A Multi-GPU Spectrometer System for Real-time Wide Bandwidth Radio Signal Analysis

Hirofumi Kondo*, Eric Heien*, Masao Okita*, Dan Werthimer† and Kenichi Hagihara*
*Graduate School of Information Science and Technology
Osaka University, Japan
{h-kondou, e-heien, okita, hagihara}@ist.osaka-u.ac.jp
†Space Sciences Laboratory
University of California, Berkeley, USA

***Abstract*—This paper describes the implementation of a large bandwidth multi-GPU signal processing system for radio astronomy observation. This system performs very large Fast Fourier Transform (FFT) and spectrum analysis to achieve real-time analysis of a large bandwidth spectrum. This is accomplished by implementing a four-step FFT algorithm in Compute Unified Device Architecture (CUDA). The key feature of this implementation is that the data size transferred between CPU and GPU is reduced using redundant calculation. We also apply pipeline execution to our system to minimize idle processor time, even with multiple GPUs on a shared bus. Using a single GPU, this system can analyze 1 GB of signal data (128 MHz bandwidth at 1 Hz resolution in single precision floating-point complex format) in 0.44 seconds. With the multi-GPU setup, using four GPUs enables 4 GB of signal data to be processed in 0.82 seconds. This is equivalent to a processing speed of around 60 GFLOPS. In particular, we focus on using this system in the Search for Extraterrestrial Radio Emissions from Nearby Developed Intelligent Populations (SERENDIP) project. By using multiple GPUs we can get enough practical performance for high bandwidth radio astronomy projects such as SERENDIP.**

I. Introduction

In recent years the field of radio astronomy has benefited from faster signal processing technologies. Projects use custom and/or off-the-shelf hardware configurations to perform various analyses of radio signal data. The aim of these projects ranges from detection of extraterrestrial intelligence [1] [2] to mapping of the interstellar medium [3]. Many of these projects require fast sophisticated analysis of large amounts of data.

In large bandwidth radio astronomy, it may not be feasible to record all the signal spectrum for later analysis due to the sheer volume of data. For example, the SERENDIP V project has the ultimate target of continuously analyzing a 2.1 GHz spectrum at a resolution of 1.5 Hz per channel. If each channel requires 16 bits (8-real and 8-imaginary), this will produce over 220 TB of data per day. Therefore real-time signal analysis systems are required.

In this paper, we describe the single and multi-GPU implementation of a large bandwidth signal processing system for radio astronomy observation. In particular, the development of this spectrometer is aimed toward use in SERENDIP or similar data intensive projects.

As an initial goal, the SERENDIP V project aims to analyze a 128 MHz bandwidth spectrum at less than 1 Hz resolutions in real-time. This is equivalent to a data bandwidth of 1 GB per second for a single-precision floating-point complex data format. If the resolution and/or bandwidth increases, the discrete spectra will also increase as well as the amount of data.

To handle these large bandwidth spectra, custom designed hardware is often used for radio astronomy observation. In recent years, these setups often consist of Field Programmable Gate Arrays (FPGA). For example, the SERENDIP V project [4] [5] uses an FPGA based system called the SETI Spectrometer [5], composed of two main components: Interconnect Break-out Board (IBOB) [6] and Berkeley Emulation Engine 2 (BEE2) [7]. The BEE2 system is composed of five FPGAs and is able to process a 128M points spectra in one second.

The current FPGA based system has enough performance for current astronomy observation, but this system has two disadvantages. First, when applied to large bandwidth applications such as this, FPGAs are often expensive compared to GPUs. The SETI Spectrometer costs approximately $20,000 while the NVIDIA Tesla S1070 used in this paper costs less than half while achieving higher performance. The second, because large computations must be split over multiple FPGAs, increasing the input bandwidth of such a system may require significant redesign work.

Based on these points, we attempted to build a large bandwidth radio signal processing system which improves on the performance of the FPGA based system. CPU based implementation that has the same function of SETI Spectrometer requires 5.94 seconds (result of preliminary experiment using FFTW [8] on Intel Xeon CPU X5450 3.00 GHz). Therefore we require GPUs to accelerate the wide bandwidth radio signal analysis.

In recent years GPUs are becoming widely used as scientific application accelerators. Most GPUs have high calculation speed with some achieving nearly one TFLOPS of theoretical maximum speed. There are three significant benefits to building a spectrometer system with GPUs:

1) GPU has relatively low cost because it is mass-produced hardware.
2) Availability of the CUDA [9] programming environment and associated libraries facilitates programming on GPU
3) New signal processing functionality can be easily added

978-0-7695-4190-7/10 $26.00 © 2010 IEEE
DOI 10.1109/ISPA.2010.53

using the CUDA environment

We implemented the GPU spectrometer initially using one GPU, then later four GPUs working together. Using one GPU, the size of VRAM limits the maximum length of spectra that one GPU can process in real-time. To process longer length spectra, we must use multi-GPU and distribute input data to GPUs.

The GPU spectrometer requires a CUDA implementation of very large 1-D FFT (over 128M points) for frequency analysis. In the case of the FPGA based SETI Spectrometer, only a 32k-point 1-D FFT is required despite the 128M point input spectra. This is because it divides the input spectrum by frequency domain and performs many small FFTs. To divide the spectra by frequency domain, the SETI Spectrometer uses a steep cut-off bandpass filter. This bandpass filter cannot divide the frequency band precisely, and causes some noise in the signal. As a division method, the signal also can be divided in the time domain, but this time decomposition decreases the frequency resolution of FFT results. Therefore performing a very large FFT without any division is the optimal solution which we aimed for with the GPU spectrometer. A very large FFT was accomplished by implementing a four-step FFT algorithm on GPU. For multiple GPUs, this four-step FFT algorithm is also applied, but it is distributed over the GPUs.

There are two main problems in the implementation of this GPU spectrometer, both caused by data transfer. The first problem involves memory copy from the host machine to the GPU. To process the signals, we have to transfer radio signal data from the controlling CPU to the co-processing GPU. Transferring large amounts of data can negatively affect performance by idling the GPU processing units.

The second problem arises from using multiple GPUs connected to the host machine with a shared PCI-Express bus. In this case we have to consider the timing of data transfers to prevent bus conflicts.

To overcome these problems, we use two techniques. First, to reduce the total amount of transferred data from CPU to GPUs, we decompose the input data over GPUs and places an overlap region (referred to as *ghost zone* [10]) on each GPUs. This technique causes the redundant calculation on GPUs. Second, data bus conflicts are avoided by applying pipeline execution to GPU spectrometer. By applying these techniques to our implementation, we get practical performance for the SERENDIP project.

The paper is organized as follows. In Section II we describe related work and how our system is new and different. In Section III we provide a brief overview of the CUDA library, and in Section IV some background on the required spectrometer functionality and limitations. The implementation on a single GPU and related issues are described in Section V, while the implementation on multiple GPUs is described in Section VI. We offer experimental results for both the single and multi-GPU implementation in Section VII and finish with conclusions and directions for future work in Section VIII.

