array is then scaled in the thread block by the DM value that is being evaluated. The second part required for calculation of Δt consists of the two DM coefficients: $T_{\rm bl}$, which is constant within the thread block; and $T_{\rm diff}$, which relates to the DM calculated by a thread (see Algorithm 3).

Further performance improvements can be achieved by processing multiple time samples $N_{\rm REG}$ per thread. This exploits instruction-level parallelism, where a thread can process multiple independent instructions. The adverse effect is that processing too many time samples per thread increases register usage too much, leading to fewer active threads and lower performance.

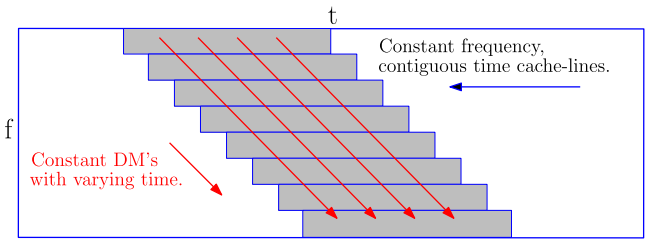

Figure 5. Data are loaded from cache lines of constant frequency and contiguous time (gray boxes). Multiple dedispersion trials DM = constant, *t* values (red lines) are held in registers, and each thread updates a set of these in parallel.

When working with input data of 8 bits or less, we pack the data into 32-bit words and then use integer addition to achieve SIMD (single instruction, multiple data) in word. In the case of 8-bit data, this allows us to process two time samples per operation, increasing throughput significantly.

The values of $N_{\rm REG}$, $D_{\rm t}$, $D_{\rm dm}$, and $D_{\rm ch}$ have a significant impact on the performance of this code and must be tuned for a given telescope configuration and DM plan in order to gain maximum performance.

Algorithm 3. Pseudocode (GPU kernel) for the shared-memory-based GPU algorithm.

Data: $x(f, t)$, $\mathrm{DM}_{\rm start}$, $\mathrm{DM}_{\rm step}$, $\Delta t_{\rm pc}$
Result: DM(dm, t)
Initiate local accumulator;
$S_{\rm loc}[N_{\rm REG}] = 0$;
Shared memory **buffer** *to store local copy of x(f,t)*;
`__shared__` $B_{\rm loc}(f, t)$;
Time shift depends on position in DM(dm, t) *plane*;
$T_{\rm diff}$ = `ThreadIdx.y` × $\mathrm{DM}_{\rm step}$;
$T_{\rm bl} = \mathrm{DM}_{\rm start}$ + `BlockIdx.y` × D_{dm} × $\mathrm{DM}_{\rm step}$;
for $c = 0$ **to** N_{ch} **by** $D_{\rm ch}$ **do**
Data segment is stored into shared memory;
$B_{\rm loc}(f, t) = x(f, t + T_{\rm bl}\Delta t_{\rm pc}(c))$;
`__synchthreads();`
for $l = 0$ **to** $D_{\rm ch}$ **do**
Dedisperse local data into accumulators;
$S_{\rm loc}$= `Dedispers` $(B_{\rm loc}(f, t + T_{\rm diff}\Delta t_{\rm pc}(l)))$;
end
end
Store local results into output DM(dm, t);
DM(dm, t) = $S_{\rm loc}$;
return DM(dm, t);

5. Results

In this section, we first explore how the performance of our GPU dedispersion code, which is part of the Astro-Accelerate package, depends on the parameters of input data and the dedispersion plan. To compare our dedispersion to other implementations, we have considered two test cases. The first test measures the execution time of the direct dedispersion transform for a varying number of frequency channels using a fixed dedispersion plan (Section 5.2). The

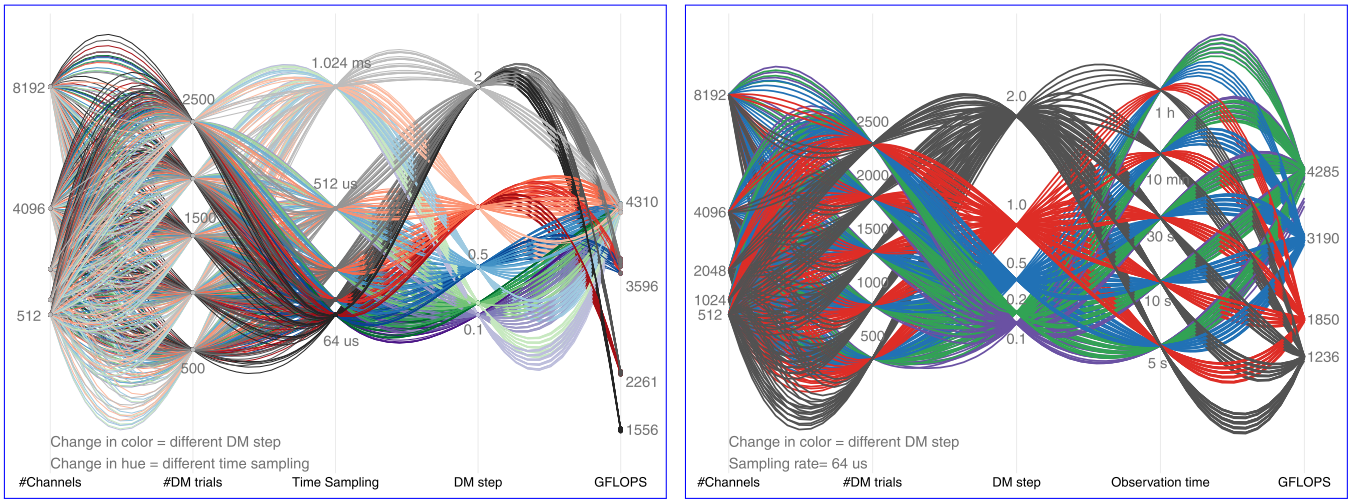

Figure 6. Flow plot representing how AA performance (GFLOPS) depends on parameters of the data and dedispersion plan. Each line symbolizes one run of the autotuned AA GPU dedispersion, with data and a dedispersion plan described by parameters shown in the figure. These parameters are the number of channels, DM trials, DM step, sampling time of the data, and observation time. For simplicity, only the 8-bit input data and a telescope central frequency set to 1400 MHz are shown. The left plot, where we have fixed the observation time and emphasized the DM step (color) and sampling time (hue), shows that the performance is mainly affected by the combination of two parameters: the DM step and the sampling time. Lines of rich dark color, where high sampling time is combined with a large DM step, show the lowest-performing case, where shared-memory GPU code cannot be used. Lines of brighter color have increasing performance as the sampling time gets lower. The right plot, where the sampling time is fixed and only the DM step is emphasized, shows that the performance is independent of the observation time.

second test, described in Section 5.3, demonstrates the performance of our dedispersion implementation when used in a data-processing pipeline. For this test, we have used three data sets with different central frequencies and dedispersion plans. We compare the output from the different implementations tested in Section 5.4. Finally, we demonstrate AstroAccelerate performance on selected radio telescopes in Section 5.5.

