MNRAS **000**, 1–10 (2015) Preprint 5 October 2020 Compiled using MNRAS LATEX style file v3.0

HCGrid: A Convolution-based Gridding Framework for Radio Astronomy in Hybrid Computing Environments

Hao Wang,[1] Ce Yu,[1] Bo Zhang,[2,3]★ Jian Xiao[1]† and Qi Luo[1]
[1]*College of Intelligence and computing, Tianjin University, No.135 Yaguan Rood, Haihe Education Park, Tianjin, 300350, China*
[2]*National Astronomical Observatories, Chinese Academy of Sciences, No.20 Datun Road, Chaoyang District, Beijing, 100012, China*
[3]*CAS Key Laboratory of FAST, National Astronomical Observatories, Chinese Academy of Sciences*

Accepted XXX. Received YYY; in original form ZZZ

ABSTRACT

Gridding operation is to map non-uniform data samples onto a uniformly distributed grid, is one of the key steps in radio astronomical data reduction process. One of the main bottlenecks of gridding is the computing performance, and a typical solution for such performance issue is the implementation of multi-core CPU platforms. Although such a method could usually achieve good results, in many cases, the performance of gridding is still restricted to an extent due to the limitations of CPU, since the main workload of gridding is a combination of a large number of single instruction, multi-data-stream operations, which is more suitable for GPU, rather than CPU implementations. To meet the challenge of massive data gridding for the modern large single-dish radio telescopes, e.g., the Five-hundred-meter Aperture Spherical radio Telescope (FAST), inspired by existing multi-core CPU gridding algorithms such as Cygrid, here we present an open-source convolutional gridding framework **HCGrid**, in CPU-GPU heterogeneous platforms. It optimises data search by employing multi-threading on CPU, and accelerates the convolution process by utilizing massive parallelisation of GPU. In order to make HCGrid more adaptive, we also propose the strategies of thread organisation and coarsening, as well as optimal parameter settings under various GPU architectures. HCGrid is easy to install, high-performance, and publicly available. A thorough analysis of computing time and performance gain with several GPU parallel optimisation strategies show that it can lead to excellent performance in hybrid computing environments.

Key words: methods: data analysis – techniques: image processing – software: public release

1 INTRODUCTION

In radio astronomy, gridding operation is a key step towards generating 2D sky maps or 3D cubes with evenly-spaced grids for scientific research or data release (Winkel et al. 2016b). However, the original observed data are usually not so uniformly distributed. Taking the planned Commensal Radio Astronomy FAST Survey (CRAFTS, see Li et al. (2018)) of the Five-hundred-meter Aperture Spherical Radio Telescope (FAST) (Nan 2006; Nan et al. 2011) as an example. During this survey, the FAST telescope will be operated under the drift-scan mode, that is, the telescope will remain fixed along local meridian, with different patches of the sky entering the field of view due to rotation of the Earth. With a beam size of $\sim 2'.9$ at ~ 1.42 GHz, and rotation angle of ~ 23.4 degrees, the 19-beam receiver of FAST (Smith et al. 2017) can achieve a beam spacing of $\sim 1'09''$ along declination, with super Nyquist coverage guaranteed. While on the right ascension direction, typically the telescope's spectral line backends can record data once every second (Li et al. 2018), with a maximum sampling rate of 10 times per second, thus providing a much denser sampling rate. Hence, gridding is a necessity to resample such observations uniformly, for the convenience of data analysis, visualization or storage (e.g. Léna et al. 2012).

However, gridding is a computing and I/O intensive job (Giovanelli et al. 2005), comprising one of the most time-consuming steps in the complete data reduction process. And for the latest instruments with high data production rates such as FAST, the speed of gridding operation should be of great concern. **Our experiment has shown that with a 16-core CPU platform, it would take** ~ 21 **CPU hours to perform gridding with 1 TB raw data using the multi-thread method.** However, the CRAFTS project is expected to generate as many as $10 \sim 20$ PB-sized data every year (Li et al. 2018). That is, hundreds of CPU-hours are required to grid the raw data generated by FAST in real-time with existing method, which could greatly increase the cost of computation for this survey.

In the past decade, various studies on optimization of gridding performance have been carried out. For example, Winkel et al. (2016b) have presented a versatile gridding module (**Cygrid**) for radio astronomical data reduction, featuring a single level lookup table based on the C++ standard template library (STL) for gridding

★ E-mail: zhangbo@nao.cas.cn
† E-mail: xiaojian@tju.edu.cn

parallelization, as well as the HEALPix spherical tessellation for fast neighbor searching. And the performance of the Cygrid code can be significantly improved when using multiple CPU cores. However, utilising the STL vector, which features dynamical arrangements of storage allocations with a relatively small initial storage volume, to construct a lookup table for input raw data is not always practical. For example, if the raw samples are unevenly distributed, with more data points in the central regions and less on all sides, the required storage needs to be expanded in the middle part of each HEALPix-tessellated ring frequently. In this case, the STL vectors have to make massive data copies of the data frequently, which would consume broad bandwidth, thus resulting in gridding-performance degradation.

In addition, the main workload of gridding can be considered as a combination of a large number of single-instruction, multi-data stream operations. For such tasks, the high parallelization level, as well as throughput of GPUs brings a possibility for better performance. Thus, Romein (2012) has developed a work-distribution scheme for radio data gridding with GPUs, which could significantly reduce the memory access time of computing device, and map observed samples onto a grid with high efficiency. And Merry (2016) has further optimized this algorithm with thread coarsening strategy, making one thread handle multiple samples simultaneously. However, the method proposed by Romein (2012) and Merry (2016) has been deeply customized for radio telescope arrays, strongly relying on the spatial coherence of the interference matrix, thus cannot be easily adapted for single-dish telescopes.

Accordingly, in order to meet the requirements of large data volume from the latest single-dish radio telescopes, and to overcome the shortcomings of GPU-based gridding methods mainly designed for radio telescope array, inspired by Winkel et al. (2016b), and based upon our previous work (Luo et al. 2018), here we present a convolutional gridding framework with hybrid computing environments, **HCGrid**, for radio astronomy, which can greatly improve the performance of the gridding process. Key features of our framework can be summarized as follows:

(i) We have designed a gridding module with high performance for large single-dish radio telescopes.

(ii) We partitioned the sample space based on HEALPix (Hierarchical Equal Area Latitude Pixelation, Górski et al. (2005)). To effectively utilise the storage resources, and to accelerate the searching process of effective contributing data points, we adopted the two-level lookup table scheme proposed by our previous work, which can further improve the degree of parallelism of gridding.

(iii) We have proposed our scheme for thread organization, thread coarsening, as well as data layout for further optimization of the performance gain.

(iv) We have conducted comparative experiments and comprehensive analysis with three mainstream GPU architectures, Kepler, Turning, and Volta, resulting in a detailed and easy-to-use performance optimization guide for various scenarios of applications.

The rest of this paper is organized as follows. Section 2 presents the details of convolution-based gridding algorithms in radio astronomy and the corresponding parallelization strategy. Section 3 focuses on the design of our framework. And the results of the related experiments are described in Section 4. Our conclusion is drawn in Section 5 with further discussions.

2 GRIDDING ALGORITHMS IN ASTRONOMY

2.1 Convolution-based gridding algorithm

The convolution-based algorithm **is among** the most common choice for gridding of radio astronomical data. For example, Kalberla et al. (2005, 2010) have applied this technique to produce data cubes and HI column density maps for the Leiden/Argentine/Bonn Survey, while Winkel et al. (2016a) have adopted a similar approach in the Effelsberg–Bonn HI Survey. The first step of convolutional gridding is to design a target grid, and then the resampled value for each grid cell is calculated.