II. Related Work

Many astronomy observation projects use an FPGA based system. For example, the SERENDIP V project uses a Xilinx Virtex-II FPGA based system called the SETI Spectrometer. This is a 128M channel, 200 MHz, 1 polarization spectrometer. The feature of this system is splitting the 200 MHz input spectra into sub-bands of 24.4 kHz each using a PFB (Polyphase Filter Bank) [11] [12] and performing frequency analysis on each bandwidth chunk.

In other work, GPU based systems are used to accelerate astronomy observation. Harris et al [13] use CUDA to implement a GPU FX spectrometer. This system performs correlation for telescopes with multiple antennae and achieves speeds one to two orders of magnitude faster than a CPU based system.

Our system requires a 1-D FFT on GPU for frequency analysis. Currently there are a number of FFT implementations available for GPUs. Recently, FFT implementations for the CUDA API have achieved very good performance [14][15][16].

Govindaraju et al [14] present the implementation of many FFT algorithms on GPU. Their implementation supports not only power-of-two but also non-power-of-two FFT and achieved over 300 GFLOPS calculation performance. They presents the performance results of up to 16M-point 1-D FFT but this length of FFT is not enough for our GPU spectrometer (e.g. over 128M-point 1-D). The GPU vendor NVIDIA provides an FFT library called CUFFT [16] that also has very good performance, but does not support FFTs of very large inputs either. The CUFFT supports up to only 8M-point 1-D FFT.

Nukada et al [15] discuss a 3-D bandwidth intensive FFT implementation on GPU. Their 3-D FFT covers large sizes up to 512^3 points in single-precision floating-point format. They use an out-of-core method to do very large 3-D FFT.

To the best of our knowledge, there are no very large FFT implementations supporting multiple GPUs. However, there is an FFT implementation for clusters (Takahashi et al [17]) that handles FFTs up to 2 billion points. This implementation requires all-to-all communication between the cluster nodes. However, all-to-all communication is not appropriate for multi-GPU systems because current generation GPUs cannot directly communicate with each other.

III. Compute Unified Device Architecture(CUDA)

Compute Unified Device Architecture (CUDA) is NVIDIA's parallel computing architecture for GPU. We can imagine that a GPU is a SIMD parallel machine that executes hundreds to thousands threads. The CUDA program (called *kernel*) is launched from CPU.

The computing unit in a GPU has a hierarchical structure. The GPU is organized with several multiprocessors (MPs), where each MP has several stream processors (SPs) for processing threads. There is an on-chip memory on each MP (called *shared memory*) that SPs can use to share processing data. Depending on this structure, threads are divided into groups (called *thread blocks*), where each thread block is

arbitrarily assigned to an MP. In regards to this point, we must program a kernel that has no data dependency between different thread blocks.

To get high performance in the kernel, it is necessary to program it being aware of the memory access pattern. Global memory is an off-chip memory in the GPU which takes relatively long to access. To reduce memory access latencies, memory access by threads is coalesced into memory transactions if they satisfy some conditions. To satisfy these conditions, we have to be conscious of the memory access pattern. In other instances, the shared memory has a bank structure. If several threads access the same bank, the memory access instructions are serialized (called *bank conflict*).

Other specifications about CUDA are written in the programming manual [9] in detail.

IV. GPU-Spectrometer Overview

The GPU Spectrometer accelerates frequency analysis of an input radio spectrum. The SERENDIP project uses spectrometers to search radio bands that could contain non-natural signals from extraterrestrial intelligence (ETI). This project assumes that the ETI sends their message on a narrow bandwidth of unknown frequency. Based on this assumption, the spectrometer must detect narrow band signals of relatively higher strength than the background noise.

This section provides an overview of our GPU spectrometer and describes the input data, calculations performed and resulting output.

A. Spectrometer Input and Output

Here we describe the characteristics of input and output data for the spectrometer.

The frequency band of interest is first read by a radio telescope and sent to an Analog Digital Converter (ADC). The ADC converts the observed band into digital format then a Decimating quadrature DownConverter (DDC) converts the band into a baseband. We assume that the GPU spectrometer can use the FPGA based ADC and DDC that SETI Spectrometer uses. Depending on the application, this may pass through additional filters before eventually being written to the main memory of the host machine.

The input data is digitized radio data that the GPU spectrometer can read from the main memory of host machine. One sample of the digitized radio data is described with a 16-bit complex data sample comprised of two characters (8-bit real and 8-bit imaginary). For example, the size of input data is 256 MB for a 128 MHz spectrum at 1 Hz resolution.

The output data consists of detected signals from the input spectra that have relatively strong energy. The GPU spectrometer outputs these spectra into a file in binary format. The output information is detected signal strength, frequency band that the signal is in (index of input data) and mean power of the local spectrum. Usually the number of signals output by the spectrometer is roughly 5 orders of magnitude smaller than the input data.

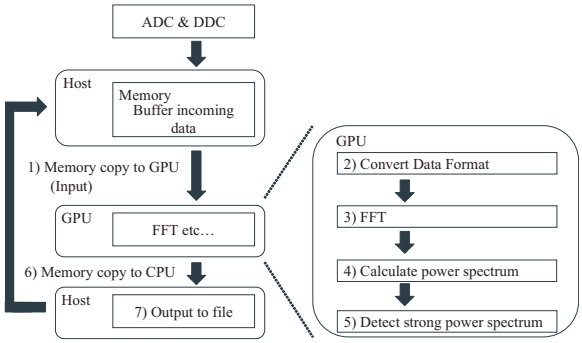


Fig. 1. Processing flow of GPU-Spectrometer

B. Processing flow

The GPU spectrometer operation involves seven stages which are repeated continuously while the spectrometer is operating. In the case of stages performed on GPU, each stage share processing data on global memory on GPU. For example, the FFT stage reads input data from global memory and writes result back to global memory. The GPU spectrometer operates some stages simultaneously to reduce the number of global memory access. The processing flow is illustrated in Figure 1.

1) Host to GPU memory copy: Transfer input data from host memory to GPU over the PCI-Express bus.

2) Convert data format: Convert the input complex data from character (1 byte) format to float (4 bytes) format on GPU. This is because of insufficient precision if the GPU does calculations with a character format.

3) FFT: Perform FFT on GPU. This FFT is the most computationally intensive stage.

4) Calculate power spectrum: Calculate power spectrum from the results of FFT. If element n in the FFT result is described as $x_n + iy_n$, the corresponding power spectrum element is $x_n^2 + y_n^2$.