We compare our results with currently existing and in-use implementations of direct dedispersion suited for HPC environments. Specifically, for the first test and the numerical difference test, we use the codes "Dedisp" (Barsdell et al. 2012) and "Dedispersion" (Sclocco et al. 2016). Due to their similar names, we will refer to these dedispersion implementations by the names of their associated processing pipelines. The Heimdall[10] pipeline is a GPU-accelerated transient detection pipeline that utilizes "Dedisp," while "Dedispersion" is used in the Amber pipeline (Sclocco et al. 2020).

The Amber dedispersion code is written using the Open Computing Language (OpenCL) programming language, while Heimdall's Dedisp uses the CUDA C/C++ programming model. Both implementations can run in two modes /regimes. Heimdall, in "adaptive" mode, changes the DM step during dedispersion depending on the parameter of DM tolerance set by the user. The number of DM trials thus varies. The second mode of Heimdall can be described as "fixed," meaning that the user selects a fixed step size in the DM range. Thus, the pipeline outputs a fixed number of DM trials. In the case of Amber, for single DM, the frequency channels are divided into sub-bands, which are first dedispersed ("step one") and then dedispersed within each sub-band ("step two"). Our code performs a fixed number of DM trials with a fixed DM step for each DM range, akin to the "fixed" mode in Heimdall and one of the modes in Amber.

[10] https://sourceforge.net/projects/heimdall-astro/

Table 2
Dedispersion Plans Used in the Frequency Resolution Test (Section 5.2)

DM Range (pc cm^{-3})	DM Step (pc cm^{-3})	No. of DM Trials
0–150	0.10	1500
150–300	0.20	750
300–500	0.25	800

In all following tests, the input signal is generated using "fake" from the pulsar processing package SIGPROC. We generated synthetic filterbank files for 4-bit, 8-bit, and 16-bit precisions.

We have used two GPU cards (NVIDIA Tesla V100 –Volta generation and NVIDIA A100 –Ampere generation), two Intel processors (Xeon Phi 7290 –Knights Landing (KNL) and Xeon Gold 6230 –Cascade Lake), and an AMD processor EPYC 7601 –Naples. Their hardware specifications can be found in the Table 1, where the GPU shared memory bandwidth is calculated as

$$\begin{aligned}\mathrm{BW(bytes\ s^{-1})} = &(\text{bank bandwidth (bytes)})\\ &\times(\text{clock frequency (Hz)})\\ &\times(32\ \text{banks})\ \times\ (\text{No. multiprocessors}),\end{aligned} \tag{8}$$

and the theoretical peak performance of the CPUs by

$$\begin{aligned}\text{Peak (GFLOPS)} = &(\text{clock frequency (GHz)})\\ &\times(\text{No. of cores})\ \times\ (\text{vector width})\\ &\times(\text{instructions/cycle})\ \times\ (\text{FLOPS/instruction}),\end{aligned} \tag{9}$$

where FLOPS/instruction = 2 in the case of fused multiply add (FMA), and the vector width is 16 for single precision and 8 for double precision. The Intel CPUs (Xeon Gold 6230 and Xeon Phi 7290) have two AVX-512 units, and thus instructions/cycle = 2, while the AMD EPYC 7601 has instructions/

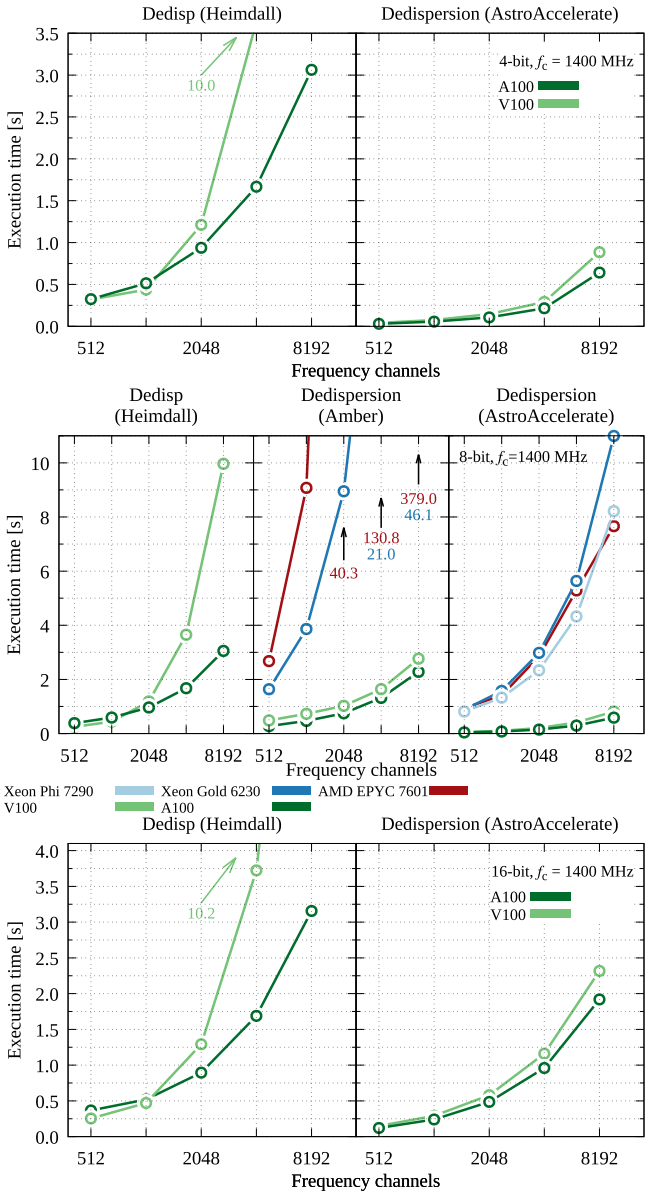

Figure 7. The execution time of the corresponding dedispersion plan (see Table 2) for 4-bit, 8-bit, and 16-bit precision (from top to bottom) input data with an increasing number of channels, observation time $T = 10\ s$, central frequency $f_c = 1400$ MHz, and total bandwidth of 300 MHz.

