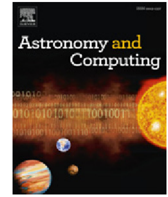
Astronomy and Computing 16 (2016) 140–145

Contents lists available at ScienceDirect

Astronomy and Computing

journal homepage: www.elsevier.com/locate/ascom



Full length article

Faster GPU-based convolutional gridding via thread coarsening

B. Merry

SKA South Africa, 3rd Floor, The Park, Park Road, 7405, South Africa

ARTICLE INFO

Article history:
Received 23 January 2016
Accepted 18 May 2016
Available online 31 May 2016

Keywords:
Techniques: interferometric
Methods: numerical
Computing methodologies: graphics processors

ABSTRACT

Convolutional gridding is a processor-intensive step in interferometric imaging. While it is possible to use graphics processing units (GPUs) to accelerate this operation, existing methods use only a fraction of the available flops. We apply thread coarsening to improve the efficiency of an existing algorithm, and observe performance gains of up to 3.2× for single-polarization gridding and 1.9× for quad-polarization gridding on a GeForce GTX 980, and smaller but still significant gains on a Radeon R9 290X.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Interferometric imaging is a key tool in radio astronomy, but as modern instruments provide more antennas, longer baselines, and more channels, it is becoming increasingly computationally costly. A major component of an imaging pipeline is *convolutional gridding*, as well as the corresponding degridding for predicting visibilities.

Given the computational cost of gridding, it is natural to apply accelerator hardware, of which the cheapest and most ubiquitous is the Graphics Processing Unit (GPU). However, the irregular data access patterns make this a non-trivial task. One of the first really practical algorithms for GPU-accelerated gridding is due to Romein (2012). Despite being state of the art, it typically spends only about 25% of a GPU's compute power on the actual convolution operations. There are bottlenecks in the memory system, but also computational overheads associated with address calculations. Our goal is to reduce these overheads to make more flops available for the convolution calculations.

Our contribution is a modification to the algorithm in which each thread of execution processes multiple elements of the grid. This is a standard transformation called *thread coarsening*, but which we have adapted to this specific problem. This allows some overheads to be amortized across multiple grid elements, thus increasing performance.

2. Background

2.1. Graphics processing units

While originally designed for computer graphics, GPUs have become a common and accessible approach to accelerating general-purpose computations. Here we provide only a brief introduction to GPU architecture; a complete discussion is beyond the scope of this paper. Two common APIs used to program GPUs are CUDA (a proprietary standard from NVIDIA), and OpenCL (a cross-vendor standard that is also applicable to CPUs and FPGAs). We will use the OpenCL terminology as it is more generic, although our implementation runs on both CUDA and OpenCL. For readers more familiar with CUDA, substitute thread for work-item, thread-block for work-group, grid for kernel-instance, shared memory for local memory, and streaming multiprocessor for compute unit.

OpenCL works on a single-program multiple-data model. A single program, called a *kernel*, is executed many times in parallel. Each execution is a *work-item*. Work-items are arranged into *work-groups*. The work-items of a work-group are guaranteed to execute concurrently, and can synchronize and communicate with each other. The set of all work-items launched at one time is called a *kernel-instance*. GPUs comprise multiple *compute units* which operate largely independently, each with their own schedulers, L1 caches, register file and execution units—similar to CPU cores. Each work-group is assigned to one compute unit, but a compute unit can run multiple work-groups concurrently.

GPUs also have multiple memory systems. The slowest, largest memory is *global memory*, which is generally off-chip DRAM. There are usually also several levels of cache for this global memory. *Local*

E-mail address: bmerry@ska.ac.za.

http://dx.doi.org/10.1016/j.ascom.2016.05.004
2213-1337/© 2016 Elsevier B.V. All rights reserved.

memory is fast on-chip memory local to a compute unit, which can be used for work-items in a work-group to communicate with each other, and is also used as a software-managed cache. The fastest memory is registers, which are local to a work-item. There are other special-purpose memory types, but they are not relevant here.

2.2. Convolutional gridding

Consider the full-Sky radio interferometry measurement equation (RIME) (Smirnov, 2011, eq 17):

$$K_{pq} = e^{-2\pi i(u_{pq}l+v_{pq}m+w_{pq}(n-1))}$$
$$V_{pq} = G_p\left(\iint_{lm} \frac{1}{n} K_{pq}E_pBE_q^H\, dl\, dm\right) G_q^H. \tag{1}$$

Here, l, m, n are direction cosines parameterizing the sky, (u_{pq}, v_{pq}, w_{pq}) is the baseline vector between antennas p and q, B is the brightness matrix at (l, m, n), E_p is a Jones matrix for direction-dependent effects, G_p is a Jones matrix for direction-independent effects, and V_{pq} is the predicted visibility.

With the exception of the $w_{pq}(n-1)$ term in the exponent, this is a Fourier transform relationship between visibilities and the sky. Evaluating or inverting the RIME directly is prohibitively expensive, so it is typically done using fast Fourier transforms (FFTs) (Cooley and Tukey, 1965). However, visibilities are not sampled on a regular grid, so an extra *gridding* step must be taken to generate such a grid before using the FFT to produce an image.

