GPU-Accelerated Convolutional Gridding for the SKA

SETH HALL
ADAM CAMPBELL
ANDREW ENSOR
Introduction

We have been looking at utilizing GPUs (Graphics Processing Units) to create an efficient convolutional gridding solution for the SKA imaging pipeline:

Our objectives:

- Use GPU to obtain highest possible data throughput.

- Maintain accuracy of gridded visibilities.

- Keep power consumption to a minimum.

- Bonus: A solution that will work across many different GPU models.
Basic Imaging Pipeline

**Input visibility data:**
Each visibility consists of $u,v,w$ coordinate and an intensity (complex)

**2D Output Gridded data:**
Each component in grid holds an accumulated intensity (complex) and weight in the Fourier domain
Challenges for SKA:

Grid size needed for SKA approximately 65536 x 65536 elements, 3-4 floats per element on grid.

Currently impossible to fit entire grid (roughly 52GB) into memory using latest GPU (currently 16GB maximum).

40-75 billion visibilities going into the gridder per second in SKA-1 (5 floats per visibility)

Transfer the resulting gridout of GPU to main memory after “dump time” takes a significant duration, especially with a large grid.

Need many pre-calculated w-planes, used for w-projection to smear visibility on grid at different oversample rates.

Our solution on a Titan X can fit an 18000 x 18000 element grid into GPU memory (so will need to either group visibilities into sub-grids keeping full scale grid off GPU memory or use a multi-GPU solution for current SKA requirements).

Currently using SDP synthesized data provided by Oxford University, which uses only contains approximately 32 million visibilities.
Each visibility is composed of a $u, v, w$ 3D coordinate, which represents the vector distance in wavelengths of the baseline antennae. The $u, v$ pair is converted to grid coordinates and smeared on the grid using a “w-projection” kernel, which is determined by the visibility’s $w$ value.

Each visibility also consists of a complex number, which measures the strength of cross correlation between baseline antennae.
Griddler

Visibility \( (5+0i, u=5, v=6, w=0) \).

Visibility strength is multiplied by kernel (convolution) and smeared on grid.

Applied w-kernel

<table>
<thead>
<tr>
<th></th>
<th>0.053</th>
<th>0.087</th>
<th>0.107</th>
<th>0.087</th>
<th>0.053</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.087</td>
<td>0.144</td>
<td>0.157</td>
<td>0.144</td>
<td>0.087</td>
<td></td>
</tr>
<tr>
<td>0.107</td>
<td>0.157</td>
<td>0.178</td>
<td>0.157</td>
<td>0.107</td>
<td></td>
</tr>
<tr>
<td>0.087</td>
<td>0.144</td>
<td>0.157</td>
<td>0.144</td>
<td>0.087</td>
<td></td>
</tr>
<tr>
<td>0.053</td>
<td>0.087</td>
<td>0.107</td>
<td>0.087</td>
<td>0.053</td>
<td></td>
</tr>
</tbody>
</table>
Visibility (10+0i, u=9,v=9,w=10).

Visibility strength is multiplied by kernel and smeared on grid and accumulated with previous visibility values.

<table>
<thead>
<tr>
<th>0.019</th>
<th>0.034</th>
<th>0.047</th>
<th>0.052</th>
<th>0.047</th>
<th>0.034</th>
<th>0.019</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.034</td>
<td>0.058</td>
<td>0.080</td>
<td>0.090</td>
<td>0.080</td>
<td>0.058</td>
<td>0.034</td>
</tr>
<tr>
<td>0.047</td>
<td>0.080</td>
<td>0.111</td>
<td>0.124</td>
<td>0.111</td>
<td>0.080</td>
<td>0.047</td>
</tr>
<tr>
<td>0.052</td>
<td>0.090</td>
<td>0.124</td>
<td>0.139</td>
<td>0.124</td>
<td>0.090</td>
<td>0.052</td>
</tr>
<tr>
<td>0.047</td>
<td>0.080</td>
<td>0.111</td>
<td>0.124</td>
<td>0.111</td>
<td>0.080</td>
<td>0.047</td>
</tr>
<tr>
<td>0.034</td>
<td>0.058</td>
<td>0.080</td>
<td>0.090</td>
<td>0.080</td>
<td>0.058</td>
<td>0.034</td>
</tr>
<tr>
<td>0.019</td>
<td>0.034</td>
<td>0.047</td>
<td>0.052</td>
<td>0.047</td>
<td>0.034</td>
<td>0.019</td>
</tr>
</tbody>
</table>

Overlapping kernels. Naïve approach to parallelization is to allocate threads to each visibility. However overlapping kernels could cause errors with concurrent updates of grid points.
Griddler

Visibility (3+0i, u=8.3, v=4, w=0)
In reality uv coordinates may not fall exactly on a grid point. So we need to create w-planes that are oversampled (values shifted in both x and y direction). The visibility is snapped to the nearest grid point and applied with the nearest oversampled kernel.

