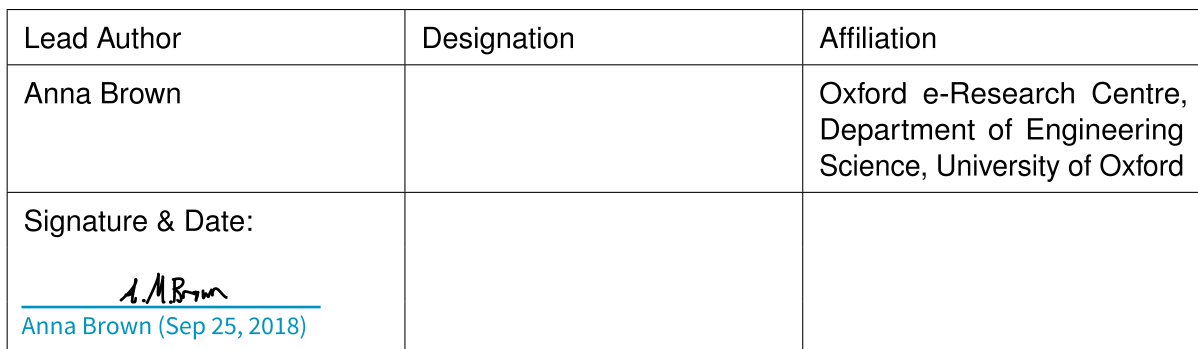
SDP Memo 74: Optimisation of the w-projection gridding algorithm for FPGA using Intel OpenCL

Document Number .... SDP Memo 74
Document Type .... MEMO
Revision 1
Author .... Anna Brown, Suleyman Demirsoy
Release Date 2018-09-25



Lead Author	Designation	Affiliation
Anna Brown		Oxford e-Research Centre, Department of Engineering Science, University of Oxford
Signature & Date: Anna Brown (Sep 25, 2018)		

Revision	Date of issue	Prepared by	Comments
1	2018-09-25	Anna Brown	

SDP Memo Disclaimer

The SDP memos are designed to allow the quick recording of investigations and research done by members of the SDP. They are also designed to raise questions about parts of the SDP design or SDP process. The contents of a memo may be the opinion of the author, not the whole of the SDP.

Table of Contents

1 Abstract **3**

2 Introduction **3**

3 FPGA optimisation background **3**

4 Algorithm features **5**

5 Timing method and reference results **5**

6 Optimisations **6**

6.1 Single serial pipeline 6

6.2 Single parallel pipeline 10

7 Performance projections – multiple parallel pipelines **11**

8 Developer time **12**

9 Conclusions **12**

10 Acknowledgements **13**

References **13**

1 Abstract

An initial attempt at optimising the w-projection gridding algorithm for FPGA was made using the Intel OpenCL SDK for FPGA on an Arria 10 development device. In the time available only a serial pipelined version could be completed, with partial progress on a parallel pipeline version. A multiple pipeline version of this parallel pipeline code could be expected to run on the hardware available in at best 11255 ms, compared to a CPU time of 2310ms and a GPU time of 438 ms. However, further possible optimisations to the parallel pipeline code remain, including more efficient parallel access to local memory and trimmed down local memory usage. The OpenCL SDK was found to be generally easy to use for a beginner to FPGA programming with no VHDL/Verilog experience.

2 Introduction

This report summarises work done to port and optimise the w-projection gridding algorithm for FPGA. The goals of the work were to estimate how the FPGA architecture would perform for gridding in SDP, as well as evaluate how the availability of the Intel FPGA SDK for OpenCL affects developer time on the FPGA. This report also includes some guidelines for optimisation learnt as part of this work.

A description of the w-projection algorithm can be found in [2]. The data set used to test the performance of the FPGA is available at the gridding challenge page `https://confluence.ska-sdp.org/display/WBS/Gridding+Challenge` as 'SKA 56-82'. The FPGA code was verified using a single threaded CPU version available as part of the gridding challenge.

The starting point for the optimisation was a tile-based approach for exposing parallelism, used in GPU and multithreaded CPU versions described in SDP memo 36 [1]. Several optimisations were tried on an Arria 10 development board, with guidance by Suleyman Demirsoy of Intel. The performance and resource usage of the prototype optimisation was then used to estimate performance with further parallelism introduced.