The algorithm we adopted is mainly from Winkel et al. (2016b). The output value for each targeted grid cell equals the weighted sum of all neighboring samples. Let $\mathbb{S} = \{s_1, s_2, \cdots, s_N\}$ denote N non-uniformly spaced samples distributed across the RA-Dec plane. For nth sample $s_n \in \mathbb{S}$, its equatorial coordinates should be expressed as (α_n, δ_n) (α_n means the right ascension, δ_n the declination), with a sampled value of $V[s_n]$. For our target grid, suppose the RA-Dec plane is divided into a regular grid with $I \times J$ cells $\mathbb{G} = \{g_{1,1}, g_{1,2}, \cdots, g_{I,J}\}$. For any cell $g_{i,j} \in \mathbb{G}$ with central coordinates $(\alpha_{i,j}, \delta_{i,j})$, its resampled value $V[g_{i,j}]$ is equivalent to the weighted sum of original data $\mathbb{S}$ related to $g_{i,j}$ (Winkel et al. 2016b)

$$V[g_{i,j}] = \frac{1}{W_{i,j}} \sum_n V[s_n]\omega(\alpha_{i,j}, \delta_{i,j}; \alpha_n, \delta_n) \quad (1)$$

Here s_n represents any original input sample with a weighted contribution to $g_{i,j}$; $\omega(\alpha_{i,j}, \delta_{i,j}; \alpha_n, \delta_n)$ is a convolution kernel (weighting function) depending on positions of the target cell and original data points, usually related to distances between input and output coordinates; and $W_{i,j} = \sum_n \omega(\alpha_{i,j}, \delta_{i,j}; \alpha_n, \delta_n)$ is the normalization coefficient (Winkel et al. 2016b). Since each input sample can influence different output cells with different degrees, with the introduction of $\omega_{i,j}$, the resampled output values can be correctly calculated.

Generally speaking, the resolution limit of the target (output) grid σ_{grid} (that is, the distance between any two adjacent cells) is determined by the instrument resolution σ_{data}) and the size of the convolution kernel function σ_{kernel} as $\sigma_{grid} = \sqrt{\sigma^2_{data} + \sigma^2_{kernal}}$, with $\sigma_{kernel} \sim \frac{1}{2}\sigma_{data}$ as a common choice Winkel et al. (2016b). Besides, since by adopting the coordinates of FITS (Wells & Greisen 1979) world coordinate system ((WCS, Calabretta & Greisen 2002), as well as (Mink 2006)) in calculations, the gridding process can directly perform convolutions in WCS space, which has proved to be more convenient for astronomical users, we mainly focus on astronomy-specific cases of WCS-targeted gridding in the following discussions.

2.2 Gridding parallelizations

In order to speed up the computing process, the gridding technique is usually parallelized with multithreading using the multiple CPUs or the CPU-GPU heterogeneous computing environments. Two types of parallelization strategies exist, scatter and lookup table-based gather.

The scatter strategy calculates the contribution of each input data point to all target cells within the influence of the weighting kernel, thus facing the risk of "writing competition", that is, conflict when calculating the same target cell value with different set of input data. Although efficient scheduling algorithms can be invoked to avoid the risk of such competitions (McCool et al. 2012),

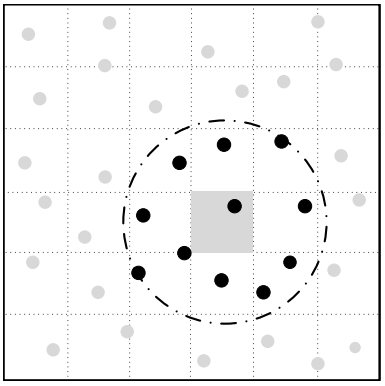

Figure 1. Gather method: one approach of implementing convolution-based gridding. Each output data point (as shown with light-colored grid) collects its resampled value from all neighboring input data (darker dots) lying within a kernel (dotted circle).

the algorithms themselves often show difficulties for GPU implementation, thus rendering the scatter method less effective when deploying under GPU environments (van Amesfoort et al. 2009). On the other hand, although atomic operations, which cannot be interrupted by thread scheduling, can also be utilised to avoid writing competitions, its could also make parallel-scatters degenerate into serialized algorithms in the presence of densely sampled raw data or frequent competitions (Schweizer et al. 2015), thus affecting the speed of gridding operations.

While as show in Fig. 1, the gather method identifies all adjacent input points within the range of influence of the convolution kernel for each output cell, and performs convolution with such contributors to calculate the final results. Although the writing competition issue no longer exists in this case, it is nearly impossible to locate all the input points within a certain kernel radius directly for any designated cell, due to the non-uniformity of the input samples. Thus, massive searching operations are required. To avoid unnecessary searching operations, and to improve the capability of the entire gridding process, the gather strategy usually performs pre-ordering the input data in advance, partitioning the sampled data according to their coordinate information, with the establishment of block number - sampled point lookup table (one-to-many mapping). After that, blocks falling into the convolution kernels of each grid cell can be quickly determined and accessed (see Luo et al. (2018) and references herein). HEALPix (Górski et al. 2005) is a tessellation scheme for the sphere surface. Its properties make it a useful tool for constructions of lookup-table schemes. Based on HEALPix, the gather strategy can take more advantages of GPU, compared with the scatter one. Therefore, we also adopt the gather method in our gridding framework for CPU-GPU hybrid platforms. And the main reference work of this paper, Cygrid, is also based on the gather method.

3 DETAILS OF THE GRIDDING FRAMEWORK IMPLEMENTATION

3.1 Overview of HCGrid

Fig. 2 shows the basic architecture of HCGrid, which consists of three modules:

(i) **Initialisation module** for raw data initialization. This module imports FITS-format input data files, extracting related parameters

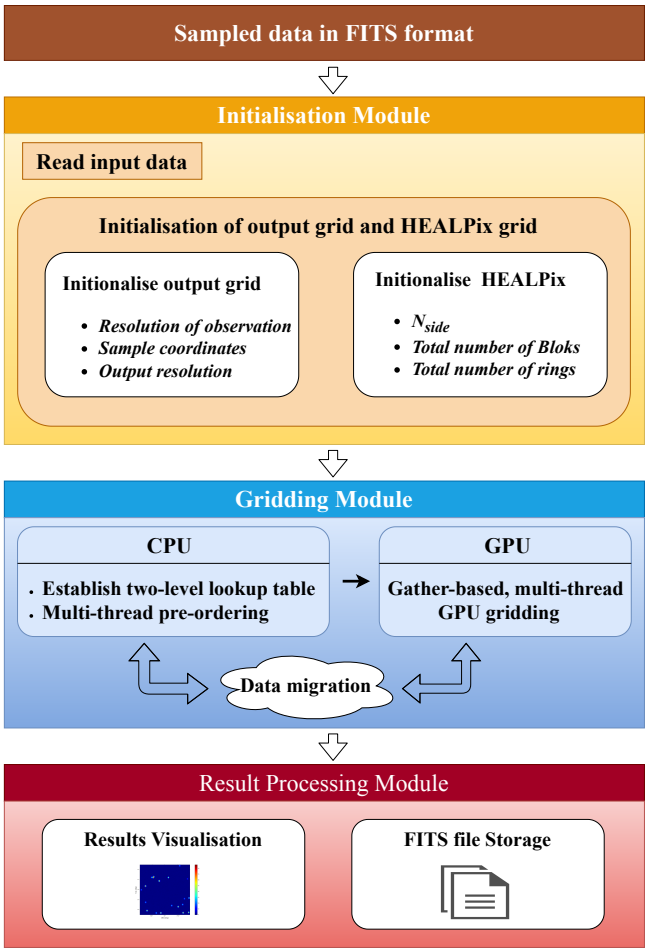

Figure 2. The architecture of HCGrid. HCGrid consists of three modules, including initialization, which is mainly used for initialization of parameters involved in the calculation process; gridding, which consists of the core of HCGrid; and finally, result processing for gridding output visualization and data storage.