This is necessary as it is too computationally expensive to calculate entire w-projection kernel for each visibility.

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.19</td>
<td>0.34</td>
<td>0.47</td>
<td>0.52</td>
<td>0.47</td>
<td>0.34</td>
<td>0.19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.34</td>
<td>0.58</td>
<td>0.8</td>
<td>0.9</td>
<td>0.8</td>
<td>0.58</td>
<td>0.34</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.47</td>
<td>0.8</td>
<td>1.11</td>
<td>1.24</td>
<td>1.11</td>
<td>0.8</td>
<td>0.47</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.52</td>
<td>0.9</td>
<td>1.24</td>
<td>1.39</td>
<td>1.24</td>
<td>0.9</td>
<td>0.52</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.265</td>
<td>0.435</td>
<td>0.535</td>
<td>0.905</td>
<td>1.065</td>
<td>1.11</td>
<td>1.24</td>
<td>1.11</td>
<td>0.8</td>
<td>0.47</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.435</td>
<td>0.72</td>
<td>0.785</td>
<td>1.06</td>
<td>1.015</td>
<td>0.8</td>
<td>0.9</td>
<td>0.8</td>
<td>0.58</td>
<td>0.34</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.535</td>
<td>0.785</td>
<td>0.89</td>
<td>1.054</td>
<td>1.057</td>
<td>0.774</td>
<td>0.788</td>
<td>0.630</td>
<td>0.34</td>
<td>0.19</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.435</td>
<td>0.72</td>
<td>0.785</td>
<td>0.850</td>
<td>0.737</td>
<td>0.447</td>
<td>0.444</td>
<td>0.263</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.265</td>
<td>0.435</td>
<td>0.535</td>
<td>0.595</td>
<td>0.594</td>
<td>0.507</td>
<td>0.485</td>
<td>0.324</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.130</td>
<td>0.302</td>
<td>0.447</td>
<td>0.444</td>
<td>0.263</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.079</td>
<td>0.182</td>
<td>0.304</td>
<td>0.268</td>
<td>0.160</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Kernel values at 0.25 oversample, shifted right
Nowadays GPUs are also used for high performance computing problems.

A GPU has many cores each with hundreds of worker threads.

GPUs should only be used for compute-intensive tasks that can be broken up and run in parallel.

Threads on GPU work in *lock-step*, meaning they iterate through each GPU assigned code instruction simultaneously: Not good for conditional branching!

Images taken from Nvidia website
API for GPU

- NVIDIA CUDA – Designed for high performance computing, keeps underlying details of GPU “hidden”. Platform is a software layer that gives direct access to the GPU’s virtual instruction set and parallel computational elements for the execution of compute kernels. This accessibility makes it easier for specialists in parallel programming to use GPU resources, in contrast to prior APIs like Direct3D and OpenGL.

- OPEN CL - Framework for writing programs that execute across heterogeneous platforms consisting of multicore central processing units (CPUs), graphics processing units (GPUs), digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and other processors or hardware accelerators.

- OPEN GL – Developed and maintained by the Kronos Group, is a cross platform application programming interface (API) for rendering 2D and 3D vector graphics. The API is typically used to interact with a graphics processing unit (GPU), to achieve hardware-accelerated rendering.

- DIRECT X - is a collection of APIs or handling tasks related to multimedia, game programming and video, on Microsoft platforms.
Existing Approaches – Romein Griddler

First known attempt to parallelize griddler for GPU, created by John W Romein

Grid broken up into parallelized worker threads (A-Y) separated by kernel width

Each worker is responsible for its own grid points spread over the entire grid.

The worker iterates over visibilities whose kernel overlaps it and updates the grid point as necessary.