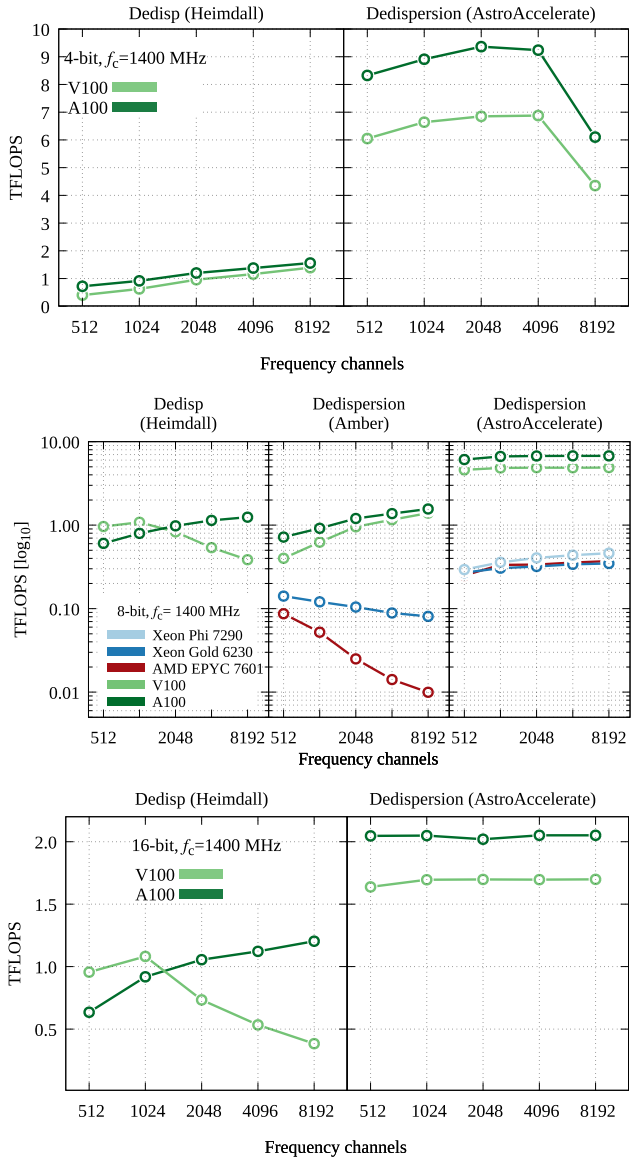

Figure 8. Performance in FLOPS with an increasing number of channels for all tested codes. The first three rows correspond to 4-bit, 8-bit, and 16-bit precision.

cycle = 1. In the case when all cores utilize AVX-512 instructions, the core clock frequency is reduced by 100–200 GHz.[11]

For the time measurements of Heimdall and AstroAccelerate, we used the NVIDIA Compute profiler software (ncu) for the GPUs and the `omp_get_wtime` function for the CPU case, while in the case of Amber, we followed the supplied timer using OpenCL functions. Where possible, we use the supplied autotuning scripts for each test and all codes to achieve the best performance. Unless otherwise stated, all execution times show kernel runtime only; that is, no data transfers from host to device (GPU) are taken into account. The GPU codes are compiled with `nvcc` compiler, and the CPU codes with the Intel compiler (`ICC`). Codes based on OpenCL are compiled with `nvcc` for GPUs and `ICC` for CPUs, with the appropriate OpenCL flag enabled. The compiler flags we used are summarized in Table 1.

5.1. Performance Dependency

To illustrate the dependency of AstroAccelerate's performance on different input data parameters, we have visualized all distinct cases in the form of a flow plot shown in Figure 6. The parameters used are the number of frequency channels, sampling time of the data, observation time, the number of DM trials, and the size of the DM step. The performance is expressed in GFLOPS, as this can be understood as an average performance per second. The execution time will still depend on the size of the task. The left plot in Figure 6 shows that the right combination of time sampling with DM step size is essential for high performance. The plot on the right of Figure 6

[11] See the Intel Xeon Phi Processor product brief (https://objects.icecat.biz/objects/mmo_32741035_1471257902_943_16737.pdf), and the Intel Xeon Processor Scalable Family Specification Update (https://www.intel.com/content/www/us/en/content-details/613537/intel-xeon-processor-scalable-family-specification-update.html).

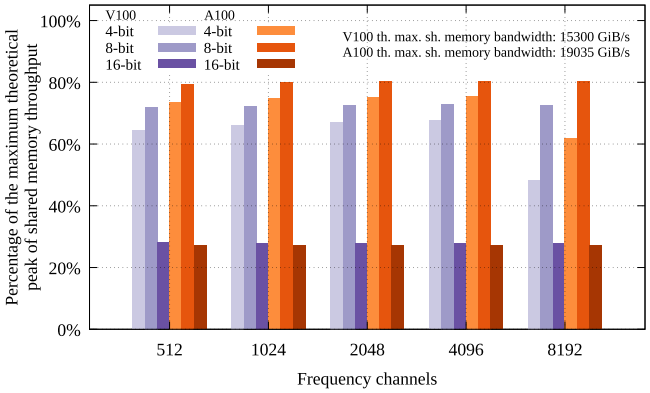

Figure 9. Shared memory bandwidth, shown as a percentage of the theoretical maximum achieved by AstroAccelerate dedispersion for different bit precisions and an increasing number of channels. The theoretical maximum of shared memory bandwidth for each GPU card is in the top right corner.

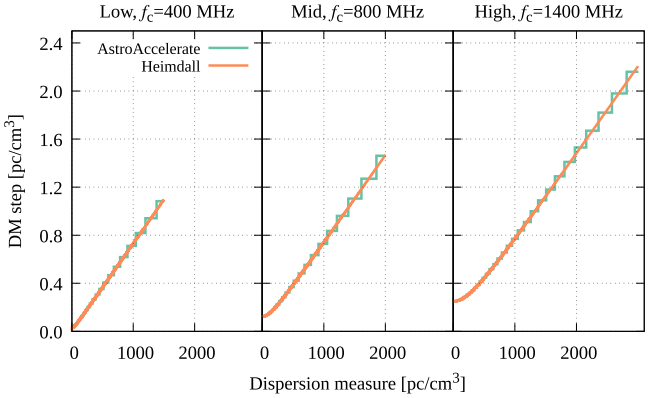

Figure 10. Visualization of the continuous Heimdall survey DM plan with a discrete AstroAccelerate DM plan for all three cases: $f_c = 400$, 800, and 1400 MHz (from left to right). Details can be found in Table 3.