(such as σ_{data}, sample coordinates, as well as output resolution) from therein, and initializes the output grid and HEALPix with extracted parameters.

(ii) **Gridding module**, which is the core module of the HCGrid framework. This module performs data pre-ordering on CPU, and gridding on GPU, along with data migration between CPU and GPU.

(iii) **Result-processing module**, which visualise the gridding results, as well as exporting the final products as FITS files.

The gridding module as our main topic of interest for this work, we will mainly discuss the gridding module in the following sections. The source code of HCGrid can be accessed openly via GitHub[1], and all suggestions and user's feedback are welcome.

3.2 HEALPix-based partition on sampling space

As mentioned in Section 2.2, the aim of input-data pre-ordering is to improve the searching efficiency on the contributing data points,

[1] https://github.com/HWang-Summit/HCGrid

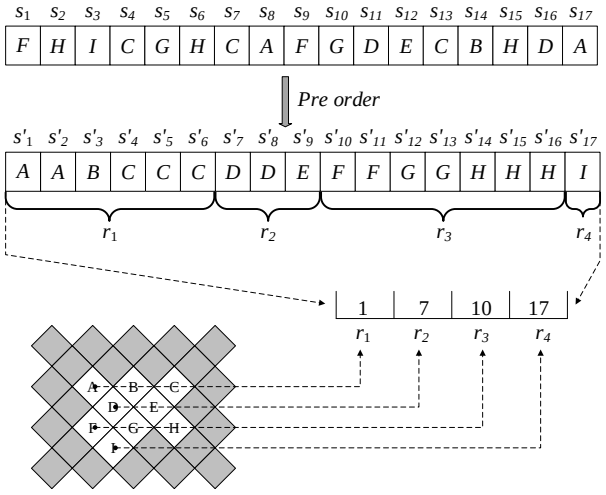

Figure 3. The construction of the two level lookup table. The 17 sampling points can be partitioned into 9 HEALPix pixels, and rearranged according to their pixel indices, using the latitudinal ring structure. Then a secondary lookup table can be established, as $R_{start}[1] = 1$, $R_{start}[2] = 7$, $R_{start}[3] = 10$, and $R_{start}[4] = 17$.

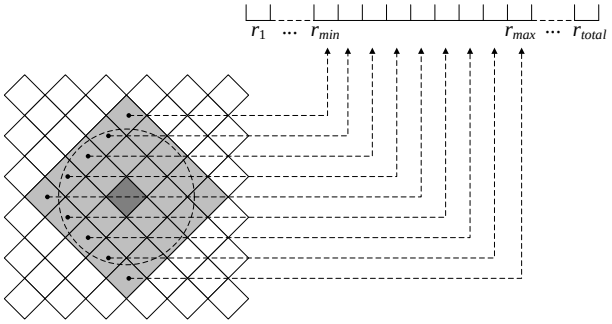

Figure 4. The searching algorithm for contributing data points. r_{min} and r_{max} are the minimum and maximum values of the latitudinal ring number falling inside the range of influence of the convolution kernel, respectively.

as well as the performance of the gridding process, by means of rearrangements of the original samples. Here we adopt HEALPix for the sampling space partition.

HEALPix (Górski et al. 2005) is a software package for hierarchical equal-area isolatitude pixelation on spherical surfaces, thus making fast, accurate statistical or astrophysical analysis of massive all-sky data sets possible. With the help of HEALPix-based raw-data space partition, one can determine pixel and ring indices of raw data within a certain area, thus reducing the workload of searching operation, and enabling more reasonable distributions of raw data covering a spherical surface. HEALPix provides two possible implementations of pixel indexations, the ring scheme and the nested scheme, arranged on isolatitude rings, or in a nested tree fashion, respectively. Both schemes can map input samples to a one-dimensional numbered sequence. Since with the ring scheme, the pixel indices increase strictly linearly along the latitude ring, which makes it easier to establish a lookup table based on positions of each pixel, here we choose to utilise such a scheme as basis for data pre-ordering and lookup-table construct.

The HEALPix software library supports coordinate conversions based on the WCS standard ((Calabretta & Greisen 2002) and (Mink 2006)). Through related APIs of this library, one can make a series of pixel manipulations. For example, given a partition level "N_{side}", we can obtain the corresponding ring index of a certain pixel, the world coordinate of the pixel center, the index number of the pixel, the starting pixel index of the ring, and so on. Thus, the indexing and searching-related operations can be implemented with these HEALPix APIs.

3.3 CPU-based, multi-thread pre-ordering

As shown in Section 1, single-level lookup table makes it possible to parallelize the gridding, thus improving the efficiency in evaluating effective data samples (Winkel et al. 2016b). However, in fact, FAST integrates the results of multiple observations into a large sky area and then performs gridding operations. This strategy could lead to a huge lookup table for sufficiently high resolution of the final data cube. In this case, the single-level table could become one of the bottlenecks of gridding performance improvements and is not applicable for FAST surveys.

In this work, we adopt the CPU-based multi-thread pre-ordering algorithm and a two-level lookup table scheme proposed by our previous work (Luo et al. 2018), to improve the efficiency of sample points searching, with less storage required. The two-level lookup table, based on latitudinal rings, can implement flexible conversion between the ring index and pixel index. Fig. 3 shows the pre-ordering algorithm and the process of constructing the two level lookup table. The details are as follows:

I: ***Pre-ordering of raw data:*** We performed pre-ordering of the raw data based on pixel indices of the HEALPix, in order to construct a two-level lookup table more conveniently. Firstly, given any partition level N_{side}, for any raw data $S_n \in \mathbb{S}$ with input coordinates (α_n, δ_n), one can get the pixel index $P_n \in \mathbb{P}$ to which each sampling point belongs, with the help of related HEALPix APIs. Secondly, we perform key-value sorting of P_n and n by non-descending order, with n as the subscript of data array P_n. Finally, we rearrange the raw data S_n as $\mathbb{S}'$ according to sorted n, with corresponding coordinate pairs (α_n, δ_n).

II: ***Construction of the first-level lookup table:*** As described above, $\mathbb{S}'$ and $\mathbb{P}'$ denote the sorted sampling points and the corresponding HEALPix pixels to which they belong. Considering the arrays $\mathbb{S}'$ and $\mathbb{P}'$ to be of the same size, for a given HEALPix pixel ($P_i \in \mathbb{P}'$), the corresponding starting sampled data point ($S_{start} \in \mathbb{S}'$) should meet the following criteria: 1) the corresponding HEALPix pixel index should equal to P_i; 2) the array index of the starting point is the smallest among all data that satisfies criteria 1). Based on such principles, we construct the first level lookup table based on the mapping of $P_i \mapsto S_{start}$.