No need to worry about two workers trying to update the same grid point:

**Disadvantage:** Kernels have to be fixed width! (MAX SUPPORT)
Existing Approaches – NAG Gridder

From Numerical Algorithm Group in Manchester.

Breaks grid up into “tiles” where a single work group works on a single tile.

Visibilities are first bucket sorted into which tile regions they fall. If a visibility overlaps multiple tiles, the visibility is duplicated and placed into all regions it overlaps.

Region Buckets: Blue worker: A,C,D
Red worker: B,C
Pink worker: C
Green worker C,D

Each worker iterates through the visibilities in the tiled region and updates its corresponding grid points.

Only the parts of the kernel that fall within the tile are processed.

Disadvantage: Need to sort visibilities (difficult with so many coming in from telescope per second). Many visibilities landing over multiple tiled regions causing visibility duplication meaning more data transferred into GPU.
GPU Pipelines for Graphics
Basic Open GL pipeline: multi-pass rendering example
Our Approach: HEC Griddler

To view the griddler more as a rendering problem rather than an HPC problem.

We use the Open GL graphics pipeline to allocate a giant output texture for the grid attached to a “frame buffer” object. This frame buffer is set to “add mode” which accumulates visibility values over time.

Grid data is copied back to main memory from the GPU when the dump time has been reached. Possibly kept on GPU to do dFFT first.

Visibility values and their $u,v,w$ coordinates are passed into the graphics pipeline as GL_POINT geometry attributes.

**Vertex Shader on GPU** – Used to convert each visibility’s $u,v$ coordinates into grid coordinates. Geometry for each GL_Point is scaled depending on its $w$ coordinate.

**Frag Shader on GPU** – Used to smear visibility on grid using the bound sprite texture (w-plane kernel). A texture lookup is used to perform a complex multiplication of the visibility strength with the values in the texture kernel.

The inbuilt hardware rasterizer is responsible for all parallelization and GPU thread allocation, converting vertex geometry into fragments – no need to worry about concurrent update problems of overlapping visibilities.

In a nutshell, the griddler is now just rendering millions of overlapping sprite textures to a giant display. Exactly what a GPU is designed to accomplish.

All data is single precision.
Implementation of W-Projection

*W-projection* is used to correct the non-coplanar baseline effect.

Many kernels are pre-calculated at fixed size depending on w-coordinate support range. Too computationally expensive to calculate precise kernel at “w”, so usually w-projection done using the nearest w-plane.

In our implementation each kernel in a w-plane is stored in a 3D sprite texture. Texture is set to “linear interpolation”. If a $u,v$ coordinate does not land on an exact grid point and w-plane, the texture unit will automatically interpolate values from texture coordinates around the grid point in all three dimensions.

This roughly halves the amount of oversampling needed, as values are interpolated instead of being snapped to the nearest grid oversample or w-plane.

Only need to store kernels for positive “w” terms as the negative terms are just the *complex conjugate*. 

Kernel is symmetric around centre, so actually only need quarter of each plane and set texture to “reflect”.
Oversampling textures

Usually when $w=0$, the applied kernel on a visibility is small and when $w=\text{max}$ then the maximum kernel size is applied.

Because we are using 3D sprites each w-plane must initially have the same dimensions.

So instead we oversample smaller kernels, such that each plane is the same size. Smaller kernels will have better resolution allowing a more accurate kernels when $w$ is small (usual case).

Each texture kernel is scaled on the GPU as it is applied to a visibility depending on its $w$-value.

If $W$ is big, phase screen more spread out, so applied kernel needs to be bigger. If $W$ small, then phase screen more condensed, so need to sample more to make w-Plane size equal for texturing
HEC Gridder

Texture sprites just drawn around central \(u,v\) coordinate. Texture size scaled down on GPU to match kernel size and interpolated between \(w\)-planes.

Any elements that are overlapped with texture are given interpolated values.
Results – GPU Comparison

Testing done with *NVidia GTX Titan X* and compared to NAG Gridder code. (Not fully optimized for Titan X?)