Table 3
Specifications of the Input Data Used for Pipeline Comparison

	Central Frequency	Total Bandwidth	Sampling Rate	No. of Channels	DM Range
Low	400 MHz	200 MHz	256 μs	1024	0–1500
Mid	800 MHz	200 MHz	128 μs	4096	0–2000
High	1400 MHz	300 MHz	64 μs	4096	0–3000

Note. The observation length corresponds ∼300 s for the low central frequency and ∼50 s for medium and high central frequencies.

shows that the performance of the AA dedispersion code does not depend on the observation time, as in all other tested codes.

5.2. Frequency Resolution Test

The following test shows the behavior of the execution time of all mentioned codes when we change the number of channels.

For testing, we created a synthetic signal with 4-bit, 8-bit, and 16-bit samples with a time sampling of 64μs for five different channelizations: 512, 1024, 2048, 4096, and 8192, where the last one corresponds to the maximum number of channels that Heimdall can safely process, due to integer overflow. The length of the input signal corresponds to 10 s of observation data with a central frequency 1400 MHz and a total bandwidth of 300 MHz. This observation length is sufficient to get representative performance measurements. Moreover, it allows us to also extend our testing up to 8192 channels for the Heimdall dedispersion code, which cannot be run at this number of channels for longer observation times. For the survey plan, we use three DM ranges without time binning (also known as downsampling), which together search for signals ranging from DM 0 to 500 pc cm^{-3}. We limited the search to 500 pc cm^{-3} because at high DM it is common to use a downsampling/scrunch factor, which we do not include in this test. Moreover, the high-DM searches are covered in the second test. The dedispersion plan used is summarized in Table 2.

The execution times of the dedispersion plan using differing numbers of frequency channels and bit precisions for all tested codes are shown in Figure 7. The results of Amber dedispersion for the 4-bit and 16-bit are missing because the code did not return the correct results for the injected signal from SIGPROC. There are also no results for our CPU implementation because it only supports 8-bit precision. Although the OpenCL parallel language can be used across platforms, Intel officially does not support KNL. Therefore, we do not show execution times for Amber.[12] Figure 8 shows how performance, expressed in the floating point operations per second (FLOPS), changes with increasing numbers of channels.

By analyzing our implementation of the incoherent dedispersion on the GPU using the NVIDIA Nsight Compute, we see that, in the cases of 4-bit and 8-bit precision, our implementation is limited by the shared memory bandwidth on both GPU cards, while the 16-bit precision version is limited by the special function units (≈95% of utilization) also on both GPU cards. The special function units take care of type conversions that are required by the 16-bit implementation. The summary of shared memory bandwidth utilization by AstroAccelerate for each card and bit precision is presented in Figure 9.

5.3. Processing Pipelines

In this section, we analyze the execution time of the AstroAccelerate running the dedispersion plan with different DM ranges (0–3000 pc cm^{-3}) and time binning factors (also known as downsampling/scrunch factors). We compare the results alongside the GPU-accelerated pipeline—Heimdall. We have not compared the Amber pipeline, as this was compared to Heimdall by Sclocco et al. (2020).

As the radio telescopes operate on a wide range of central frequencies, we selected three scenarios to demonstrate the performance and behavior of both AstroAccelerate and Heimdall. The selected central frequencies are $f_c = 400$ (low), $f_c = 800$ (mid), and $f_c = 1400$ (high), each with the typical bandwidth, sampling rate, number of channels, and DM survey plans (for details see Table 3). The synthetic input data were generated as in the previous cases using SIGPROC "fake." The observation lengths correspond to ∼300 s for the low central frequency case, and ∼50 s for the others.

To compare both pipelines fairly and use all the implemented features, we must ensure they use the same or a comparable

[12] Even though it is still possible to get OpenCL code running on KNL by using an older Intel OpenCL driver without the support of AVX-512 (Johnston & Milthorpe 2018), i.e., it provides only half of the theoretical peak performance.