III: ***Construction the second-level lookup table:*** According to the coding rule of HEALPix, the sampled data point with the same HEALPix ring index (R_i) should be stored in a contiguous segment of $\mathbb{S}'$. Naturally, we can obtain the minimum ring index R_{min}, as well as the maximum ring index R_{max} of $\mathbb{P}'$. Then we iterate over all the rings, searching for the starting pixel index (P_{start}) for each ring. Accordingly, we construct the second-level lookup table based on the mapping $R_i \mapsto P_{start}$, and symbolise it as $R_{start}[i]$.

3.4 Gather-based, multi-thread GPU gridding

The main concern of the GPU gridding process is to determine the contributing points that falling into the radius of influence by the gridding kernel. Such data are considered to be contributing to the target output grid cells. And then for each target output grid cell, the algorithm traverses the corresponding contributing input data, and performs the convolutional computations. Fig. 4 shows the searching process of the contributing points based on the two-level lookup table described in Section 3.3. The detailed steps are listed as follows:

(i) Based on the kernel size of the convolution kernel, compute the minimum and maximum ring index R_{min} and R_{max} within the range of influence of the convolution kernel.
(ii) Iterate over each latitude rings $R_i \in [R_{min}, R_{max}]$, and search for the corresponding starting contribute pixel in $\mathbb{P}'$ with array index falls within range of $[R_{start}[i],\ R_{start}[i+1]-1]$.
(iii) Search for starting contributing data sample S_{start} within each contributing HEALPix pixel, with the help of the first-level lookup table.
(iv) Traverse all the contributing data samples located within R_i in $\mathbb{S}'$, starting with S_{start}.
(v) For each target grid, perform the convolutional computations utilising the contributing points that belong to it.

3.5 The GPU-optimization strategy

We implemented GPU gridding in their native programming environment, CUDA for NVIDIA GPUs, with a runtime system and a set of C/C++ extensions. Compared with CPU, GPU usually boast a much higher amount of computing cores, in the form of Streaming Processors (SP), which can be further organized into small groups (Streaming Multiprocessors, SMs). SM is the core of the GPU architecture, the resource of the SM determined the performance of the GPU when using different thread management schemes. The optimization of computing performance for NVIDIA GPU includes two sides: thread-based optimization and memory-based optimization.

I: Thread management: NVIDIA put forward the concept of the hierarchical structure of thread, to facilitate the organizing of the thread. The hierarchical structure is a two-level thread hierarchy, consisting of thread block and thread grid (Sanders & Kandrot 2010). CUDA can organize three-dimensional grids and blocks, Fig. 5 is an example of a thread hierarchy (Cheng et al. 2014), which structure is a two-dimensional grid containing two-dimensional blocks. And the dimensions of the grids and blocks are determined by the built-in variables gridDim and blodkDim, respectively. Each component of the built-in variable can be obtained through its x, y, z fields, for example, blockDim.x, blockDim.y, blockDim.z. When a function is executed by threads in the thread grid level, the GPU spawns a set of grid consisting of thread blocks (both are user-specified dimensions), and dispatches the thread blocks onto SMs (Veenboer et al. 2017). The threads in the thread blocks can be further divided as thread warps, which are the basic units of executions on SMs.

For any given data volume, the general steps to determine the grid and block sizes are first to determine the size of the blocks, and then compute the size of the grids based on the data volume and block sizes. The sizes of the block are determined by the properties of the kernel function and the resources of GPU. Here we propose a thread organising and coarsening strategy for thread management, in order to improve the performance of multi-thread GPU gridding. The details are shown as follows:

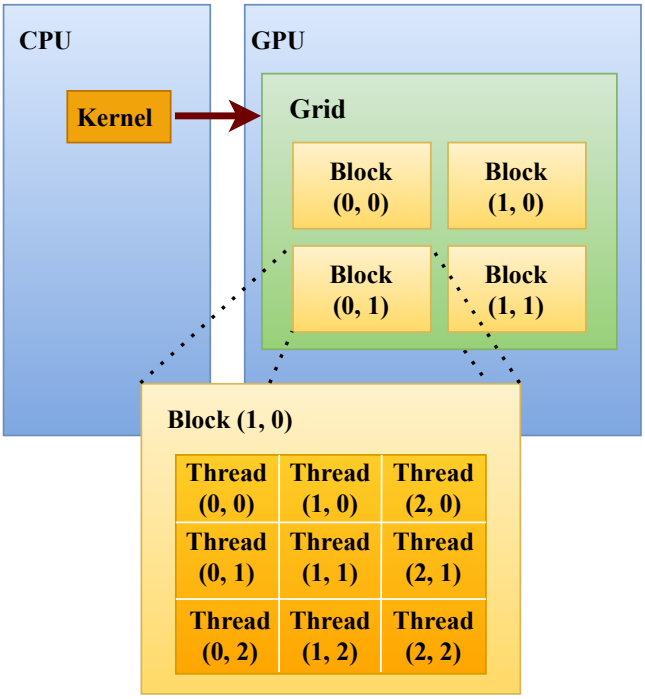

Figure 5. The basic structure of the CUDA two-level thread hierarchy. This example includes several two-dimension blocks nested in a two-dimensional grid. The dimensions of the grid and the block are determined by the built-in variables $gridDim$ and $blodkDim$, respectively.

(i) Thread organization: We analyzed the effect of thread organization on SM executing efficiency. Since the input data samples located in the HEALPix rings are stored as a one-dimensional array, the thread hierarchy we designed is consisted of one-dimensional grid and one-dimensional block, that is, gridDim.y = gridDim.z = 1, blockDim.y = blockDim.z = 1. To improve the executing efficiency of thread warps, the thread block will only allocate threads on X direction, and each thread within one block is responsible for the computing task of each target output grid cell located on the same latitude, respectively. Then the optimal number of threads in each block for the best performance is established according to various factors, including target output resolution, GPU architecture, and so on. We will make a detailed analysis of the thread organisation strategy in Section 4.2.
(ii) Thread coarsening: For a higher output resolution, it is necessary to start a larger number of threads. However, due to limited available resources, not all threads can be executed on GPU simultaneously. Through analysis, we found that adjacent target output grid cells on the same latitudinal ring largely share the same set of contributing input samples. Therefore, we apply the thread coarsening technique, which uses one thread only to calculate γ consecutive target output pixel on the same latitudinal ring, with γ as the thread coarsening factor. With a given factor γ, each thread only needs to perform GPU gridding once for consecutive, adjacent γ target pixels that this thread is responsible for. In this scheme, all these target pixels share the same starting contributing input data samples, with weightings of each contributing input sample and the sum of all the weights of each target pixel stored in different registers.

II: Memory management:
The performance of the kernel function (here is the gridding computing on GPU) is not only depended on the execution of the

thread warps, but also related to the memory access mode of CUDA. The CUDA memory model has proposed a variety of programmable memories, including register, shared memory, local memory, and so on. Fig. 6 shows the hierarchy structure of the CUDA memory, from the picture, one can conclude that different memory mode has different scope in the thread grid.

In our work, we utilise different types of CUDA memories for HCGrid's data storage. Optimization strategies include:

(i) The raw data, gridding results and the first-level lookup table are stored in the global memory;
(ii) Texture memory is a type of global memory accessed through specially designated, read-only cache. Data in the texture memory are stored globally, which means can be accessed by all threads, in the form of one-, two, or three-dimensional array. The hardware design of the cache here guarantees high speed access of data in texture memory. Generally, such memories are suitable for the situation of image processing o lookup tables utilised. Thus, the second-level lookup table is stored in an array structure in one-dimensional texture memory, to speed up data access.
(iii) Constant memory achieves the best performance when all threads in the warps read data from the same memory address. Considering all threads in one warp use the same set of gridding parameters, e.g., kernel size, output resolution, etc., to perform the same computations with different data, we choose to store such parameters in the constant memory.
(iv) In every thread, for a given output grid cell, the weight and the kernel-weighted value ("local output") of each contributing input data point should be stored one by one with designated arrays, only to be summed up to get the final cell reading once all contributing samples have been computed. We utilise the register to store these local output data, thus reducing the global memory access.