Also included is NAG’s results in paper for all 4 tests. Highly optimized for *Tesla P100*

**GTX Titan X:**
- 28 Streaming multiprocessors
- 128 cores per SM
- 336 GB/s bandwidth
- 12 GB GDDR5 Memory
- 6.6 teraflops (single)
- 250W Power

**Tesla P100**
- 56 Streaming multiprocessors
- 64 cores per SM
- 720 GB/s bandwidth
- 16 GB HB M2 Memory
- 10.6 teraflops (single)
- 300W Power

# CUDA Benchmarks

<table>
<thead>
<tr>
<th>Device</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tesla V100-SXM2-16GB</td>
<td>697836</td>
</tr>
<tr>
<td>Tesla V100-PCIE-16GB</td>
<td>511768</td>
</tr>
<tr>
<td>Tesla P100-PCIE-16GB</td>
<td>319429</td>
</tr>
<tr>
<td>TITAN Xp</td>
<td>245494</td>
</tr>
<tr>
<td>TITAN X (Pascal)</td>
<td>237756</td>
</tr>
<tr>
<td>GeForce GTX 1080 Ti</td>
<td>236124</td>
</tr>
<tr>
<td>GeForce GTX 1080</td>
<td>207239</td>
</tr>
<tr>
<td>Graphics Device</td>
<td>201845</td>
</tr>
<tr>
<td>GeForce GTX 1070 Ti</td>
<td>196628</td>
</tr>
<tr>
<td>GeForce GTX 1070</td>
<td>180936</td>
</tr>
<tr>
<td>Quadro P5000</td>
<td>179126</td>
</tr>
<tr>
<td>GeForce GTX 980 Ti</td>
<td>151160</td>
</tr>
<tr>
<td>GeForce GTX TITAN X</td>
<td>144570</td>
</tr>
</tbody>
</table>
# Results: Throughput

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Grid Size</th>
<th>Num Visibilities</th>
<th>Num W Planes</th>
<th>Min-Max Full Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthetic</td>
<td>10K</td>
<td>2,152,800</td>
<td>714</td>
<td>8 – 73</td>
</tr>
<tr>
<td>EL30-56</td>
<td>18K</td>
<td>31,395,840</td>
<td>922</td>
<td>8 – 191</td>
</tr>
<tr>
<td>EL56-82</td>
<td>18K</td>
<td>31,395,840</td>
<td>601</td>
<td>8 – 145</td>
</tr>
<tr>
<td>EL82-70</td>
<td>18K</td>
<td>31,395,840</td>
<td>339</td>
<td>8 – 89</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Titan X (NAG - **Not fully optimized?) GPU Processing Time + Transfer (ms)</th>
<th>P100 (NAG) reported in white paper (ms)</th>
<th>Titan X (HEC) GPU Processing Time (ms)</th>
<th>Transfer time HEC Entire grid (3K portion)</th>
<th>Titan X (HEC) Full processing time + full grid transfer (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthetic</td>
<td>1,735</td>
<td>74</td>
<td>113</td>
<td>1,079 (83)</td>
<td>1,192</td>
</tr>
<tr>
<td>EL30-56</td>
<td>27,444</td>
<td>1410</td>
<td>3,682</td>
<td>3,559 (81)</td>
<td>7,241</td>
</tr>
<tr>
<td>EL56-82</td>
<td>20,486</td>
<td>781</td>
<td>2,010</td>
<td>3,525 (78)</td>
<td>5,535</td>
</tr>
<tr>
<td>EL82-70</td>
<td>16,548</td>
<td>553</td>
<td>1,041</td>
<td>3,509 (83)</td>
<td>4,550</td>
</tr>
</tbody>
</table>
Results - Power

IDLE POWER: 97 Watts for machine, both tests
NAG Gridder: 248 Watts.
HEC Gridder: 280 Watts.
Screenshot of Gridder

EL82-70 sample.
Max support 89
Min support 8
339 W planes
31,395,840 visibilities
Advantages and Disadvantages

Advantages:
◦ Using OpenGL which is standard language for GPUs – will work on almost any model GPU
◦ Closer to how an actual GPU pipeline works, grids faster than NAG gridder (preliminary results).
◦ Inbuilt hardware interpolation of kernel textures saves kernel memory, no need to store 8x Oversampled kernels and also can cut down on amount of w-planes. Future use of Radial Texturing will reduce memory even more.

Disadvantages:
◦ Graphics pipeline uses only single precision values
◦ OpenGL context expects a display, need to mimic a connected screen.
◦ Transfer time of memory from GPU back to CPU seems inefficient with OpenGL – although not a big deal considering grid transfer time occurs once per major cycle.
Questions