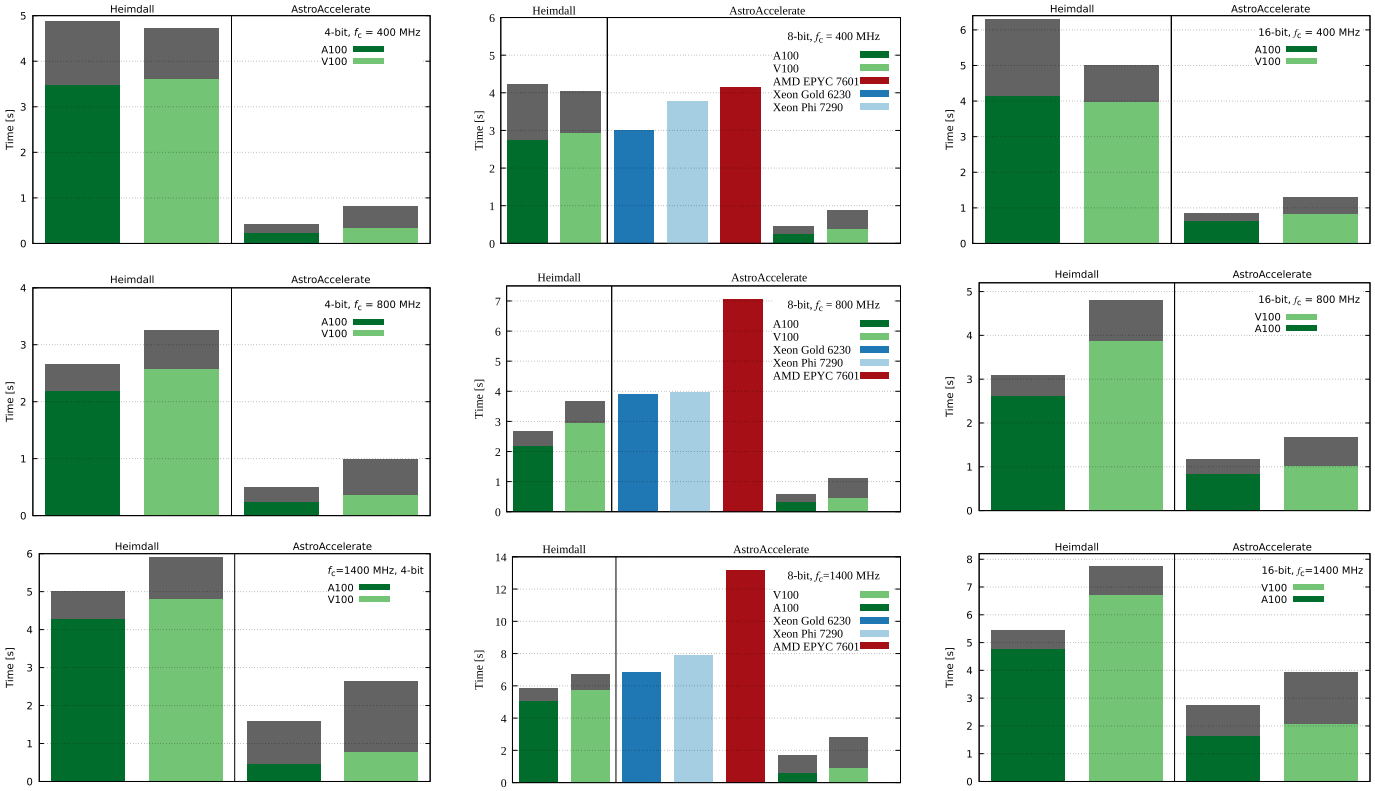

Figure 11. Execution times of AstroAccelerate and Heimdall on different many-core platforms for three central frequencies and bit precisions operating on a simulated signal of an observation length of ∼300 s for the low central frequency and ∼50 s for all other central frequencies. The top row corresponds to the central frequency $f_c = 400$, the middle row to $f_c = 800$, and the bottom row to $f_c = 1400$ MHz (low, mid, and high). In the left column are the results for 4-bit precision, in the middle column those for 8-bit precision, and in the right column those for 16-bit precision. The gray boxes show the execution time including all PCIe transfers and overheads needed to finish the plan. The color (lime green or dark green) bar shows the execution time of all kernels, for example, transposing of the input data, downsampling data, etc., needed by the pipeline.

DM plan. Heimdall, by default, uses an "adaptive" mode. That is, Heimdall generates a list of DMs during the startup phase of the pipeline based on the smearing tolerance factor between individual DM trials (default value 1.25), thereby changing its DM step for each DM trial. AstroAccelerate, on the other hand, uses a user-defined fixed DM step for each DM range. To obtain a similar DM plan for AstroAccelerate, we used the "continuous" plan from Heimdall and created the closest "discrete" plan for our pipeline. A visualization of these plans for all three cases is shown in Figure 10. We note that the highest DM is different for each case (∼1500, 2000, and 3000 pc cm^{-3}).

The execution times for each bit precision and many-core system (GPU or CPU) are summarized in Figure 11. Similarly, as in the previous subsection, we found that our implementation is limited by the shared memory bandwidth (4-bit and 8-bit) or by special functional units (16-bit).

5.4. Comparison of the Dedispersed Planes

The dedispersion output plane normalized to its own maximum from AstroAccelerate, Heimdall, and Amber are presented in Figure 12. We used synthetic 8-bit data with an injected signal of DM = 90.0 pc cm^{-3}. Each code was run in "fixed" mode with a simple survey plan searching pulses between DMs of 0–200 pc cm^{-3} with a step 0.5 pc cm^{-3}. As shown in Figure 12, all implementations recover the correct dispersion measure of the injected signal. However, as shown in Figure 13, there are small numerical differences (∼1%). These are introduced by the differences in the calculation of the time shift function, including the use of a slightly different constant of proportionality ($C_{\rm DM}$) used in Equation (1) in the different implementations. In addition, Heimdall dedispersion internally rescales the output dedispersed values, thus incurring a round-off error. Finally, in Figure 14, we provide a view of a single DM trial from the dedispersed outputs between AstroAccelerate, Heimdall, Amber, and our CPU implementation around the DM of the injected signal and their percentage difference.

5.5. Performance on Selected Telescopes

In this section, we provide performance results for AstroAccelerate for 10 selected telescopes and their settings for two GPUs, namely the Tesla V100 GPU and A100 GPU. We have determined the DM survey plan for each telescope setup using the `DDplan.py` tool from the PRESTO pulsar search and analysis software (Ransom 2011). We measure the performance in units of fractions of real time computed as

$$R = \frac{t_{\rm obs}}{t_{\rm c}}, \tag{10}$$

where $t_{\rm obs}$ is the observation time that is being processed and $t_{\rm c}$ is the execution time of the AstroAccelerate pipeline. The execution time $t_{\rm c}$ includes the time required for dedispersion, the transpose of the input data (if necessary), and downsampling (time binning). We do not include the input data transfer time from the host to the GPU device memory. It

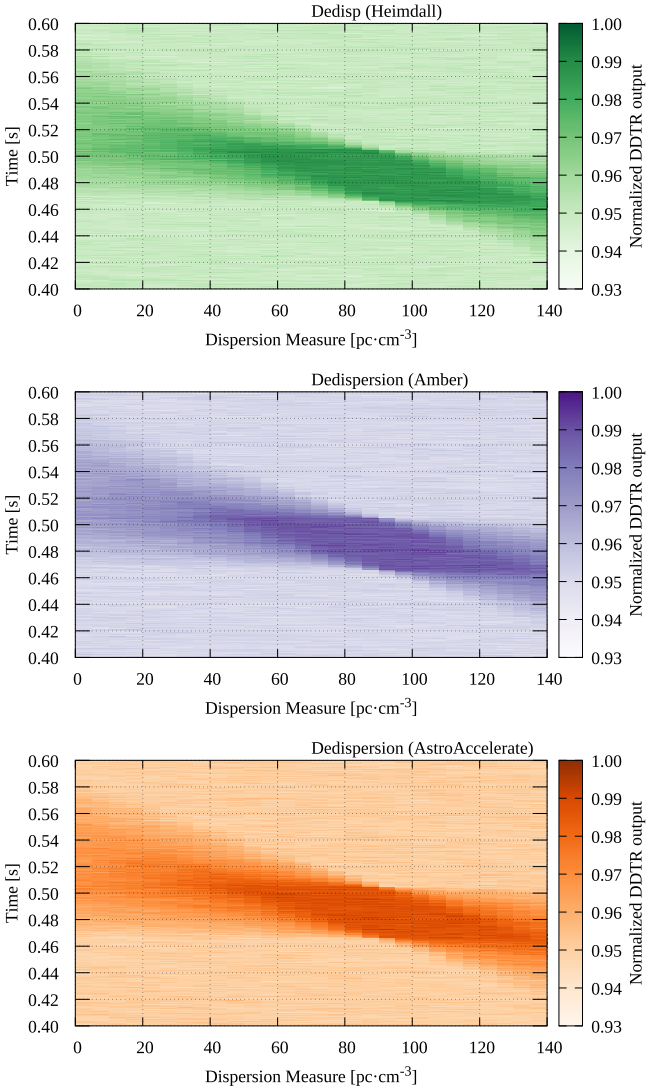

Figure 12. Zoomed-in normalized output of the dedispersion transform (DDTR) for Heimdall (top, green), Amber (middle, purple), and AstroAccelerate (bottom, orange) of an injected signal with DM = 90.0 pc cm^{-3}.