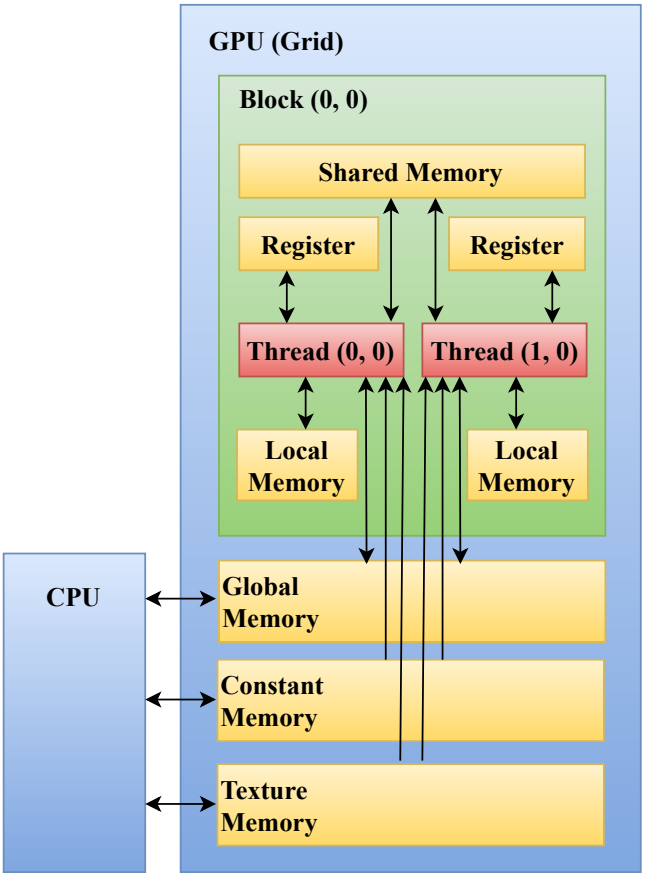

Figure 6. The hierarchy of CUDA memory. All threads can access global memory, the constants memory and the texture memory are the read-only memory that the threads can access. **Picture source: (Cheng et al. 2014), Fig. 4-2**

3.6 The basic scheme of HCGrid

In this section, we present the details of implementations of HCGrid. The whole gridding process begins with loading raw samples, as well as designated target grid into the framework. Then the relevant parameters of the convolution kernel should be set, and the input data be pre-sorted with CPU. Finally, the GPU-based convolutional computings are performed.

(i) **Loading raw data:** HCGrid requires observed data and corresponding coordinates pairs as input. In our preliminary tests, we adopt the original FITS files recorded by FAST as input data format, and transition into HDF5 (Folk et al. 2011) data format is still in progress, since the FAST sky survey pipeline utilise HDF5 as intermediate format (Ji et al. 2019).
(ii) **Reading target output map:** The most suitable size of the target grid is determined according to the area of the target sky coverage, as well as the beam width of the telescope. Those parameters are input as settings for empty target grid of HCGrid.
(iii) **Initialisation of convolutional kernel:** For the specific application scenarios, the shape of the convolution kernel can lead up to different results of the gridding process, or even affect the gridding performance. O'sullivan (1985) noted that the optimal convolutional kernel should be in the form of a sinc function with infinite length. However, such a function can bring huge computational loads. Thus, convolutional functions with finite lengths are often adopted, instead infinite ones. In this work, the Gaussian function is adopted as our default kernel. The parameter initialisation of the kernel function mainly follows a previous relevant work, Winkel et al. (2016b). Let $sphere\_radius$ be the range of influence of the convolutional kernel (rather than its half-width), it is obvious that more accurate results can be obtained with a larger $sphere\_radius$. However, in case of large amount of data, a larger range can pose great challenges to computing resources. As suggested by Winkel et al. (2016b), with radially symmetric Gaussian function as the gridding kernel, usually the $sphere\_radius$ should be set as some value between $3\sigma_{kernel} - 5\sigma_{kernel}$, depending on desired resolution, where σ_{kernel} is the standard deviation of the kernel function. The value of σ_{kernel} should be set according to resolution of raw data. Assuming the resolution of the telescope to be σ_{data}, as shown in Winkel et al. (2016b), $\sigma_{kernel} \approx 0.5\sigma_{data}$ should be adopted to get reasonable output. In this case, the resolution of gridded data should be $\sigma_{data}^{gridded} \approx 1.12\sigma_{data}$. Here, we choose $\sigma_{data} \approx 2.'9$, which is the approximate beam size of FAST at $1.42GHz$. And the resolution of HEALPix grid $hpx\_max\_resolution$ is directly related to σ_{kernel}. For example, if $sphere\_radius$ is set to be $3\sigma_{kernel}$, $hpx\_max\_resolution$ should be chosen as $\sim \sigma_{kernel}/2$.
(iv) **Pre-ordering of raw data:** In our Project, HCGrid provide four interfaces for raw data pro-ordering, including the C++ STL sort, parallel_stable_sort and block_indirect_sort, both based on

Boost[2], and sort_by_key of Thrust[3]. The first three are sorting interfaces for CPUs, while the last one performs GPU sortings. Through comparative analysis, it is found that block_indirect_sort can provide the highest capability, thus making it the default choice of pre-ordering interface.

(v) **GPU-based convolutional computation:** As mentioned in Section 3.5, we optimise the performance of GPU-based convolution computing from two aspects, thread as well as memory managements. Detailed analysis of such strategies will be analysed in Section 4.2.

4 EXPERIMENTS AND RESULTS

In this section, we perform benchmark analysis of HCGrid. At present, since the calibration and RFI flagging Yang et al. (2020) codes for the FAST spectral line data reduction pipeline is still in developments, it is difficult to conduct a full-scale test of HCGrid with observed data only. Thus, in this work, we mainly use simulated data to demonstrate code performance, with a small amount of real data received by FAST to show HCGrid gridding output. We generate simulated data in the similar way as Winkel et al. (2016b), that is, one Gaussian-distributed random value is assigned to each evenly-distributed random position n with coordinates (α_n, δ_n) in a pre-defined field of view.

4.1 Experimental setup

As listed in Table 1, we carried out our experiments on three workstations with different GPU hardware architectures. These workstations can be referred to as K40 (hosting an NVIDIA Tesla K40 GPU), T4 (equipped with an NVIDIA Tesla T4 GPU), and V100 (with an NVIDIA Tesla V100 GPU). We have adopted CUDA 8.0.61, GPU driver version 390.116 for K40; CUDA 8.0.61, GPU driver version 440.33.01 for T4; and CUDA 8.0.61 with GPU driver version 440.33.01 for V100.

To achieve performance optimization, we made a full consideration of the resource allocation under different GPU architecture, thus reasonably allocate resources to achieve relatively better performance.

Details of our **HCGrid** performance analysis include:

(i) We analyse the effects of the thread organization strategy on gridding performance, with the sampling space, the output resolution, and the amount of the input data fixed.