5) Detect strong power spectrum: Our system detects relatively strong elements of the power spectrum. Strong elements in the power spectrum are defined as those whose power exceeds the mean power of the local spectrum times a user-specified threshold value. For example, if the threshold is 10 and the mean spectrum power is 20, then signals over power 200 will be reported. This stage is composed of these calculations:

i) Elements of the power spectrum are divided into blocks of user-specified size (called *boxcars*)
ii) The average strength of the power spectrum for each boxcar is calculated
iii) Spectrum elements with power over the boxcar average times threshold are located

6) GPU to CPU memory copy: Transfer detected power spectrum from GPU to CPU.

7) Output detected power spectrum: Output transferred data into a file on the host machine.

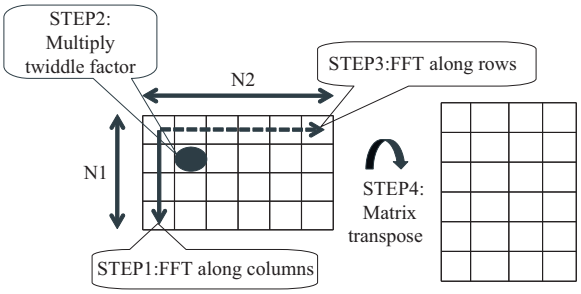


Fig. 2. Four-step FFT algorithm

C. Detailed specification

Here we describe the detailed specification. We assume that the input signal is a power of two because the FFT performance is declined. We also assume that the GPUs that compose this system are homogeneous and the number of GPU is a power-of-two. The reason of this assumption is to uniform the load of each GPU if the same length input is given to each GPU. It should be possible to create a system without these assumptions, however the performance will likely be lower than described in this paper.

V. Single-GPU spectrometer implementation

This section describes the implementation using single GPU. Because the spectrometer operates in real-time, it must analyze a given set of data within a certain time limit. Due to memory limitations, this is one second worth of signal data that must be processed within one second of wall time.

As an initial goal, the GPU spectrometer is processing a 128M points spectra in one second. We intend to replace the SETI Spectrometer with our GPU spectrometer, so that we decide the goal as same as the SETI Spectrometer.

The most challenging part of the spectrometer is the FFT implementation because it is very computationally intensive. To accelerate the performance of large FFT, we design an FFT implementation with the intention of preventing inappropriate memory accesses on the GPU.

In addition, we discuss the maximum length FFT a single-GPU spectrometer can handle under the execution time and GPU hardware limitations.

A. FFT implementation

In this implementation we use the *four-step FFT algorithm* [18]. Assume that the FFT data is stored in the array A of length N which can be expressed as the product of two positive integers N_1 and N_2, or $N = N_1 N_2$.

True to its name, the four-step FFT algorithm consists of the following four steps (Figure 2). Fundamentally, this algorithm performs a large FFT by dividing it into two small FFTs.

STEP1) Perform N_2 simultaneous N_1-point 1-D FFTs on the input data A treated as a $N_1 \times N_2$ (row-major) complex matrix.

STEP2) Multiply the elements A_{jk} in the resulting array considered as a $N_1 \times N_2$ matrix, by $e^{\pm 2\pi ijk/n}$ (twiddle factor). The $\pm$ sign is the sign of the transform.

STEP3) Perform N_1 simultaneous N_2-point 1-D FFTs on the $N_1 \times N_2$ complex matrix.

STEP4) Transpose the $N_1 \times N_2$ complex matrix into a $N_2 \times N_1$ matrix.

Note that the FFT along the matrix columns (STEP1) has a problem. Such FFT requires non-successive memory address access, which is inappropriate for GPU and causes a performance degradation. In terms of efficient GPU throughput, access of successive memory addresses is desirable. Most FFT implementations on GPU, such as CUFFT, assume that the FFT is performed with successive address access.

If we intend to apply CUFFT to STEP1, we have to transpose the matrix before STEP1. In that case, we have to transpose the matrix again before STEP3. These additional matrix transposes requires relatively long time, which spoils the advantage of the use of CUFTT.

To solve this problem, we implement a register-intensive FFT kernel and apply it to STEP1 without additional matrix transposes. Up to 16-point FFT, the kernel manages the sufficient number of active threads on multiprocessors because it requires only around fifty registers per thread. The kernel accesses to the global memory just for twice for each element of the matrix, and threads is moreover coalesced at the accesses. Therefore the kernel has relatively good performance even for non-successive addressed data. A disadvantage of this kernel is the limitation of N_1 depended on the number of registers. We decide N_1 as 16 in our implementation.

On the other hand, we apply CUFFT 2.3 library to STEP3 because this library has enough performance for real-time analysis. The limitation of CUFFT determines N_2. CUFFT can perform up to 8M-point FFT, so that the maximum size we can calculate on single GPU is 16 $\times$ 8M = 128M point.

For STEP4, we implement a matrix transpose kernel based on the transpose kernel included in CUDA SDK 2.3[19]. This CUDA kernel has good performance because of the fully coalesced access and bank conflict avoidance. We extend the kernel to perform the matrix transpose while simultaneously calculating power spectra. Our kernel reduces the number of memory accesses compared to performing power spectrum calculation after the matrix transpose.

B. Maximum signal length

The memory size of VRAM determines the maximum signal length that a single GPU can handle in real-time.

For our single-GPU spectrometer, all input data must be stored in the GPU VRAM. From the perspective of the implementation, our system could perform very large (over a few giga points) FFT. In the four-step FFT algorithm, STEP1 and STEP3 perform many independent FFTs. This means we can divide input data into parts and repeat as follows: send a part of data to GPU, perform small FFTs and send results back to CPU. Such an out-of-core method increases the total

amount of data transferred between GPU and CPU, requiring very long total time.

To complete the processing in real-time, all data should be transferred to GPU at one time and the FFT should be performed without any data transfer during its calculation. This means to ensure real-time execution of the GPU spectrometer, the maximum FFT length that a single-GPU spectrometer can handle is limited by the GPU VRAM capacity. The current GPU, such as Tesla, has up to 4 GB VRAM, so that it can perform the 128M-point FFT well within memory limitations. We discuss the limitation by the VRAM capacity later in Section VII-B.

C. Convert data format

The FFT is a time consuming part but the others are not because the calculations are relatively easy to parallelize.

The conversion of input spectra from character to float are easy to parallelize. This is because neither data dependency nor control dependency exist among the elements. Each complex input spectrum element can be converted independently. We can divide data into parts such that one thread block processes one part and each thread in a thread block processes one element of the part.

D. Detect strong power spectrum

The detection of strong power spectrum is somewhat complicated to parallelize.

This is because we have to calculate the average value of the power spectrum within a boxcar. We cannot arbitrarily divide input data into parts, but we can divide data into boxcars. One thread block manages each boxcar and calculates the average value of the power spectra within it.

Be carefully managing threads and memory access, the GPU spectrometer is able to perform these calculations at high efficiency.