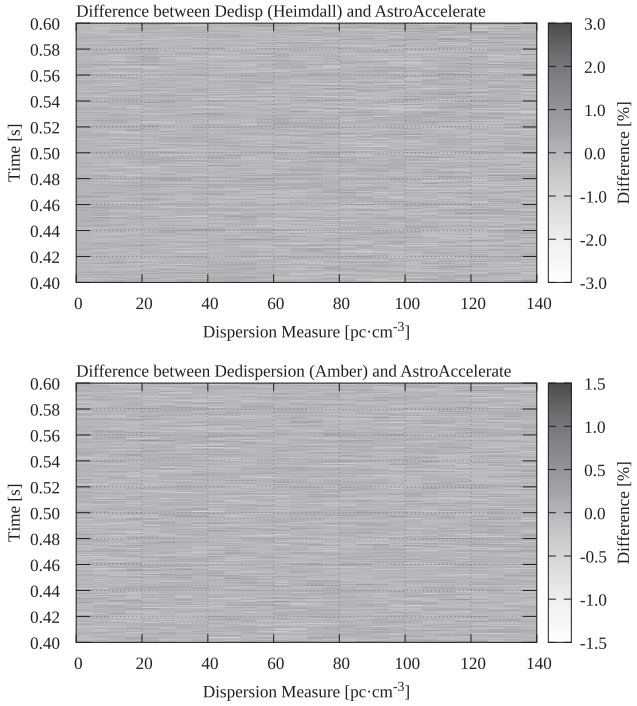

Figure 13. Percentage difference of the normalized dedispersion outputs. From top to bottom: AstroAccelerate and Heimdall; AstroAccelerate and Amber.

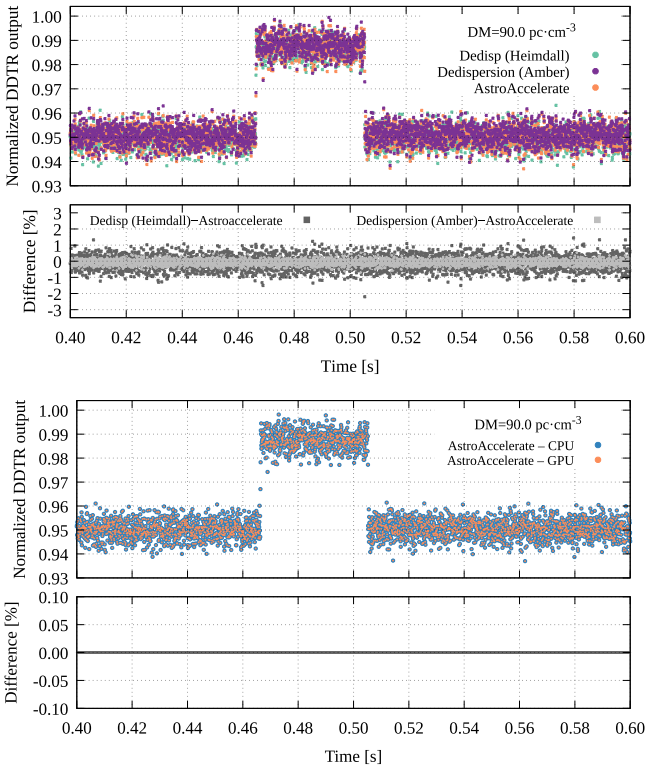

Figure 14. Comparison of the dedispersed (DDTR) outputs and their percentage difference at DM = 90 pc cm^{-3} between AstroAccelerate, Heimdall, and Amber at the top, and our CPU implementation at the bottom.

should be noted that $R > 1$ means that the computing is performed in real time or greater, that is, the observation time is longer than the time needed for its processing.

Table 4 summarizes the basic characteristics of the selected telescopes, the range of the DM survey plan, and the performance of the autotuned AstroAccelerate software pipeline on NVIDIA Tesla V100 and NVIDIA A100 GPU in units of fractions of real time. As can be seen, in all cases, AstroAccelerate operates in real time or greater. In the worst-case scenario, AstroAccelerate achieved $R = 20$ on the NVIDIA Tesla V100 GPU and $R = 25$ on NVIDIA A100 GPU. For more details, such as the DM survey plans for individual telescopes, memory transfers, CPU performance tests, etc., please see https://github.com/AstroAccelerateOrg/SupportingMaterialForPapers/tree/main/Dedispersion.

6. Discussion and Conclusions

In this paper, we present our CPU and GPU implementations of the incoherent dedispersion method for removing the effect of the frequency delay introduced due to the interstellar medium. Although dedispersion is only a part of the pulsar search process, its computational intensity scales rapidly with the amount of data, and as such, it becomes a substantial contribution to the total processing time of the pipeline. We compare three different many-core implementations of the incoherent dedispersion transform, namely Dedisp by Barsdell

Table 4
List of Selected Radio Telescopes, Their Characteristics, and Their Search Ranges, and the Performance of the AstroAccelerate in Fractions of Real Time

Telescope		Central Frequency (MHz)	Bandwidth (MHz)	Time Sampling (μs)	Channels	DM Range (pc cm^{-3})	Fractions of Real Time	
							Tesla V100	A100
Apertif[a]	ALERT	1400	300	40.92	1536	0–10,000	28	41
Arecibo[b]	PALFA	1375	322	65.5	1024	0–9866	75	91
ASKAP[c]		1400	336	1265	336	0–3763	4613	5835
CHIME[d]	FRB	600	400	1000	16384	0–2000	20	25
Green Bank Telescope[e]		820	200	20.48	512	0–2000	69	110
		2000	800 (600)	10.24	512	0–1000	59	82
GMRT[f]		400	200	1310.72	4096	0–2000	187	239
Lovell[g]		1532	400	256	800	0–10,000	622	837
Parkes—SUPERB[h]	F-pipeline	1382	400	64	1024	0–2000	98	133
	T-pipeline	1382	400	64	1024	0–10,000	89	127
UTMOST[i]		835.5	31.25	655.36	320	0–10,000	7940	8930
VLA[j]		3000	1024	5000	256	0–10,000	254,200	327,665

Notes.
[a] Maan & van Leeuwen (2017).
[b] Spitler et al. (2014); Scholz et al. (2016).
[c] Bannister et al. (2017).
[d] Amiri et al. (2018); Mikhailov & Sclocco (2018).
[e] Masui et al. (2015); Scholz et al. (2016).
[f] Bhattacharyya (2017); Bhattacharyya et al. (2019); Singh et al. (2022).
[g] Scholz et al. (2016).
[h] Keane et al. (2017).
[i] Bailes et al. (2017); Caleb et al. (2017).
[j] Law et al. (2017).

et al. (2012) that is part of the Heimdall pipeline, Dedispersion by Sclocco et al. (2016) from the Amber pipeline, and our implementation that is part of the AstroAccelerate project. We demonstrate that our implementation of dedispersion is faster and covers a wider range of different input data parameters.