(ii) We analyse the effects of the thread coarsening factors on gridding performance at different output resolutions, with the sampling space coverage and the input data amount fixed.

(iii) We analyse the changing of HCGrid computing time on input data amounts, with the sampling space coverage and the resolution of output data remains fixed.

(iv) We analyse the change of HCGrid computing time on target grid coverage, with the input data amount and the resolution of output data remain fixed.

(v) We make a relative comparison between the Cygrid and HCGrid to illustrate the advantage of the GPU on the gridding computing.

(vi) We apply HCGrid on HI sky survey data obtained by FAST, with gridding results presented.

[2] https://www.boost.org/
[3] https://thrust.github.io/

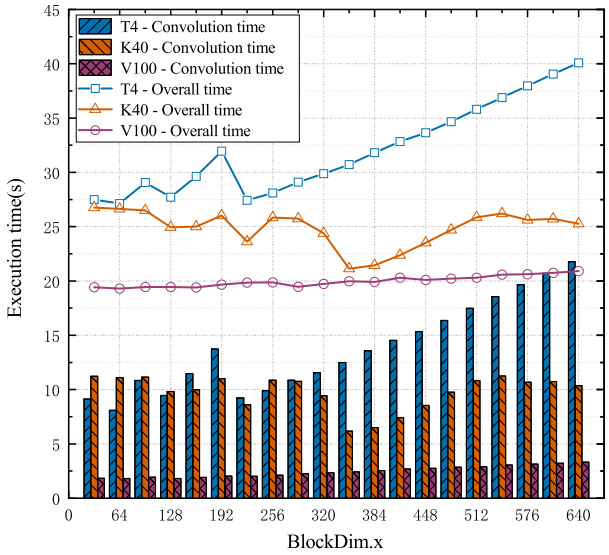

Figure 7. Benchmark results of thread organization strategy. With 10^8 input samples, and the thread coarsening factor $\gamma = 1$, the processing time of GPU convolution, as well as HCGrid processing time changes with different settings of thread organization strategy.

4.2 Performance analysis

1) Performance vs thread organization strategy: With a field size of $5^\circ \times 5^\circ$, the output resolutions of $\sigma_{grid} = 200''$(which is approximately the average beam size of the FAST across the observing band of its 19-beam receive), and input sample amount of $N = 10^8$, we implement various thread configurations with different GPU architectures by configuring the values of thread grids and thread blocks, to analyze the impact of the choice of thread organization strategies on the execution time of GPU convolution, as well as the overall execution time of HCGrid.

Take workstation K40 with a CUDA capability of 3.5 as an example. Here the total number of registers available for each block of is $64K$, while our compilation report shows that the core function of **HCGrid** utilizes 184 registers, without using shared memory to store parameters. Thus, it is expected that each thread block can execute $\sim 64K/184 \approx 356$ threads simultaneously. Fig. 7 shows that the execution time varies with thread organization methods. With the number of blocks along X-dimension $blockdim.x = 352$, and the number of grids in the same dimension ($griddim.x$) as 25, the fastest execution speed can be obtained. That is because, in this case, the value of $blockdim.x$ is close to the maximum thread count 356, thus enabling the full utilization of the computing capability of the K40 GPU.

And for workstation T4 with a CUDA capability of 7.5, the total number of registers available for each block is also $64K$. However, the number of SPs in T4 is 2560, which is smaller than K40's 2880. That is, T4 can execute 2560 threads at most at the same time. Partially due to this reason, the execution time of HCGrid on T4 is longer than that of K40. And since each SM of T4 has 64 SPs, one SM of T4 can only execute 64 threads simultaneously. Thus, if we allocate $blockdim.x = 352$ as we did to K40, the thread waiting should be expected, and thread scheduling overhead could be increased to some extent. Therefore, experiments show that the best computing performance can be obtained for $blockdim.x = 64$. In this configuration, it is possible to achieve maximum thread parallelization for each SM with minimal thread scheduling overhead.

Table 1. The configurations of the workstation for experiments

Model	Type	Architecture	Clock (GHz)	Cores	SMs	Mem size (GB)	Mem bw (GB/s)	CUDA Capability
Intel Xeon E5-2620	CPU	Sandy-EP	2.4	6	-	32	-	-
NVIDIA Tesla K40	GPU	Kepler	0.745	2880	15	12	288.4	3.5
Intel Xeon Platinum 8255C	CPU	Cascade Lake	2.5	8	-	32	-	-
NVIDIA Tesla T4	GPU	Turing	0.585	2560	40	16	320	7.5
Intel Xeon Silver 4114	CPU	Skylake-EP	2.2	16	-	32	-	-
NVIDIA Tesla V100	GPU	Volta	1.246	5120	80	16	897.0	7.0

Similarly, for workstation V100 with CUDA capability of 7.0, and the largest SP among all our workstations, a prominent advantage in terms of thread organization, with the shortest GPU convolution time can be achieved. And although the architecture of V100 is different from T4, the number of SPs in each SM of these two GPUs are the same. Thus, the thread organization strategies for T4 can be adopted for V100, with $blockdim.x$ remains to be 64 for the best performance. It can be seen in Fig. 7 that our conclusions have been confirmed.

Based on the analysis above, for the thread organization configuration, the architecture of the GPU and the number of SPs in the SM should be carefully adjusted, in order to select the most appropriate scheme to improve the performance of GPU parallelization. For mainstream GPU architectures by NVIDIA, including Turing, Volta, Pascal, Kepler, Fermi, and Maxwell, the minimum number of SPs in each SM equals to 32 (for Fermi architecture). Thus, when taking thread configuration into consideration only, we get the empirical Eq. (3) as follows:

$$T_{max} = (Register\_num)/184 \tag{2}$$

$$blockdim.x = \begin{cases} SP & 32 \le SP < \frac{1}{2}T_{max} \\ T_{max} & SP \ge \frac{1}{2}T_{max} \\ other & Based\ on\ actual\ test\ results \end{cases} \tag{3}$$

In Eq. (2), $Register\_num$ represents the total number of registers available for each thread block of the GPU, and T_{max} is the maximum number of threads that each thread block can execute simultaneously when running HCGrid. In Eq. (3), SP is the number of SPs in each SM of the GPU. For example, for K40, $T_{max} = 356$, and the number of SPs in each SM is 192, which is greater than $\frac{1}{2}T_{max}$. Thus, the value of $blockdim.x$ should be set as 352. However, since thread configuration may also be affected by various factors, such as GPU clock frequency and memory bandwidth, the GPU thread organization parameters should be adjusted according to the specific computing environment to get optimized results.

2) Performance vs of thread coarsening strategy: when make thread coarsening experiment on K40 workstations, Fig. 8 shows the execution time for convolution with different thread coarsening factors (γ = 1,2,3) at different output grid resolutions. The relative performance improvements for each run relative to the processing time for the $\gamma = 1$ case are also shown in the same figure. It can be seen that the finner the output grid resolution, the more improvement of gridding performance can be obtained with thread coarsening.

