Zero-Copy-Based Pulsar GPU Coherent Dedispersion Algorithm Postprint

Authors: Wang Boqun, Zhang Hailong, Wang Jie, Ye Xinchen, Wang Wanqiong, Jia Li, Zhang Meng, Yazhou Zhang, Fu Pengfei

Date: 2020-12-21T00:00:00+00:00

Abstract

Radio pulsar signals experience profile broadening and distortion due to the influence of the interstellar medium during propagation, necessitating dedispersion processing in research. This paper designs and implements a zero-copy GPU-based coherent dedispersion algorithm for pulsar data, which employs device memory mapping to eliminate host-to-device copy overhead, utilizes CUDA's cuFFT library for multi-batch Fourier transforms to enhance DFT efficiency, and concurrently employs multi-threading to accelerate transfer function computation. Experimental results demonstrate that the proposed algorithm exhibits favorable performance with large data volumes compared to traditional CPU and GPU algorithms.

Full Text

Coherent De-dispersion Algorithm for Pulsar GPU Based on Zero-Copy

Boqun Wang[12], Hailong Zhang[1234]*, Jie Wang[124], Xinchen Ye[14], Wanqiong Wang[1], Jia Li[1], Meng Zhang[12], Yazhou Zhang[12], Pengfei Fu[12]

[1]Xinjiang Astronomical Observatory, Chinese Academy of Sciences, Urumqi 830011, China
[2]University of Chinese Academy of Sciences, Beijing 100049, China
[3]Key Laboratory of Radio Astronomy, Chinese Academy of Sciences, Nanjing 210008, China
[4]National Astronomical Data Center, Beijing 100101, China

Abstract: Radio pulse signals are broadened and distorted by the interstellar medium during propagation, necessitating de-dispersion processing in pulsar research. This paper designs and implements a zero-copy-based GPU coherent de-dispersion algorithm for pulsar data. The algorithm employs device memory

mapping to eliminate copy overhead from host to device, utilizes CUDA's cuFFT library for multi-batch Fourier transforms to improve DFT efficiency, and adopts multithreading to accelerate transfer function calculations. Experimental results demonstrate that the proposed algorithm performs well with large data volumes compared to traditional CPU and GPU algorithms.

Keywords: zero-copy; coherent de-dispersion; GPU; CUDA; cuFFT

1. Dispersion and De-dispersion Algorithms

Pulsars are rapidly rotating neutron stars with extremely strong magnetic fields, first discovered by Jocelyn Bell in 1967. Since then, radio pulsars have become an important research direction in modern astronomy. As tools, pulsars enable investigations into high-precision timing and time transfer, celestial dynamics and astrometry, gravitational physics in strong-field regimes, exoplanets, galaxies and interstellar medium, ultra-dense matter, and plasma physics under extreme conditions [**?**]. In recent years, projects such as gravitational wave detection using millisecond pulsars and pulsar navigation have emerged, making high-precision pulsar observation techniques a hot topic in radio astronomy.

Pulsar radiation is extremely weak, requiring radio telescopes with high sensitivity. Building large antennas, reducing receiver system noise, and increasing receiver bandwidth are essential methods for improving sensitivity. Among these, increasing observation bandwidth is an economical and feasible strategy, but bandwidth cannot be increased indefinitely. Larger bandwidths create enormous pressure on subsequent processing and cause severe dispersion effects on pulsar signals by the interstellar medium, resulting in pulse profile broadening and deformation. In severe cases, this may even make pulse profiles unobservable. Therefore, de-dispersion processing is mandatory in pulsar-related research.

In recent years, GPU-based parallel computing technology has become a research hotspot in high-performance computing, significantly accelerating scientific analysis and simulation applications [**?**]. NVIDIA' s Compute Unified Device Architecture (CUDA) enables GPUs to solve relatively complex problems [**?**], providing a foundation for computational platform construction. Internationally, using CPU+GPU heterogeneous computing platforms for real-time pulsar signal processing has become mainstream. GPU-based de-dispersion processing reduces CPU load and substantially improves computational system performance.

This paper focuses on pulsar observation data, first analyzing the working principles of GPU de-dispersion, then experimentally investigating performance bottlenecks in GPU de-dispersion algorithms, and finally designing and implementing a zero-copy-based GPU coherent de-dispersion algorithm with validated performance through experiments.

The interstellar medium is a low-temperature plasma. When pulsar electromagnetic radiation propagates through it to Earth, dispersion occurs, causing high-frequency components to arrive earlier than low-frequency components. Dispersion is the most important characteristic of pulse signals, commonly quantified by Dispersion Measure (DM), which represents the integral of electron density along the line of sight between the pulsar and Earth, as expressed in equation (1):

$$DM = \int n_e dl$$

where n_e is the electron number density and l is the line-of-sight distance between the pulsar and the observing telescope.