VI. Multi-GPU spectrometer implementation

The goal of Multi-GPU spectrometer is processing more large spectra in one second, without any spectrum division. As mentioned in Section I, high resolution analysis requires a very large FFT. As the size of spectrum increases, the VRAM capacity of one GPU becomes insufficient. To perform the analysis in real-time, we must distribute input data and calculations over multiple GPUs. This implementation can support large spectra, up to 512M point.

The most challenging part is a distributed FFT implementation because it requires a large data transfer between the CPU and GPUs. The large data transfer causes performance degradation.

Our implementation uses four techniques to improve performance.

1) reduce the total amount of transferred data by applying ghost zone.
2) overlap redundant calculation and data transfer to prevent the redundant calculation from increasing total execution time.

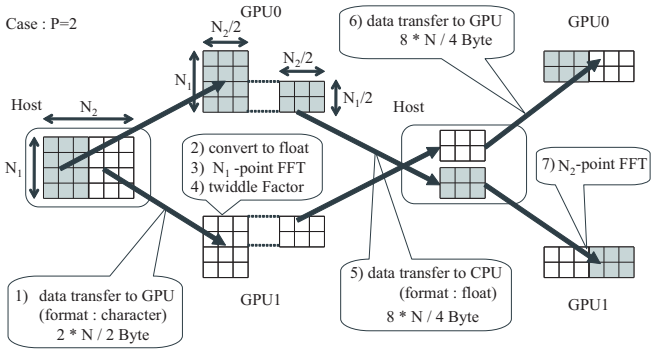


Fig. 3. Data flow and transferred data size on naive FFT implementation for P=2

3) apply pipeline execution in our GPU spectrometer to minimize idle processor time.
4) omit the matrix transpose required in the four-step FFT algorithm to reduce the transferred data size.

A. Naive FFT implementation

Before explaining our implementation, we describe the naive distributed four-step FFT implementation for comparison.

The naive implementation is the realization of a four-step FFT distributed on multiple GPUs. Assume there are P GPUs and $N = N_1 N_2$ point input spectra (considered as a $N_1 \times N_2$ complex matrix). The multi-GPU spectrometer would then operate as follows:

1) $N_1 \times N_2/P$ point spectra is sent from CPU to each GPU. Transferred data size for each GPU is $2 \times N/P$ bytes, resulting in a total of $2 \times N$ bytes.
2) convert input data from character format to float format
3) perform N_2/P simultaneous N_1-point FFTs (STEP1)
4) apply twiddle factor to result of STEP1 (STEP2)
5) transfer $(N_1 \times (P-1)/P) \times (N_2/P)$ point result of STEP2 from GPU to CPU. The transferred data size is $8 \times N \times (P-1)/P^2$ bytes for each GPU and $8 \times N \times (P-1)/P$ bytes total.
6) transfer $(N_1/P) \times (N_2 \times (P-1)/P)$ point result from CPU to GPU. The transferred data size is $8 \times N \times (P-1)/P^2$ bytes for each GPU and $8 \times N \times (P-1)/P$ bytes total.
7) perform N_1/P simultaneous N_2-point FFTs (STEP3)

For P=2, we illustrate the data flow and data size of naive FFT implementation in Figure 3. Before STEP4, the total amount of data transferred between CPU and GPU is $(2 \times N + 16 \times N \times (P-1)/P)$ bytes.

The advantage of this implementation is the scalability with the number of GPUs. When the number of GPUs increases, the total amount of transferred data stays constant (at most $18 \times N$ bytes) because the amount of calculation on each GPU decreases. On the other hand, when the bandwidth between GPU and CPU is small, this $18 \times N$ byte data transfer will be a bottleneck of the GPU spectrometer. For

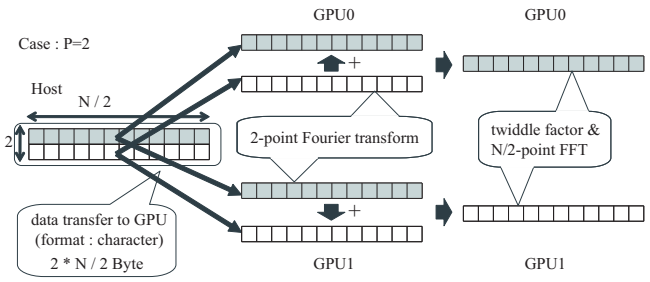


Fig. 4. Data flow and transferred data size on our FFT implementation for P=2

example, the data transfer size is 9 GB when the input data length is 512M point (N=512). A 9 GB data transfer requires over one second execution time assuming the experimental setup has 8 GB/s bandwidth between GPU and CPU, making this implementation infeasible for real-time large scale signal analysis.

B. Ghost zone

The naive implementation requires very large data transfer between GPUs. To eliminate the data transfer between GPUs, we apply the ghost zone technique to GPU spectrometer. This ghost zone uses all input data rather than just a part. In other words, each GPU receives all input data then performs a Fourier transform, which includes redundant calculation. To reduce the redundant calculation, we set the matrix form of four-step FFT algorithm as rectangular where the matrix row is very short. We have to perform FFT along columns of this matrix as STEP1. Each GPU cannot store all input data in VRAM at the same time due to memory limitations. We transfer only one row of matrix to GPU, then each GPU performs a Fourier transform (not *fast Fourier transform*) on the received data. This Fourier transform is repeated from the first row to the last row sequentially.

We assume that there are P GPUs and an N point input spectrum. We consider this N point spectrum as rectangular $P \times N/P$ matrix of complex values, with matrix elements indicated as $x_n(m)(0 \leq n \leq N/P-1, 0 \leq m \leq P-1)$. Based on the four-step FFT algorithm, first we perform N/P simultaneous P-point $(x_n(0), x_n(1), \cdots, x_n(P-1))$ FFTs. The result of N/P simultaneous P-point FFTs is represented as a matrix of complex values and each element is indicated with $F_n(m)$.

GPUk $(0 \leq k \leq P-1)$ performs a Fourier transform of $(F_0(k), F_1(k), \ldots, F_{N/P}(k))$. In other words, one GPU performs a Fourier transform to calculate one row of $P \times N/P$ matrix. To calculate $F_n(k)$ on each GPUk, we transfer each row of the matrix to all GPUs sequentially until all data is transferred. When a GPU receives one row of the matrix, it computes the sum of the received data and previous data in such a way that it ends up performing a P-point Fourier transform.