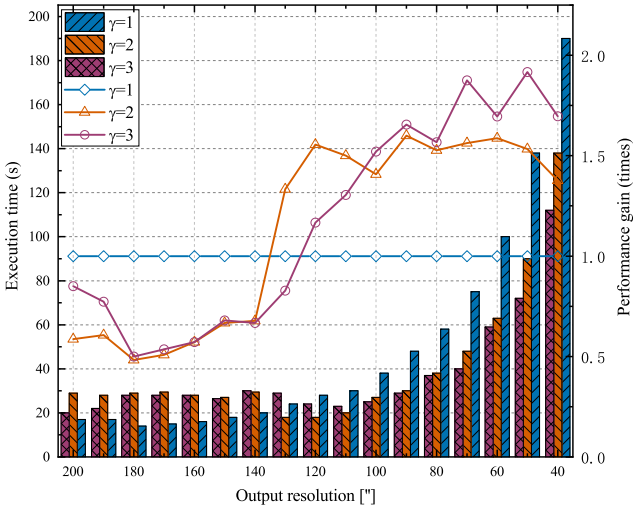

Figure 8. The processing time and performance gain of the convolution part on GPU. The histogram represents the execution time, and the curves show the performance gain. In our calculations, 10^8 samples are gridded onto a target field of $5^\circ \times 5^\circ$, with different thread coarsening factor γ.

When starting a large number of threads, on the one hand, it is possible that not all threads can be run concurrently on the GPU, due to limited resources available. On the other hand, in this case, the workload of thread scheduling would be increased, resulting in performance degradations. Thus, thread coarsening technique is often applied, with multiple input samples handled by one thread only. As shown in Fig. 8, With γ = 2 or 3, the number of starting threads is reduced, hence the scheduling workload can be reduced, which can lead to better gridding performance as a result.

We have also performed thread coarsening experiments on T4 and V100 workstations, with similar results to K40 obtained. Suggestions for thread coarsening parameter settings are listed as follows:

(i) With $150'' \le \sigma_{data} \le 200''$, it is reasonable to set $\gamma = 1$ for better computing performance;
(ii) If $110'' \le \sigma_{data} \le 140''$, a more reasonable choice should be $\gamma = 2$;
(iii) $\gamma = 3$ works better for $40'' \le \sigma_{data} \le 100''$.

3) Performance vs input data volume: With a sampling space

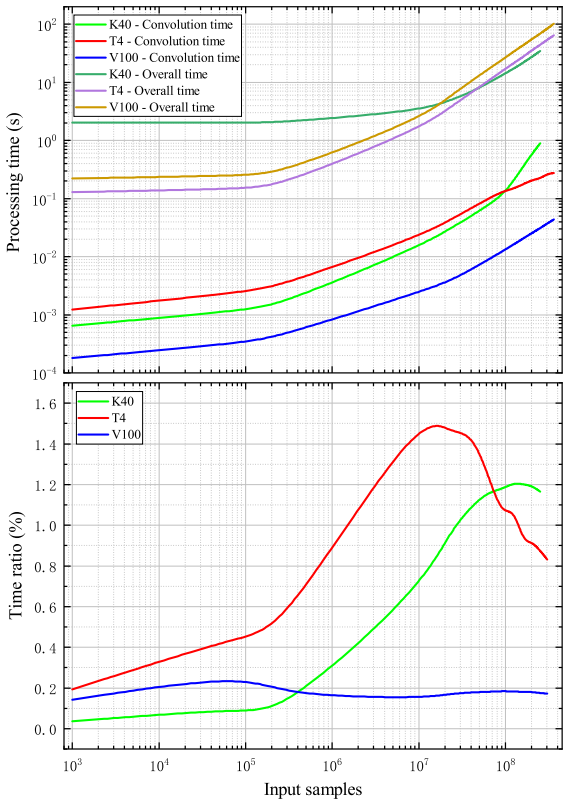

Figure 9. Results of computational complexity analysis. The top panel shows the HCGrid processing time for different input data volume, while the lower one presents the ratio of time for convolutional computations to the whole executing time.

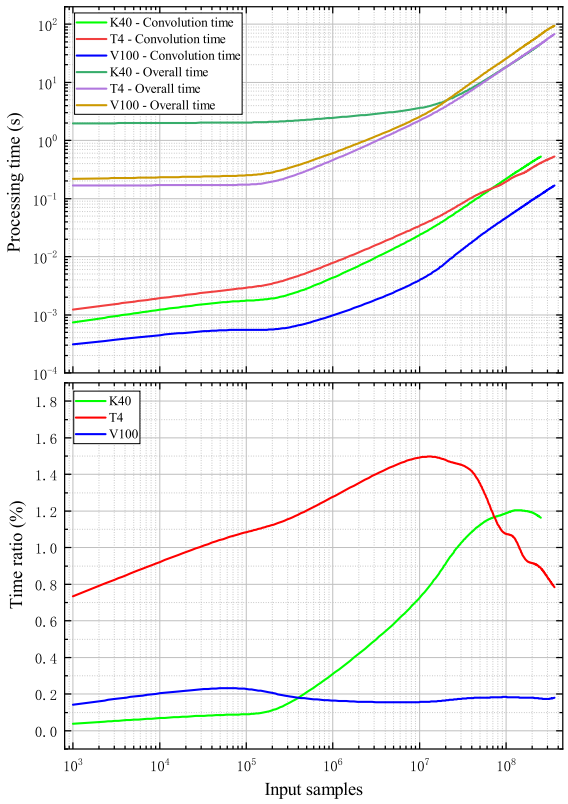

Figure 10. Analysis of HCGrid performance in relationship with field size. The filed size varies from $0.1° \times 0.1°$ to $60° \times 60°$, with input sample density as 10^5 per square degree.

of $5° \times 5°$, and a target resolution of $\sigma_{grid} = 200''$, we have $I \times J = (5°/200'') \times (5°/200'') = 90 \times 90$. The processing time of GPU convolution and the overall computing time of HCGrid on three workstations with different configurations for input data with $10^3 \sim 3.6 \times 10^8$ samples are shown in the top panel of Fig. 9. It can be seen that the processing time of HCGrid increases quasi-linearly with the input data size. If the number of input samples is with an order of magnitude less than 10^5, the HCGrid computational complexity is $O(1)$; while with samples larger than 10^5, the complexity changes to $O(n)$. Thus, it can be concluded that the computational complexity of HCGrid should lie somewhere between $O(1)$ and $O(n)$. And the lower panel of Fig. 9 presents the ratio of the convolution time to the whole processing time of HCGrid, with a highest reading as 1.5% for the workstation with T4 GPU. In one word, with the help of GPU, it is demonstrated that the convolution computing, which should be the most computationally intensive computing task in gridding process, is no longer the most time-consuming component for HCGrid.

4) Performance vs filed size: To further analyze the impact of different field sizes, we conduct similar experiments, as shown in Fig. 9. Here the field size varies from $0.1° \times 0.1°$ to $60° \times 60°$, with the input data amount as 10^5 samples per square degree. As depicts in Fig. 10, the result is similar to Fig. 9, which demonstrated that the processing time of HCGrid is mostly influenced by input data volume, with limited relevance to the sizes of the field.

4.3 Comparison with Cygrid

As mentioned above, Cygrid is a high-performance gridding implementation for CPUs. By comparison, the GPU-implemented HCGrid proposed by this work improves the hardware utilisation, adding with the introducing of the pre-sorting operation of the input samples.This section gives a relative comparison of HCGrid and Cygrid performance, in order to demonstrate the advantage of GPU for gridding operations.