In Figure 6, we show how the performance of AstroAccelerate depends on different input data parameters. The most important parameter for performance is the combination of sampling time and DM step. For a large enough DM step, the required data may no longer fit into the GPU's shared memory, and the cache dedispersion algorithm that relies on L1 or a slower L2 cache must be used. This leads to a performance loss of up to 4×. In such a case, it might be beneficial to lower the data sampling rate—for example, through time binning—and benefit from higher performance due to the smaller data size and being in a more cache-friendly regime.

In the first benchmark in Section 5.2, we compare all three implementations (Heimdall, Amber, and our AstroAccelerate), processing input data of 4-bit, 8-bit, and 16-bit precision, and producing a fixed number of DM trials but using different numbers of frequency channels. We demonstrate that the AstroAccelerate GPU implementation is at least 10× faster in the case of Heimdall and from 6× (with 512 channels) to 3.4× (8192 channels) faster than Amber on the NVIDIA Ampere generation A100 GPU. Similar results apply for the previous-generation card, the NVIDIA Tesla V100 GPU. The worst performance compared to Heimdall is for 16-bit precision, where AstroAccelerate is only 2× faster.

Our CPU version of the incoherent dedispersion on all tested CPUs is comparable in performance to the Heimdall code on an NVIDIA Tesla V100 GPU for a higher number of frequency channels. Compared to the CPU version of Amber, our code is from 2× (512 channels) to 15× (8192 channels) faster for all tested CPUs.

Figure 8 shows that our implementations achieve, on average, stable performance for all tested frequency channels in terms of FLOPS. That is, except for the case of 8192 channels for 4-bit input data, due to the algorithm change. Our implementation achieves an average of ∼6.5 TFLOPS on a Tesla V100 GPU for 4-bit, ∼4.5 TFLOPS for 8-bit precision data, and ∼1.6 TFLOPS for 16-bit input data. On a Tesla A100, it achieves ∼8.5 TFLOPS for 4-bit precision data, ∼5 TFLOPS for 8-bit, and ∼2 GFLOPS in the case of 16-bit. The performance improvement of the Ampere generation compared to the Volta generation is mainly due to the increased shared memory bandwidth (from ∼ 14 TB s^{-1} to ∼ 18 TB s^{-1}. On the tested CPUs, we get ∼0.4 TFLOPS on KNL, ∼0.33 TFLOPS in the case of the AMD EPYC 7601, and ∼0.31 TFLOPS for an Intel Xeon Gold 6230.

The performance of other tested codes is mostly stable or improves with an increasing number of channels. However, Heimdall has a particular problem with the NVIDIA Tesla V100 GPU, as the performance decreases significantly for a high number of frequency channels, something that is not observed with the same code on the NVIDIA A100 GPU. We have observed an unusual behavior of the Amber pipeline on the AMD CPU, where the performance decreases significantly for a higher number of frequency channels. This contradicts our expectations based on Amber's performance on the Intel CPU and the NVIDIA GPUs. This may indicate the use of platform-specific optimization in Amber or the lack of OpenCL support for AMD EPYC CPUs. It also shows that, even though the OpenCL programming model is easily portable to different many-core systems, it does not always guarantee good performance.

In Section 5.3, we perform a pipeline test for three different central frequencies ($f_c = 400$ (low), $f_c = 800$ (mid), and $f_c = 1400$ (high)), with searches running from 0 up to 1500, 2000, and 3000 pc cm^{-3}, respectively, with the sampling rates and the numbers of channels depending on the central frequencies. We execute all benchmarks for AstroAccelerate and Heimdall for 4-bit, 8-bit, and 16-bit precision input data (where applicable) on the GPUs as well as on the CPUs. We find that both AstroAccelerate and Heimdall on all tested platforms operate in real-time regimes. That is, the end-to-end execution time of a pipeline (which includes all required operations like time binning, data transfers to the GPU memory, dedispersion, etc.) for the selected dedispersion plan is lower than the observation time of the input data. Overall, our GPU version, compared to Heimdall, is 4–8× faster for all tested central frequencies and input data precisions.

The performance of our CPU implementation is comparable with Heimdall running on either an NVIDIA V100 GPU or an NVIDIA A100 GPU. Only at mid-central frequencies is Heimdall substantially faster. Depending on the structure of the pipeline, the CPU dedispersion code may offer a way to distribute the tasks between the CPU and GPU, where the GPU can, for example, perform FDAS (Ransom et al. 2002; Dimoudi et al. 2018) or JERK search (Andersen & Ransom 2018; Adámek et al. 2020) while the CPU calculates DM trials, thus enabling heterogeneous systems.

Finally, we run AstroAccelerate on several telescope setups with different survey plans. The plans are created with the `DDplan.py` tool from PRESTO up to the typical dispersion measures ranges given in the corresponding telescope articles. On both the NVIDIA V100 GPU and NVIDIA A100 GPU, we achieve real-time performance in all cases, i.e., from 20 to 5000 fractions of real time and even 200,000 fractions of real time for the VLA telescope.

Acknowledgments

The authors would like to thank Aris Karastergiou, Steve Roberts, Ben Stappers, Mitch Mickaliger, Duncan Lorimer, Maciej Serylak, Evan Keen, and Cees Bassa for useful comments, advice, and help with testing. The project has also received support from the members of the Oxford pulsar group, Christopher Williams, Griffin Foster, and Jayanth Chennamangalam, as well as support from the Time Domain Team, a collaboration between Oxford, Manchester, and MPIfR Bonn, to design and build the SKA pulsar search capabilities. The authors would like to express their gratitude to the Research Centre for Theoretical Physics and Astrophysics, Institute of Physics, Silesian University in Opava for institutional support.