Here we describe the FFT implementation algorithm in pseudo-code on GPU:

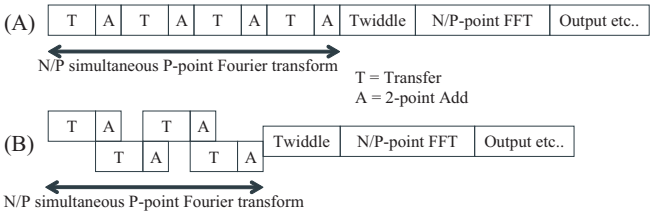


Fig. 5. Execution history

$F_n(k) = 0 \ (0 \leq n \leq N/P-1)$
for $m = 0$ to $P-1$ **do**
  Memcpy_to_GPU($x_n(m)$) $(0 \leq n \leq N/P-1)$
  $F_n(k) += x_n(m)e^{-2\pi imk/P} (0 \leq n \leq N/P-1)$
end for
Multiply_twiddle_factor($F_n(k)$)
Perform_N/P-point_FFT($F_n(k)$)

Each GPU can perform the twiddle factor multiplication and N/P-point FFT independently. Our implementation data flow is illustrated in Figure 4. There is no data transfer between GPUs in our FFT implementation. The maximum data length for the multi-GPU spectrometer is $P\times$ 128M point, or in other words $N/P = 128$M. We can perform the 128M-point FFT based on the single-GPU spectrometer.

This implementation requires $2 \times P \times N$ bytes of data transfer because each GPU requires all the spectrum. When the number of GPU is 2 or 4, the total amount of transferred data size is smaller than that of the naive implementation. This is because the transferred data between GPU and CPU is only character format data.

While the amount of transferred data is smaller than that of the naive implementation, the amount of calculation is larger than that of the naive implementation. For example, in the case of $P = 4$, $F_n(0)$ and $F_n(2)$ are calculated by these equations:

$$\begin{aligned} F_n(0) &= \{x_n(0) + x_n(2)\} + \{x_n(1) + x_n(3)\} \\ F_n(2) &= \{x_n(0) + x_n(2)\} - \{x_n(1) + x_n(3)\} \end{aligned}$$

$F_n(0)$ is calculated on GPU0 and $F_n(2)$ is calculated on GPU2. GPU0 and GPU2 partially perform the same calculation which is redundant calculation.

A problem of our FFT implementation is low scalability. The total amount of transferred data, $2 \times P \times N$ bytes, is proportional to the number of GPUs P. To the best of our knowledge, the maximum number of GPUs in modern multi-GPU environments is 8. The naive implementation requires a maximum of $18 \times N$ bytes of data transfer while our implementation requires $2 \times P \times N$ bytes. Therefore, our implementation is more effective than the naive implementation on current multi-GPU environments.

C. Overlap calculation and transfer

These redundant calculations are irrelevant to the total execution time because they can be overlapped by data transfer. This is because the data transfer time is relatively longer than calculation. For example, in the case of $P = 4$, we get

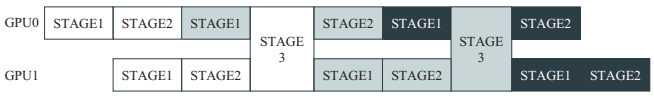


Fig. 6. Pipeline execution on 2GPUs

an execution history like Figure 5-(A) without overlapping. Figure 5-(A) shows that our system performs: first the repeat data transfer and 2-point add ($F_n(k) += x_n(m)e^{-2\pi imk/P}$), next the twiddle factor multiplication, and finally the N/P-point FFT. However, when we overlap the data transfer and 2-point add like in Figure 5-(B), the redundant calculations are hidden by data transfer. We cannot overlap the last 2-point addition calculation.

D. Pipeline execution

We focus on the PCI-Express bus architecture of GPUs to minimize the idle processor time by applying pipeline execution to the multi-GPU spectrometer.

1) PCI-Express bus architecture: GPUs are connected to their host machines on a PCI-Express (PCIe) bus. In some cases, several GPUs share one PCIe bus. For example, in the NVIDIA Tesla S1070, there are 4GPUs and 2 PCIe buses (2GPUs share one PCIe bus). Two GPUs that share one bus transfer their data at same time, degrading the bandwidth that one GPU can use.

We coordinate the timing of transfer to avoid bandwidth degradation. For the sake of simplicity, in this discussion we assume the case of 2 GPUs.

2) 2GPUs without shared PCIe bus: In this case there are 2 GPUs without a shared PCIe bus, in other words, each GPU has own PCIe bus. It is unnecessary to coordinate data transfer timing because no transfer collision occurs.

3) 2GPUs with shared PCIe bus: In this case, 2 GPUs share one PCIe bus so we must coordinate the timing of transfer to prevent conflicts.

To coordinate the transfer timing we use pipeline execution. We divide the overall operation into 3 stages:

STAGE1) data transfer includes N/P simultaneous P-point Fourier transforms
STAGE2) twiddle factor multiplication and N/P-point FFT
STAGE3) calculate power spectrum, threshold and output to file

Figure 6 shows a pseudo execution history of 2GPUs with shared PCIe bus. While one GPU processes STAGE1, another will process STAGE2. Both GPUs processes STAGE3 at the same time. This pipeline execution ensures that high bandwidth is available to each GPU.

4) Implementation on Tesla S1070: NVIDIA Tesla S1070 has two couples of GPUs and each GPU couple shares one PCIe. This environment represents both *with shared PCIe bus* and *without shared PCIe bus*. Therefore, we apply a hybrid method shown as Figure 7. In this figure, GPU0 and GPU2 share one PCIe, while GPU1 and GPU3 share the other PCIe.

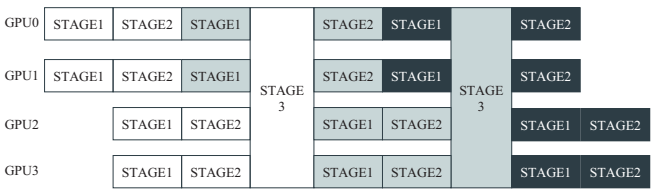


Fig. 7. Pipeline execution on the Tesla S1070

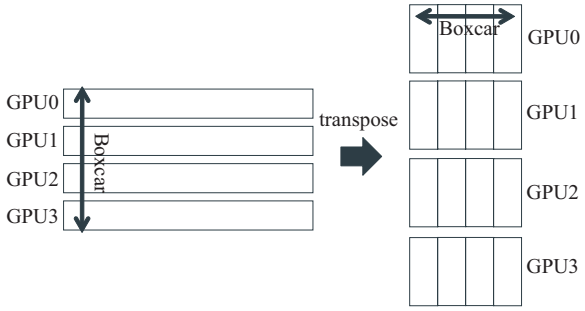


Fig. 8. Matrix transpose for arranging boxcars

E. Thresholding power spectrum

As shown in Figure 2, the four-step FFT implementation transposes a matrix at STEP4. The matrix transpose locates the FFT result in successive memory addresses. It simplifies the following step of strong signal detection (IV-B5) by localizing a boxcar in each GPU as shown in Figure 8-right. However, it also results in large data transfer between CPU and GPUs because GPUs must communicate with each other via the main memory of the host machine.

On the other hand, our implementation avoids the matrix transpose in order to reduce data transfer between CPU and GPU. If the size of a boxcar is larger than the number of GPUs, the elements of that boxcar are distributed among all GPUs (Figure 8-left). To detect strong power spectra, GPUs exchange the partial sum of their own boxcar fragment with each other instead of the whole fragment, then they calculate the mean value of the boxcar for use in thresholding. The transferred data size for these partial sums is much smaller than that required by the matrix transpose.