The grid coverage on the target field for this experiment is $5° \times 5°$, with a grid resolution of $\sigma_{grid} = 200''$, and the gridding kernel width of $\vartheta_{fwhm} = 300''$. The input data volume varies from $10^3 \sim 3.6 \times 10^8$ simulated samples. The Cygrid is executed on the V100 workstation with 16 processor cores, while HCGrid is deployed on the same computing environment with GPU exploited. Fig. 11 compared the executing time of Cygrid and HCGrid, it can be seen that the computational complexity of Cygrid, which is closer to $O(n)$ (Winkel et al. 2016b), has been verified by the trend in the corresponding curve. And with the number of input samples exceeding 10^5, the executing speed of HCGrid is several times faster than Cygrid. However, when dealing with smaller batches of input, data transmissions between CPU and GPU cause notable degradation in performance of HCGrid, resulting in a slower speed than Cygrid. In case of larger amount of input data, the performance improvements due to GPU-implemented convolutions far exceeds the overhead of such data transmissions, leading to significantly better overall performance of HCGrid.

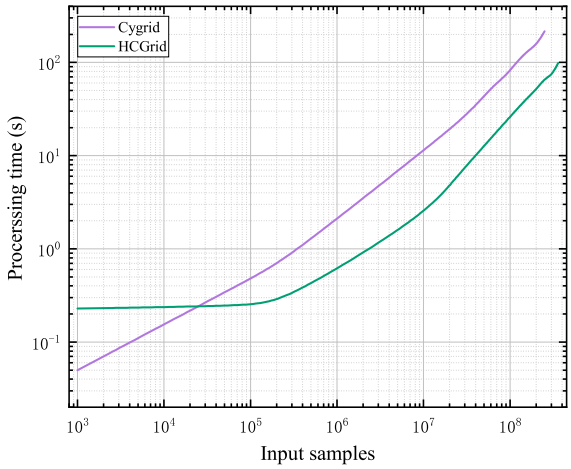

Figure 11. Comparison of Cygrid and HCGrid. The input data volume ranged from $10^3 \sim 3.6\times10^8$ sample. Here the executing platform of Cygrid is 16 processor cores of the V100 workstation, while HCGrid is deployed on the same workstation, utilising its GPU.

5 CONCLUSIONS

In this paper, we introduce a convolution-based gridding framework **HCGrid** for radio astronomy in hybrid computing environments, in order to meet the requirements of large single-dish radio telescopes such as FAST. **HCGrid** features the first implementation of convolution-based gridding with gather strategy for CPU-GPU heterogeneous platform.

To increase the searching efficiency of raw input samples, **HCGrid** makes full use of advantages of both CPU and GPU. It implemented a fast parallel ordering algorithm based on HEALPix on CPU, established a two-level lookup table to speed up the sample searching process, and accelerated the convolution operation using GPU with several optimization strategies.

To ensure the scalability of HCGrid, we have performed experiments on three workstations with various GPU architectures. All of the performance optimization strategies have been tested on all three architectures, with general guidance for related performance parameter optimizations presented. And compared with CPU implementations of gridding process, HCGrid has the performance advantage due to GPU-based carried out the time-consuming convolution computing on the GPU.

Further improvements of the **HCGrid** is expected, as minor bugs being further tested and removed, and related user manual being continuously updated. Also, the time-consuming data transmissions between CPU and GPU, as noted in Section 4.3, is under further investigations with various memthods, including the CUDA streaming technology, which will improve the transmission efficiency. And it is still underway for the final integration of HCGrid to the data reduction pipeline of FAST telescope.

ACKNOWLEDGEMENTS

We thank Mr. Benjamin Winkel for providing the Cython code of Cygrid. Also, The results of this work is partially derived with the HEALPix library (gorski2005healpix). We would also like to express our gratitude to the Python/C++ open source developers of Numpy (Walt et al. 2011), Scipy (Jones et al. 2001) and Matplotlib (Hunter 2007), we use these tools in our work. This work is supported by the Joint Research Fund in Astronomy [grant number U1731125, U1731243] under a cooperative agreement between the National Natural Science Foundation of China (NSFC) and Chinese Academy of Sciences, NSFC grant No. 11903056, as well as the Open Project Program of the Key Laboratory of FAST, NAOC, Chinese Academy of Sciences.

REFERENCES

Calabretta M. R., Greisen E. W., 2002, Astronomy & Astrophysics, 395, 1077
Cheng J., Grossman M., McKercher T., 2014, Professional Cuda C Programming. John Wiley & Sons
Folk M., Heber G., Koziol Q., Pourmal E., Robinson D., 2011, in Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases. pp 36–47
Giovanelli R., et al., 2005, The astronomical journal, 130, 2598
Górski K. M., Hivon E., Banday A. J., Wandelt B. D., Hansen F. K., Reinecke M., Bartelmann M., 2005, The Astrophysical Journal, 622, 759
Ji Y., Yu C., Xiao J., Tang S., Wang H., Zhang B., 2019, in Wen S., Zomaya A. Y., Yang L. T., eds, Lecture Notes in Computer Science Vol. 11945, Algorithms and Architectures for Parallel Processing - 19th International Conference, ICA3PP 2019, Melbourne, VIC, Australia, December 9-11, 2019, Proceedings, Part II. Springer, pp 656–672, doi:10.1007/978-3-030-38961-1_55, https://doi.org/10.1007/978-3-030-38961-1_55
Kalberla P. M., Burton W., Hartmann D., Arnal E. M., Bajaja E., Morras R., Pöppel W., 2005, Astronomy & Astrophysics, 440, 775
Kalberla P., et al., 2010, Astronomy & Astrophysics, 521, A17
Léna P., Rouan D., Lebrun F., Mignard F., Pelat D., 2012, Observational Astrophysics. Springer Science & Business Media
Li D., et al., 2018, IEEE Microwave Magazine, 19, 112
Luo Q., Xiao J., Yu C., Bi C., Ji Y., Sun J., Zhang B., Wang H., 2018, in International Conference on Algorithms and Architectures for Parallel Processing. pp 621–635
McCool M., Reinders J., Robison A., 2012, Structured parallel programming: patterns for efficient computation. Elsevier
Merry B., 2016, Astronomy and Computing, 16, 140
Mink D., 2006, in Astronomical Data Analysis Software and Systems XV. p. 204
Nan R., 2006, Science in China series G, 49, 129
Nan R., et al., 2011, International Journal of Modern Physics D, 20, 989
O'sullivan J., 1985, IEEE transactions on medical imaging, 4, 200
Romein J. W., 2012, in Proceedings of the 26th ACM international conference on Supercomputing. pp 321–330
Sanders J., Kandrot E., 2010, CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional
Schweizer H., Besta M., Hoefler T., 2015, in 2015 International Conference on Parallel Architecture and Compilation (PACT). pp 445–456
Smith S. L., Dunning A., Smart K. W., Shaw R., Mackay S., Bowen M., Hayman D., 2017, in 2017 IEEE International Symposium on Antennas and Propagation & USNC/URSI National Radio Science Meeting. pp 2137–2138
Veenboer B., Petschow M., Romein J. W., 2017, in 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp 545–554
Wells D. C., Greisen E. W., 1979, in Image Processing in Astronomy. p. 445
Winkel B., Kerp J., Flöer L., Kalberla P., Bekhti N. B., Keller R., Lenz D., 2016a, Astronomy & Astrophysics, 585, A41
Winkel B., Lenz D., Flöer L., 2016b, Astronomy & Astrophysics, 591, A12
Yang Z., Yu C., Xiao J., Zhang B., 2020, Monthly Notices of the Royal Astronomical Society, 492, 1421
van Amesfoort A. S., Varbanescu A. L., Sips H. J., Van Nieuwpoort R. V., 2009, in Proceedings of the 6th ACM conference on Computing frontiers. pp 207–216

This paper has been typeset from a TEX/LATEX file prepared by the author.