The time delay Δt between frequencies f_1 and f_2 caused by the interstellar medium is given by equation (2):

$$\Delta t = k_{DM} \cdot DM \cdot (f_1^{-2} - f_2^{-2})$$

There are two methods for pulsar signal de-dispersion: incoherent de-dispersion and coherent de-dispersion. Incoherent de-dispersion divides the time-domain signal with observation bandwidth BW into multiple narrow channels using filters, then shifts each channel' s signal in the time domain according to the delay provided by equation (2), and finally accumulates the amplitudes of all channels to obtain the final signal sequence [**?**]. Coherent de-dispersion treats the interstellar medium' s effect on pulsar signals as an equivalent filter. The typical approach involves Fourier transforming the telescope-received signal, multiplying it by the inverse of the equivalent filter' s transfer function (the chirp function), and then performing an inverse Fourier transform to obtain the original pulsar signal in the time domain [**?**].

The interstellar medium transfer function can be decomposed into amplitude and frequency responses. The equivalent filter must correct both signal amplitude and phase, leading to the discrete chirp function expression [**?**] shown in equation (3):

$$chirp = T_k H_k 0.47 B^2 \cdot \exp\left[-i\left(\pm \frac{DM f_k (2.41 \times 10^{-10})}{f_c^2 (f_c \pm f_k)}\right)\right]$$

where f_k is defined piecewise, T_k is the taper function for amplitude correction, H_k^{-1} is the inverse of the pulse signal transfer function' s phase response for phase correction, N is the number of frequency-domain points in the discrete signal, f_c is the observation center frequency, f_k is the frequency at the k-th spectral point, and B is the observation bandwidth.

2. Zero-Copy

[Figure 1: see original paper] shows the CPU+GPU heterogeneous program execution model. Its core concept places serial code on the host (CPU) and parallel code on the device (GPU), leveraging GPU multithreading to reduce computational overhead. This model assumes that host and device maintain separate memory spaces in DRAM, called host memory and device memory. During execution, preprocessed data must be copied from host memory to device memory, and after GPU processing, results must be copied back to host memory, creating significant communication overhead.

Zero-copy refers to GPU direct mapping of host memory and access via PCIe without explicit memory transfers [**?**]. It requires pinned mapped memory that kernel threads can access directly, reducing memory copy overhead. However, since data is not cached in GPU, pinned mapped memory can only be read and written once; multiple read/write operations reduce efficiency, representing zero-copy' s limitation.

[Figure 2: see original paper] illustrates the zero-copy process. The code first checks GPU support for host memory mapping via `cudaGetDeviceProperties()` returning `prop.canMapHostMemory`. It then enables pinned memory mapping by calling `cudaSetDeviceFlags()` with `cudaDeviceMapHost`. `cudaHostAlloc()` allocates pinned mapped host memory, and `cudaHostGetDevicePointer()` obtains a pointer to the mapped device address space. In zero-copy code, the kernel can reference pinned host memory using pointer `H_{MAP}` exactly as it would operate on device memory.

---

3. Zero-Copy-Based Coherent De-dispersion Algorithm

Telescopes receive pulsar signals contaminated by Radio Frequency Interference (RFI), defined as unwanted radio signals caused by human activity. RFI can degrade pulsar data quality, affecting subsequent scientific research, or even cause pulse signal loss, drowning useful signals in noise. Therefore, RFI mitigation is essential. RFI divides into narrowband and broadband interference. GPS communication signals, aircraft communication signals, and FM broadcast signals are narrowband interference—narrow in frequency domain but persistent in time domain. Lightning, high-voltage cables, electric fences, and signals causing transient electrical effects constitute broadband interference—localized in time domain but polluting multiple frequency channels, exhibiting broadband characteristics.

Since broadband interference is difficult to identify in the frequency domain but appears as short, strong pulses in the time domain, broadband RFI mitigation occurs in the time domain through the following method: (1) Take data blocks of size B composed of n points: $B_1, B_2 \ldots B_n$; (2) Calculate the root-mean-square (rms) of block values; (3) Replace values in B exceeding $s \cdot rms$ with rms, where

s is a global broadband threshold. This is described by equation (4):

$$B_k = \sqrt{\sum}, \quad B_k \geq s \cdot rms$$

Narrowband interference persists continuously in time, making separation difficult in the time domain, but appears as prominent spikes in the frequency domain. Therefore, narrowband RFI mitigation occurs in the frequency domain: (1) Transform the broadband-RFI-mitigated data block B to frequency domain as F with size $m = n/2 + 1$; (2) Calculate the modulus $|F_1|, |F_2| \ldots |F_m|$ for each point in F and compute their average ($mean$); (3) Generate a filter K of length m, setting $K_i = 0$ if $|F_i| > u \cdot mean$ and $K_i = 1$ otherwise, where u is a global narrowband threshold; (4) Multiply K with F to obtain the narrowband-RFI-mitigated frequency sequence. This is described by equations (5):

