Background
Convolution gridding is an important part of radio astronomy image processing. Interferometers measure fringe visibilities of celestial objects. These are convolved in the Fourier domain with a set of convolution kernels, which don’t all have the same size support. Results are accumulated into a grid. A visibility’s $w$ coordinate determines which kernel is used.

Parallelization Using Atomics
As a first step, the code was parallelized using atomics to handle the race conditions. It was infeasible to run the serial code on the GPU.

Tiling for Better Data Locality
Partitioning the grid into tiles improves data locality. Tiles were sized to fit into registers (GPU) or cache memory (KNL). Visibilities were assigned to tiles using a parallel bucket (radix) sort. Each tile was processed by a thread block (GPU) or thread (KNL). Because tiles don’t overlap the race conditions could be avoided.

The distribution of grid updates across tiles is highly uneven, leading to poor load balance. Ordering tiles by the number of visibilities and processing from highest to lowest mitigated this. A different approach was needed on the GPU to handle the highly concentrated central region.

Improving Memory Access Patterns
Convolution kernel data is accessed in a complicated way: it is strided for “kernel oversampling” and traverses the arrays in $x$ and $y$ directions to exploit symmetry. This leads to non-coalesced loads on the GPU and vector gathers on the KNL, both of which are expensive. We decoded the access pattern and found a way to enforce contiguous data access. This enabled coalesced loads and eliminated the vector gathers, at the expense of introducing some data duplication on the KNL only.

NAG is Approached to Optimize the Code
As part of preliminary work on the Square Kilometer Array’s digital signal processing pipeline, NAG was approached by the Astronomy group at Oxford University to optimize the code on NVIDIA P100 GPU and Intel KNL.

Performance Results
GPU results were from an NVIDIA P100, while KNL results were generated on a 68-core Xeon Phi 7250 (Knights Landing) 1.4GHz machine with 94GB of DDR RAM and 16GB of high-bandwidth MCDRAM.

Lessons Learnt
In optimizing the code on the two architectures a number of things became clear

**KNL:**
- The KNL has 37MB cache overall, but only 272KB cache per thread (compare with 2.83MB cache per thread on x86). Getting good performance for cache-hungry applications like this one is challenging.
- Threads on a core share L2 cache but there’s no way to synchronize between them, making it hard to exploit L2 cache effectively.
- Traditional x86 CPUs with bigger caches are far more forgiving.
- Tiling is important on KNL and x86: atomics are not that fast.
- Using MCDRAM made no difference as the code was bound by memory latency not bandwidth.

**P100:**
- Hardware atomics are surprisingly fast.
- Effective use of the register file is crucial for peak performance.
- Non-coalesced loads are expensive even on modern GPUs.
- Coalesced memory access is easier to achieve on GPUs than contiguous access is on CPUs, because loads can be coalesced even if consecutive threads are not accessing consecutive memory addresses.