VII. Results

A. Experimental Methodology

We performed experiments with our GPU spectrometer using the NVIDIA Tesla S1070 computing system. This system includes four NVIDIA Tesla C1060 GPUs, so we can use up to 4GPUs in multi-GPU spectrometer experiments. Each GPU has 4 GB VRAM. Tesla S1070 is fully compliant with PCI-Express 2.0 x16 (theoretical bandwidth 8 GB/s), but the host machine has two buses that are compliant with PCI-Express 2.0 x8 (theoretical bandwidth 4 GB/s). The host machine specification is shown in Table I.

The input data to our system is *white noise* with a periodic pulse. Usually a radio telescope observes white noise. White

TABLE I
THE HOST MACHINE CONFIGURATION FOR EXPERIMENT

CPU	Intel(R) Xeon(R) CPU E5450 × 4
OS	Cent OS release 5.3
Memory	16 GB
Software	CUDA Toolkit 2.3, GCC 4.1.2
PCI-Express	PCI-Express 2.0 x8 × 2

TABLE II
THE NUMBER OF POINTS OF SIMULATED POWER SPECTRUM AND ITS DATA SIZE

Input data length	Output data size	simulated power spectrum point
16M	24 kB	2047
32M	48 kB	4096
64M	96 kB	8192
128M	177 kB	15082
256M	192 kB	16383
512M	192 kB	16384

noise can be created by random number sequence that we create by a Mersenne Twister pseudo-random generator. In this experiment, to simulate the strong power spectrum at specific bands, we add periodic pulse signals to the white noise. For timings we generate various lengths of white noise with pulses and execute the GPU spectrometer 20 times on each length signal, then take the average time.

The number of points of simulated strong power spectrum and its data size are illustrated in Table II, and our GPU spectrometer outputs these simulated strong power spectrum. Note that the GPU spectrometer must also output the observation date (time stamp), observatory location, etc, for practical observation. However, in this experiment, our system does not output such information, so the output data size will be greater in actual observation. This causes the increase of total execution time depending on the applications, but we think these data size is small enough compared to strong power spectrum data size.

In this section, the FLOPS of our FFT implementation is estimated by the following formula where M simultaneous N-point FFT.

$$M \times 5 \times N \log_2 N / (FFT\ execution\ time)$$

B. Single-GPU spectrometer Result

For the single-GPU spectrometer, we test our system with varying lengths of input spectra from 16M to 128M points. Figure 9 and Figure 10 show the total execution time of the single-GPU spectrometer and the total execution time breakdown, respectively. We also describe the total execution time breakdown in Table III.

Figure 9 indicates that the total execution time is shorter than one second for all cases. This demonstrates that our system has enough performance for real-time radio signal analysis up to 128M point spectra.

Figure 10 indicates the FFT calculation accounts for the largest fraction of the total execution time. To process the spectra more faster, the FFT calculation have to be accelerated. We expect that our four-step FFT implementation has some

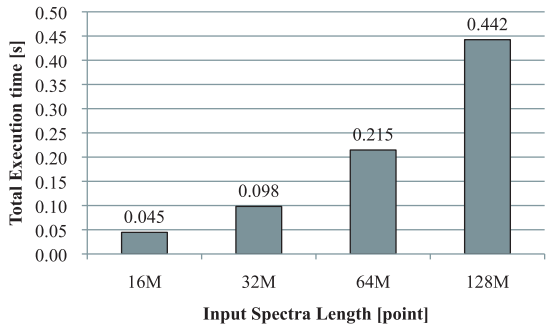


Fig. 9. Total execution time of single-GPU spectrometer with changing input data length from 16M to 128M point

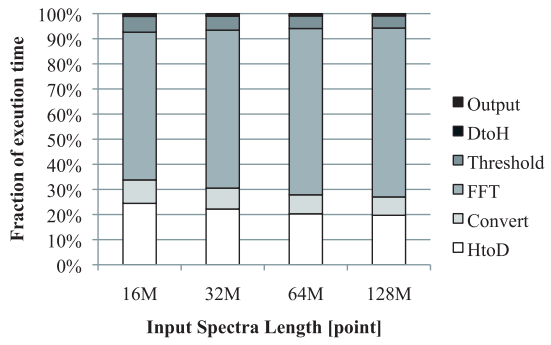


Fig. 10. Breakdown of execution time with changing input data length from 16M to 128M point: DtoH(transfer from Device to Host), HtoD(transfer from Host to Device)

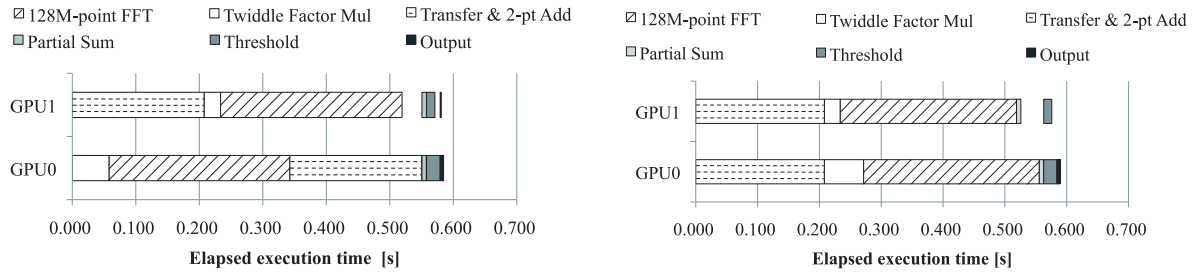
improvements. When the input spectra length is 128M point, our 16-point FFT kernel perform only 55 GB/s memory bandwidth and 69 GFLOPS. These values are smaller than the peak bandwidth (102 GB/s) and peak theoretical floating-point operation performance (933 GFLOPS).

We discuss here the memory limitation for the GPU spectrometer. The four-step FFT implementation requires $N \times 14$ bytes memory at least, where N represents the length of input spectrum. The breakdown is $N \times 2$ bytes for saving input spectra, $N \times 8$ bytes for FFT and $N \times 4$ bytes for matrix transpose including power spectrum calculation. This memory area for matrix transpose is reduced by the performing matrix transpose while simultaneously calculating power spectrum. This is because the memory area for matrix transpose requires the same size of FFT but the power spectrum requires half memory size of FFT.

Our implementation also requires the additional memory area for thresholding power spectrum, performed CUDA program binary and others. These additional memory size is much smaller than that of FFT or matrix transpose. For example, when the input spectra length is 128M-point, required memory size is 1.75 GB at least, from 2.0 GB to 2.5 GB at most.

TABLE III
BREAKDOWN OF EXECUTION TIME AND ITS PERCENTAGE TO TOTAL EXECUTION TIME. THE EXECUTION TIMES ARE ROUNDED TO THE MILLISECOND.