$$F = F \cdot K$$

$$K_i = \begin{cases} 0 & |F_i| > u \cdot mean, \quad 1 \leq i \leq m \\ 1 & \text{otherwise} \end{cases}$$

$$mean = \frac{\sum |F_i|}{m}$$

The coherent de-dispersion algorithm requires transforming time-domain sequences to the frequency domain, necessitating Discrete Fourier Transform (DFT). For efficient Fourier transforms, the original data must first be partitioned into blocks, each undergoing DFT, multiplication by the interstellar medium' s transfer function for de-dispersion, and inverse DFT (IDFT) to obtain the final time sequence.

Traditional CPUs use FFTW for DFT—a standard C library for fast DFT computation and currently the fastest open-source FFT program [**?**]. NVIDIA GPUs require cuFFT for DFT [**?**], providing a highly optimized and extensively tested FFT library that leverages GPU computational power for rapid transforms.

[Figure 4: see original paper] shows the core GPU DFT code for real-to-complex transforms. It defines the number of batches ($BATCH$) and data volume per batch (N), then declares `cufftComplex` and `cufftReal` pointers. `host_{in}` and `host_{out}` represent host-side (CPU) input/output arrays, while `device_{in}` and `device_{out}` represent device-side (GPU) arrays. A `cufftHandle` defines the cuFFT plan—a one-dimensional plan constructed via `cufftPlan1d` using $BATCH$ and N to control batch processing, informing the GPU how many one-dimensional transforms to perform and the data volume per transform, enabling DFT on different blocks.

After allocating pinned mapped memory for host arrays using zero-copy (core code shown in [Figure 2: see original paper]) to save memory copy overhead, `cufftExecR2C` executes the Fourier transform, writing results to `device_{out}`. Notably, cuFFT automatically removes symmetric DFT components during R2C transforms: if the real input has N elements, the resulting complex output contains only $N/2 + 1$ elements.

To maximize instruction throughput, all mathematical computations in the kernel use CUDA' s intrinsic functions. `sinf` and `cosf` replace conventional `sin` and `cos`, accelerating execution without affecting final results. Additionally, following David Střelák et al.' s recommendations [**?**], the following optimization strategies are adopted: (1) Truncating data to ensure lengths are powers of two, speeding execution; (2) Using single-precision transforms with out-of-place DFT result caching, improving performance; (3) Employing multi-batch transforms rather than repeated `cufftPlan1d` calls, reducing runtime.

4. Experimental Results

The development equipment used for algorithm testing is listed in , with Windows 10 as the operating platform, Visual Studio 2015 and NVIDIA Visual Profiler as development tools, and C, CUDA C, and Python as programming languages.

Table 1: Development Equipment

Device Type	Storage	Frequency
Intel Core i7-4720HQ	-	2.6 GHz
NVIDIA GeForce GTX 960M	-	1.1 GHz
Intel Core i9-9900K	-	3.6 GHz
NVIDIA GeForce RTX 2080 SUPER	-	1.65 GHz

Real pulsar baseband data in PSRDADA format with dual-polarization 8-bit sampling was used for testing. The observation source was J0437-4715, with observation details in . The baseband data recorded approximately 8 seconds, totaling 12.8 GB.

Table 2: J0437-4715 Observation Information

Parameter	Value
Period	0.005757451936712637 s
Center frequency	1382 MHz
Observation bandwidth	400 MHz
Telescope	Parkes

To verify algorithm correctness, the following experiment was conducted: First, 11.72 GB of raw data was folded to generate a profile as a control. Then, with a fixed block size of 16 MB, the algorithm processed 50, 250, and 750 data blocks (0.78 GB, 3.91 GB, and 11.72 GB respectively), generating profiles for each case. Results are shown in [Figure 5: see original paper], where subplots display pulse phase on the x-axis and normalized amplitude on the y-axis: (a) shows raw data without de-dispersion; (b), (c), and (d) show pulse profiles after de-dispersion for 50, 250, and 750 blocks respectively.

Comparing Figure 5: see original paper and (d) confirms that the algorithm correctly corrects pulse signals, completely revealing the pulse profile. Comparison with J0437-4715 profiles from the Xinjiang Astronomical Observatory Data Center validates the results. Comparing (b), (c), and (d) shows that signal-to-noise ratio gradually improves as the number of data blocks increases.

To verify execution speeds of CPU, traditional GPU, and the proposed algorithm (ZGPU) across different data volumes, the following experiment was performed: With a fixed block size of 8 MB, the number of processed blocks was set to 10, 50, 100, 200, 400, 800, and 1500 (corresponding to 0.08 GB, 0.39 GB, 0.78 GB, 1.56 GB, 3.13 GB, 6.25 GB, and 11.72 GB). Each method ran ten times per data volume, with average runtime recorded. To evaluate hardware performance impact, two equipment configurations from were tested. Results are in , with time units in seconds.