This work has been supported by the Institute for the Future of Computing, Oxford Martin School and STFC grants ST/N003713/1, ST/P005446/1, ST/R000557/1, and ST/W001969/1, and Silesian University in Opava grant IGS/11/2022. Also, the authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (Richards 2015) facility in carrying out this work.

ORCID iDs

Jan Novotný https://orcid.org/0000-0002-9667-635X
Karel Adámek https://orcid.org/0000-0003-2797-0595
M. A. Clark https://orcid.org/0000-0001-5211-2002
Mike Giles https://orcid.org/0000-0002-5445-3721
Wes Armour https://orcid.org/0000-0003-1756-3064

References

Adámek, K., & Armour, W. 2020, ApJS, 247, 56
Adámek, K., Novotný, J., Dimoudi, W. S., & Armour, S. 2020, in ASP Conf. Ser. 527, Astronomical Data Analysis Software and Systems XXIX, ed. R. Pizzo, E. R. Deul, E. R. Mol et al. (San Francisco, CA: ASP), 671
Adámek, K., Roy, J., & Armour, W. 2022, A&C, 40, 100621
Agarwal, D., Lorimer, D. R., Surnis, M. P., et al. 2020, MNRAS, 497, 352
Amiri, M., Bandura, K., Berger, P., et al. 2018, ApJ, 863, 48
Andersen, B. C., & Ransom, S. M. 2018, ApJL, 863, L13
Armour, W., Adámek, K., Novotný, J., et al. 2020, AstroAccelerate, v1.8.1, Zenodo, doi:10.5281/zenodo.4282748
Armour, W., Karastergiou, A., Giles, M., et al. 2012, in ASP Conf. Proc. 461, Astronomical Data Analysis Software and Systems XXI, ed. P. Ballester, D. Egret, & N. P. F. Lorente (San Francisco, CA: ASP), 33
Bailes, M., Jameson, A., Flynn, C., et al. 2017, PASA, 34, e045
Bannister, K. W., Shannon, R. M., Macquart, J.-P., et al. 2017, ApJL, 841, L12
Barsdell, B. R., Bailes, M., Barnes, D. G., & Fluke, C. J. 2012, MNRAS, 422, 379
Barsdell, B. R., Barnes, D. G., & Fluke, C. J. 2010, MNRAS, 408, 1936
Bassa, C. G., Pleunis, Z., & Hessels, J. W. T. 2017, A&C, 18, 40
Bassa, C. G., Romein, J. W., Veenboer, B., van der Vlugt, S., & Wijnholds, S. J. 2022, A&A, 657, A46
Bhattacharyya, B. 2017, in IAU Symp. 337, Pulsar Astrophysics the Next Fifty Years, ed. P. Weltevrede, B. B. P. Pereda, L. L. Preston et al. (Cambridge: Cambridge Univ. Press), 17
Bhattacharyya, B., Roy, J., Stappers, B. W., et al. 2019, ApJ, 881, 59
Caleb, M., Flynn, C., Bailes, M., et al. 2017, MNRAS, 468, 3746
Carilli, C., & Rawlings, S. 2004, NewAR, 48, 979
Dimoudi, S., Adamek, K., Thiagaraj, P., et al. 2018, ApJS, 239, 28
Fluke, C. J. 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), 3
Johnston, B., & Milthorpe, J. 2018, in Proc. of the 47th Int. Conf. on Parallel Processing, ed. F. Silla & J. M. Cecilia (New York: ACM), 4
Keane, E. F., Barr, E., Jameson, A., et al. 2017, MNRAS, 473, 116
Kong, X., Zheng, X., Zhu, Y., Zhang, Q., & Huang, Y. 2021, in 2021 8th IEEE Int. Conf. on Cyber Security and Cloud Computing (CSCloud)/2021 7th IEEE Int. Conf. on Edge Computing and Scalable Cloud (EdgeCom), ed. M. Qiu (Piscataway, NJ: IEEE), 103
Law, C. J., Abruzzo, M. W., Bassa, C. G., et al. 2017, ApJ, 850, 76
Lorimer, D., & Kramer, M. 2004, Handbook of Pulsar Astronomy (Cambridge: Cambridge Univ. Press)
Maan, Y., & van Leeuwen, J. 2017, in 2017 32nd General Assembly and Scientific Symp. of the Int. Union of Radio Science (URSI GASS) (Piscataway, NJ: IEEE),
Magro, A., Karastergiou, A., Salvini, S., et al. 2011, MNRAS, 417, 2642
Masui, K., Lin, H.-H., Sievers, J., et al. 2015, Natur, 528, 523
Mikhailov, K., & Sclocco, A. 2018, A&C, 25, 139
Morello, V., Rajwade, K. M., & Stappers, B. W. 2022, MNRAS, 510, 1393
Novotný, J., Adámek, K., & Armour, W. 2022, in ASP Conf. Ser. 532, Astronomical Data Analysis Software and Systems XXX, ed. J. E. Ruiz, F. Pierfedereci, & P. Teuben (San Francisco, CA: ASP), 401
Rajwade, K., Stappers, B., Williams, C., et al. 2020, Proc. SPIE, 11447, 114470J
Ransom, S., 2011 PulsaR Exploration and Search TOolkit, Astrophysics Source Code Library, ascl:1107.017
Ransom, S. M., Eikenberry, S. S., & Middleditch, J. 2002, AJ, 124, 1788
Richards, A., 2015 University of Oxford Advanced Research Computing, Technical Note v1, Zenodo, doi:10.5281/zenodo.22558
Scholz, P., Spitler, L. G., Hessels, J. W. T., et al. 2016, ApJ, 833, 177
Sclocco, A., Heldens, S., & van Werkhoven, B. 2020, SoftX, 12, 100549
Sclocco, A., van Leeuwen, J., Bal, H., & van Nieuwpoort, R. 2016, A&C, 14, 1
Singh, S., Roy, J., Panda, U., et al. 2022, ApJ, 934, 138
Spitler, L. G., Cordes, J. M., Hessels, J. W. T., et al. 2014, ApJ, 790, 101
White, J., Adámek, K., Roy, J., et al. 2023, ApJS, 265, 13
Williams, S., Waterman, A., & Patterson, D. 2009, Commun. ACM, 52, 65
Wolf, M. E., & Lam, M. S. 1991, in Proc. of the ACM SIGPLAN 1991 Conf. on Programming Language Design and Implementation, PLDI '91, ed. D. S. Wise (New York: ACM), 30
Zackay, B., & Ofek, E. O. 2017, ApJ, 835, 11