	HtoD		Convert		FFT		Threshold		DtoH		Output		TOTAL
	second	%	second	%	second	%	second	%	second	%	second	%	second
16M	0.0109	24.4	0.0042	9.3	0.0263	58.8	0.0029	6.5	0.0001	0.3	0.0003	0.6	0.045
32M	0.0218	22.2	0.0082	8.3	0.0618	62.9	0.0057	5.8	0.0003	0.3	0.0005	0.5	0.098
64M	0.0436	20.3	0.0162	8.6	0.1421	66.2	0.0113	5.2	0.0005	0.2	0.0010	0.5	0.215
128M	0.0872	19.7	0.0324	7.3	0.2975	67.2	0.0225	5.1	0.0010	0.2	0.0019	0.4	0.442



(a) CASE 1PCIe (b) CASE 2PCIe

Fig. 12. Execution history of two-GPU spectrometer (input spectra length is 256M point)

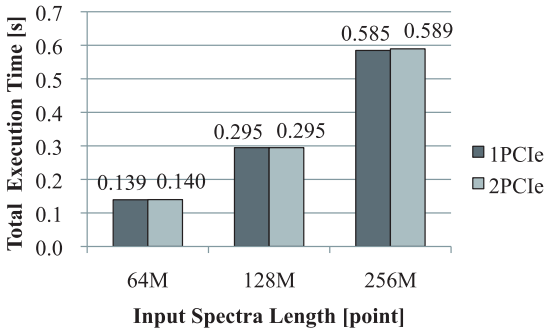


Fig. 11. Total execution time of two-GPU spectrometer

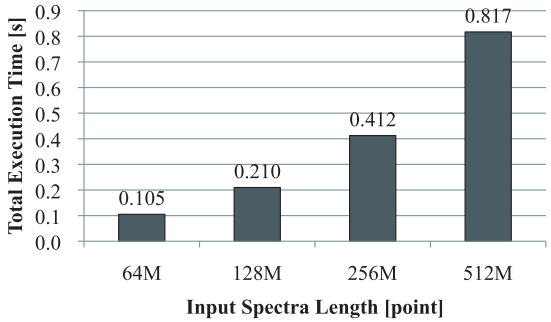


Fig. 13. Total execution time of four-GPU Spectrometer

C. Two-GPU spectrometer result

We tested our two-GPU spectrometer with varying input spectra length from 64M to 256M points. We experiment with both cases of *1PCIe* (GPUs with shared a PCIe bus) and *2PCIe* (GPUs without shared PCIe buses).

Figure 11 represents the total execution time of the two-GPU spectrometer in the case of 1PCIe and 2PCIe. This total execution time is the elapsed time from the end of outputting the power spectrum to the next end of outputting the power spectrum in pipeline execution. Both cases have approximately the same total execution time and both times are shorter than one second. The differences of execution time between the 1PCIe case and 2PCIe case are caused from the experimental error.

An execution history for 256M point is shown in Figure 12a and Figure 12b. In 1PCIe case, our system accomplished pipeline execution. The execution time of transfer and 2-point add is around 0.21 seconds, equal to 2PCIe case. This indicates that there is no decline of bandwidth in the case of 1PCIe because of no collisions between GPU0 transfer and GPU1 transfer. Thus, our two-GPU spectrometer resolves the problem of sharing a PCIe bus.

In the 2PCIe case, the 128M-point FFT time accounts for around half of the total execution time. This 128M-point FFT time is around 0.28 seconds, and this is larger than the transfer and 2-point add time. To improve the performance of 2PCIe case, we have to improve the performance of 128M-point FFT.

D. Four-GPU spectrometer result

We also tested our four-GPU Spectrometer with varying input spectra length from 64M to 512M points. As mentioned in Section VI-D4, the environment contains two couples of GPUs and each GPU couple shares one PCIe.

Figure 13 represents the total execution time of four-GPU Spectrometer. This total execution time is the elapsed time from the end of outputting the power spectrum to the next end of outputting the power spectrum. The total execution time is shorter than one second for up to 512M point using four GPUs.

TABLE IV
COMPARISON OF EXECUTION TIMES ON GPU0 (INPUT SPECTRA LENGTH IS 128M-POINT).

	FFT length per GPU(point)	Transfer Data Size per GPU (byte)	HtoD* (s)	Convert (s)	Twiddle (s)	FFT (s)	Threshold** (s)	Output (s)	Wait (s)	Total (s)
1GPU	128M	256M	0.087	0.032	-	0.298	0.023	0.002	-	0.442
2GPU(2PCIe)	64M	256M	0.105	-	0.033	0.139	0.014	0.003	-	0.295
4GPU	32M	256M	0.097	-	0.018	0.060	0.011	0.003	0.020	0.210

* In the case of 2GPU and 4GPU, HtoD includes the execution time of 2-point and 4-point FFT (2-point add) execution time, respectively.
** Threshold includes the execution time of DtoH and partial sum time.

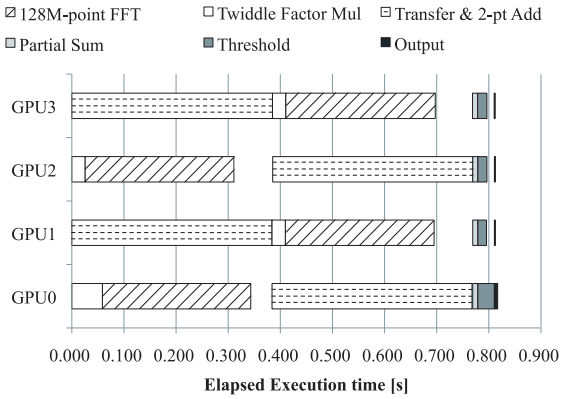


Fig. 14. Execution history of four-GPU Spectrometer for 512M point input

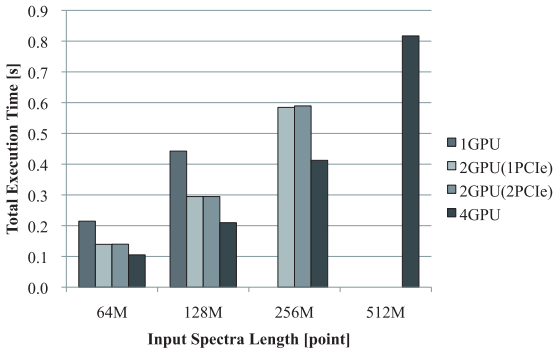


Fig. 15. Comparison of total execution times. 1GPU cannot analyze 256M and 512M-points spectra because of memory limitation. 2GPU also cannot analyze 512M-points spectra for same reason

The execution history when the input spectra length is 512M point is shown in Figure 14. Figure 14 indicates that the bottleneck of our system is data transfer. The GPUs have idle time because 128M-point FFT and twiddle factor multiplication time is shorter than the transfer time. In this experiment environment, the bandwidth between CPU and GPUs is narrow because of PCI-Express 2.0 x8. If the host machine had PCI-Express 2.0 x16 buses, the FFT would be a bottleneck. However, as the input data size increases, the transfer becomes a bottleneck even with PCI-Express 2.0 x16 buses. This means that the bottleneck of GPU spectrometer can change as the input spectra length becomes longer. The bottleneck is FFT (computation) when the input spectra length is up to 256M or 512M point, but it is data transfer when the input spectra length is over 1G-point.