Simply snapping each visibility sample to the nearest point on the grid would cause severe artifacts, particularly aliasing. Instead, each visibility sample is treated as a Dirac delta, convolved with some function, and then sampled onto the grid. Convolution in visibility space is equivalent to multiplication in image space, so using a function with bounded support in image space provides antialiasing (Greisen, 1979). The $e^{w_{pq}(n-1)}$ and E_p terms can also be handled by convolution in visibility space—these are known as W-projection (Cornwell et al., 2008) and A-projection (Bhatnagar et al., 2006) respectively.

The gridding convolution function (GCF) cannot always be computed analytically, and even when it can, it is usually expensive to do so. Thus, tables of GCFs are normally precomputed numerically. To reduce aliasing, the GCF needs to be sampled at a higher resolution than the grid itself. A typical value is 8× oversampling (Romein, 2012), but this will depend on how far from the field of view one expects to find contaminating signals.

Efficient gridding on a GPU is challenging because the problem has irregular structure, with the memory accesses depending on the uvw coordinates. There is plenty of parallelism, but multiple visibilities will contribute to each grid point and so there are data hazards. A naïve implementation will also be totally memory-bound: multiplying two single-precision complex numbers and accumulating the result into memory requires 8 flops and 16 bytes of memory traffic, while typical desktop GPUs can have compute-to-bandwidth ratios of 15–20 flops per byte.

Romein (2012) introduced the first reasonably efficient GPU-accelerated gridding algorithm. It takes advantage of the spatial coherence of the data to reduce memory bandwidth. For a single baseline and frequency, the UV-plane positions move slowly over time as the Earth rotates. Similarly, moving to an adjacent frequency bin involves a small shift in the UV plane. Thus, if one iterates over the visibilities for a single baseline, the GCF footprints will almost entirely overlap. This makes it possible to maintain sums in registers which are only occasionally flushed to global memory.

Fig. 1 shows how the algorithm works. The grid is divided into *bins*, which are at least as large as the GCF—in the original algorithm, they are the same size. A work-item is responsible for all the positions in the grid that have the same relative placement within a bin, e.g., all the grid positions marked with a dot are the responsibility of one work-item. A bin-sized bounding box is placed around the GCF footprint for one visibility, which will contain exactly one grid-point per work-item. Each work-item maintains an in-register accumulator for that grid point. When the bounding box moves, some work-items will switch to a different grid point: when this happens, those work-items flush their accumulator to global memory using an atomic addition. If the bounding box moves by one grid point, then only $O(N)$ atomic updates are made for an $N \times N$ GCF, thus greatly reducing the memory traffic.

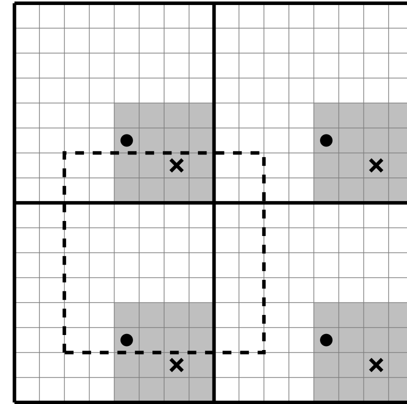


Fig. 1. Overview of Romein's gridding algorithm. The dashed box shows a bounding box containing the GCF footprint. One work-item handles grid points marked with a dot; another handles those marked with a cross, and so on. The gray box indicates a tile: once all the grid points in a tile have been handled, the same work-items are recycled to update the next tile.

Coarse-grained parallelism is achieved by assigning each baseline to a separate work-group. Because these work-groups operate independently, they may potentially update the same grid points at the same time; this is why grid updates are done using atomic instructions.

A complication arises if the bins are too large to hold an entire bin in registers at once. In this case, each bin is split into *tiles* (Fig. 1 shows one tile in gray), and a work-group handles only one tile's-worth of work-items. Romein iterates serially over tiles within the GPU code: after a work-group has iterated over all visibilities in its baseline, it iterates over them again, but taking responsibility for the next tile. Our implementation is parallel rather than serial, using a separate work-group per tile. In either case, the number of atomic updates to the grid is unaffected by tile size, but visibilities and their coordinates are loaded from memory once for each tile in a bin.

Muscat (2014) noticed that it is not necessary to grid each visibility individually. In some cases, particularly for short baselines, two adjacent visibilities have the same position on the higher-resolution grid used to sample the GCF. This means that they will be multiplied by the same GCF samples, and thus they can be added together to form a single visibility. This yields identical results (up to floating-point precision) but reduces the number of visibilities to grid. He refers to this merging process as *compression*. We use compression in our implementation, and in our results we consider only the rate for gridding these compressed visibilities, rather than the original visibilities.

3. Thread coarsening

Thread coarsening is the process of merging multiple work-items (also known as threads) into one. This is similar to loop unrolling, but applied across parallel work-items rather than across serial loop iterations. This improves instruction-level parallelism

Download English Version:

https://daneshyari.com/en/article/497546

Download Persian Version:

https://daneshyari.com/article/497546

Daneshyari.com