Table 3: Experimental Results

[The table content would be preserved here with proper formatting]

[Figure 6: see original paper] plots CPUB-CPU versus GPUB-ZGPU, [Figure 7: see original paper] plots GPUB-GPU versus GPUB-ZGPU, and [Figure 8: see original paper] plots GPUA-ZGPU versus GPUB-ZGPU. The naming convention "Model-Algorithm" indicates the platform and method (e.g., GPUA-GPU represents the traditional GPU algorithm on GPUA).

Analysis of [Figure 6: see original paper] and reveals that both algorithms exhibit linear time growth with data volume, but the traditional CPU algorithm' s average runtime far exceeds the proposed algorithm. At 11.72 GB, the CPU algorithm requires 768.446 seconds for de-dispersion, while ZGPU needs only 49.945 seconds—a 15× speedup. This demonstrates the proposed algorithm' s superiority over traditional CPU methods.

Analysis of [Figure 7: see original paper] shows linear time growth for both algorithms. At small data volumes, the difference between ZGPU and traditional GPU is negligible, but the gap grows linearly with data volume. At 400, 800, and 1500 blocks, the time differences are 0.949 s, 1.935 s, and 3.658 s respectively—doubling data volume doubles the time difference. This confirms ZGPU's advantage over traditional GPU algorithms, particularly for large data volumes.

Analysis of [Figure 8: see original paper] demonstrates that ZGPU' s average runtime on newer GPU architecture is significantly shorter. At 1500 blocks,

ZGPU runs for 166.208 seconds on GPUA but only 49.945 seconds on GPUB—a 3× speedup. This establishes hardware performance as a critical factor affecting the algorithm.

This paper designed and implemented a zero-copy-based GPU pulsar coherent de-dispersion algorithm. By properly constructing kernels for maximum parallel execution, employing zero-copy technology to improve memory throughput, optimizing instruction calls with CUDA intrinsic functions, and enhancing cuFFT efficiency in DFT, the algorithm outperforms traditional CPU methods and, for large data volumes, traditional GPU algorithms. Real data testing validates these improvements. The zero-copy GPU de-dispersion algorithm provides a novel solution for efficient pulsar data processing. Experimental data and source code are available at the Xinjiang Astronomical Observatory Data Center[1].

[1] http://data.xao.ac.cn/static/zerocopy.rar

Acknowledgments: This work was supported by the National Natural Science Foundation of China (11873082, 11803080), the National Key R&D Program (2018YFA0404704), the Youth Innovation Promotion Association of the Chinese Academy of Sciences, and the National Astronomical Data Center and the Chinese Academy of Sciences Scientific Data Center System. Data resources and technical support were provided by the Chinese Virtual Observatory, the National Astronomical Data Center, and the Chinese Academy of Sciences Scientific Data Center System.

References:

[1] Yonghua Xu, Jintao Luo, Zhixuan Li, et al. A pulsar observation system with a Mark5B recording system and a DSPSR software [J]. Astronomical Research & Technology, 2013, 10(4): 352-358.

[2] J. Gu, M. Chowdhury, K. G. Shin, et al. Tiresias: A GPU cluster manager for distributed deep learning [C]//16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). 2019: 485-500.

[3] Nvidia C. CUDA C programming guide, version 10.1 [J]. NVIDIA Corp, 2019.

[4] Yuxiang Huang, Min Wang, Longfei Hao, Zhixuan Li, Yonghua Xu. Comparative study between the coherent de-dispersion and the incoherent de-dispersion of pulsar signal [J]. Astronomical Research & Technology, 2019, 16(1): 16-24.

[5] Xianhai Wang. Research on the key techniques for the wideband real-time pulsar digital receiver [D]. Southeast University, 2016.

[6] I. H. Stairs. Observations of binary and millisecond pulsars with a baseband recording system [D]. Princeton University, 1998.

[7] Profiler N. NVIDIA visual profiler–NVIDIA developer zone [J]. 2014.

[8] R. N. Bracewell. The Fourier transform and its applications [M]. New York: McGraw-Hill, 1986.

[9] M. Frigo, S. G. Johnson. FFTW: An adaptive software architecture for the FFT [C]//Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP' 98. IEEE, 1998, 3: 1381-1384.

[10] NVIDIA. CUDA Fast Fourier Transform library (cuFFT) [J]. 2018.

[11] D. Střelák, J. Filipovič. Performance analysis and autotuning setup of the cuFFT library [C]//Proceedings of the 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems. 2018: 1-6.

Note: Figure translations are in progress. See original paper for figures.