E. Total execution time comparison

We summarize the execution times of the GPU spectrometer in Figure 15. The total execution time of the same spectra length is reduced when the number of GPUs increase and it has nonlinear speed-up.

This nonlinear speed-up is caused by the transferred data size per GPU that does not change as the number of GPUs increases. We summarize the breakdown of execution time of the GPU spectrometer in Table IV when the input spectra length is 128M-point. The execution time of FFT is reduced by half as the number of GPUs doubles because the length of FFT that one GPU perform is halved. In fact, when input spectra are 128M point, execution time of FFT is around 0.30 seconds using one GPU, and is around 0.14 seconds using two GPUs. On the other hand, the time for data transfer is not reduced by half because transferred data size is constant. In fact, when input spectra is 128M point, transfer time is around 0.087 seconds using one GPU, and is around 0.105 seconds using two GPUs.

This increment of transfer time is attributed to the calculation of last 2-point add that is not overlapped by transfer. In the case of 2GPU, the transfer time is longer than 1GPU case because HtoD includes the 2-point add operation. In the case of 4GPU, the length of last 2-point add is shorter than the case of 2GPU, then the increment of execution time of HtoD is slightly short.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we described the implementation of a large bandwidth multi-GPU signal processing system for radio astronomy observation. This system has to process very large amounts of data in real-time.

To do frequency analysis on large spectra in real-time, we implement fast large point FFT. In the case of single GPU, we implement a four-step FFT algorithm on GPU. In the case of multi-GPU, we also implement this algorithm on GPUs. We apply the ghost zone technique to reduce the total amount of transferred data between CPU and GPU. We also apply pipeline execution to minimize idle processor time.

Using a single GPU, this system can analyze 128M point spectra (1 GB of signal data) in 0.44 seconds. The single-GPU spectrometer analyzes at twice the speed of the FPGA based

system for the same amount of data. Two and four GPUs can analyze up to 256M point spectra (2 GB) and 512M point spectra (4 GB), respectively. Using four GPUs allows 4 GB of signal data to be processed in 0.82 seconds.

Future work involves improving the scalability of GPU spectrometer. Our GPU spectrometer has the disadvantage of increasing total amount of transferred data as the number of GPUs increase. We also intend to apply our implementation to other radio astronomy observation projects.

ACKNOWLEDGMENT

The authors would like to thank Paul Demorest of the National Radio Astronomy Observatory, and Terry Filiba of the University of California, Berkeley for their assistance. The authors also would like to thank the members of Hagihara laboratory in Graduate School of Information Science and Technology, Osaka University for their insightful comments and suggestions. This work was partly supported by JSPS Grant-in-Aid for Scientific Research (A)(20240002), and by the Global COE Program "in silico medicine" at Osaka University.

REFERENCES

[1] D. Werthimer, D. Anderson, C. S. Bowyer, J. Cobb, E. Heien, E. J. Korpela, M. L. Lampton, M. Lebofsky, G. W. Marcy, M. McGarry, and D. Treffers, "Berkeley radio and optical SETI programs: SETI@home, SERENDIP, and SEVENDIP," *Proc. SPIE Vol. 4273*, vol. 4273, p. 104, Aug 2001.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "SETI@home: an experiment in public-resource computing," *Commun. ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[3] W. B. Burton and D. Hartmann, "The Leiden/Dwingeloo survey of emission from galactic HI," *Astrophysics and Space Science (ISSN 0004-640X)*, vol. 217, p. 189, Jul 1994.

[4] D. Werthimer, D. Ng, S. Bowyer, and C. Donnelly, "The Berkeley SETI Program: SERENDIP III and IV Instrumentation," *Progress in the Search for Extraterrestrial Life*, vol. 74, p. 293, 1995.

[5] CASPER project , "SETI Spectrometer," [online] Available:http://casper.berkeley.edu/wiki/SETI_Spectrometer.

[6] CASPER project, "IBOB," [online] Available:http://casper.berkeley.edu/wiki/IBOB.

[7] C. Chang, J. Wawrzynek, and R. Brodersen, "BEE2: a high-end reconfigurable computing system," *Design & Test of Computers, IEEE*, vol. 22, no. 2, pp. 114 – 125, 2005.

[8] M. Frigo and S.G. Johnson, "The design and implementation of FFTW3," in *proceedings of the IEEE*, 2005, pp. 216–231.

[9] NVIDIA Corporation, "NVIDIA CUDA Programming Guide," 2009 - July.

[10] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs," in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 256–265.

[11] A. Parsons and D. Werthimer, "PFB_32, A 32-pnt Bibplex Pipelined Polyphase Filter Bank," [online] Available:http://seti.ssl.berkeley.edu/~aparsons/papers/2003-06_pfb_32.html.

[12] A. Parsons, D. Backer, C. Chang, D. Chapman, H. Chen, P. Crescini, C. de Jesus, C. Dick, P. Droz, D. MacMahon, K. Meder, J. Mock, V. Nagpal, B. Nikolic, A. Parsa, B. Richards, A. Siemion, J. Wawrzynek, D. Werthimer, and M. Wright, "PetaOp/Second FPGA Signal Processing for SETI and Radio Astronomy," *Signals, Systems and Computers, 2006. ACSSC '06. Fortieth Asilomar Conference on*, pp. 2031 – 2035, 2006.

[13] C. Harris., K. Haines., and L. S. Simith, "GPU accelerated radio astronomy signal convolution," in *Experimental Astronomy*, vol. 22, no. 1–2. Springer Netherlands, 2008, pp. 129–141.

[14] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High performance discrete Fourier transforms on graphics processors," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–12.

[15] A. Nukada, Y. Ogata, T. Endo, and S. Matsuoka, "Bandwidth intensive 3-D FFT kernel for GPUs using CUDA," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11.

[16] NVIDIA Corporation, "NVIDIA CUDA CUFFT Library Version 2.3," 2009 - July.

[17] D. Takahashi, "A parallel 1-D FFT algorithm for the Hitachi SR8000," in *Parallel Comput.*, vol. 29, no. 6. Amsterdam, The Netherlands, The Netherlands: Elsevier Science Publishers B. V., 2003, pp. 679–690.

[18] D. H. Bailey, "FFTs in external of hierarchical memory," in *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 1989, pp. 234–242.

[19] NVIDIA Corporation, "CUDA Toolkit and SDK Version 2.3," 2009 - July.