3 FPGA optimisation background

Optimisation strategies relevant to this algorithm are introduced below:

Pipelining and Initiation Interval (II)

Pipelining is one of the critical ways to achieve parallelism on the FPGA. For best performance, a new iteration of a loop will enter the pipeline on every clock cycle – this can be expressed as the initiation interval (II) being equal to 1. Higher II means each iteration needs to wait that many clock cycles before entering the pipeline, with a total time of II $\times$ number of interations + latency. Issues which can increase the II or, in the worst case, cause a section of code to not be pipelined at all, include memory dependencies between iterations of a loop and a variable sized inner loop causing an outer loop to not be pipelined.

II was measured using static analysis without needing a lengthy compile for the device, using

```
aoc -c gridding_kernels_tile.cl
```

in the root directory. This will create a html report in gridding_kernels_tile/reports, and II can be found in View Reports/Loops Analysis.

Latency and channels

The latency of a pipeline, or the total clock cycles for one iteration to pass through the pipeline, have a negligible effect on performance in the case that II=1 and the number of iterations is large. However, for a loop that is not pipelined at all, II = latency, and reducing the latency of the pipeline can have a large effect.

Channels are Intel OpenCL constructs which can be used to send data between OpenCL kernels in a decoupled way. Each channel has a specified depth, and the first kernel will write to the channel in a pipelined way, without stalling until the channel is full to the depth specified. At any time, as long as there are any items available in the channel, the second kernel can also read values from the channel.

In the case where an outer loop cannot be pipelined due to a variable size inner loop, the latency of the outer loop can therefore be hidden by moving the outer loop to one OpenCL kernel, connected by a channel to a second kernel. The second kernel consists of a light, low latency outer loop which only reads data fed from the first kernel, and processes it in the inner loop in a pipelined manner.

Localising memory

On the Arria 10, there are two memory channels each with 19,200 MB/s maximum bandwidth (the development device used here had only one channel). By contrast, the comparison GPU used, the NVIDIA P100, has a maximum theoretical bandwidth of 732 GB/s. The comparison CPU used, the Intel Xeon E2670, has a maximum theoretical bandwidth of 68BG/s per socket. It is therefore critical to ensure there is memory reuse from local memory banks to avoid being bottlenecked by global memory bandwidth. This may be less of a problem on the newer Stratix 10, which has 4 memory channels, and it is possible that the limiting factor may change completely with the addition of HBM – a maximum of 16GB at peak theoretical bandwidth of 512 GB/s are available for the Stratix 10 at the time of this report.

Accessing global memory

Where accesses to global memory cannot be avoided, they can be made more performant by accessing memory in coalesced bursts. The maximum data rate is limited by the word size to global memory – 512 bits can be read in one cycle. In addition, up to 16 words can be accessed in one request (ie 16 cycles to read 16 words), or 'burst' if the data being read is consequtive in global memory. Accessing outside of this pattern will cause global memory stalls, and stalls can also be caused by competition with other memory masters (ie other concurrent requests to the global memory, no matter if the data accessed is small).

Resource usage

There is a maximum limit to the compute and memory resources that are available to be allocated on the device. The more efficiently a single pipeline can use these resources, the more chance there is to scale the algorithm to multiple pipelines. This can be done most efficiently if any single pipeline uses a similar percentage of each resource:

- ALUTs: Arithmetic lookup tables – logic units
- FFs: Flip flops – registers
- RAMs: Block RAM
- DSPs: Digital signal processing blocks – floating point or integer addition/multiplication units

4 Algorithm features

The most naive version of the algorithm simply iterates over all visibilities serially. For each visibility, it convolves the visibility value with the correct convolution kernel based on the visibility coordinates, then adds the result to the output grid.

In order to expose parallelism in this algorithm for CPU and GPU versions, the output grid was divided in both dimensions into tiles.

The main gridding function consists of four loops as follows:

```
for all tiles in global grid:
    for all visibilities in tile:
         for v dimension in convolution kernel (that overlaps with tile)
              for u dimension in convolution kernel (that overlaps with tile)
                   convolve visibility value with convolution kernel
                   and add back to grid
```

5 Timing method and reference results

The tile-based version of the algorithm requires additional data preparation functions in addition to the main gridding function, in order to bucket sort the visibilities into tiles. As the time taken for these functions is short compared to the time for the main gridding step, only the gridding function time was compared between CPU, GPU and FPGA.

The reference CPU time was collected on a 2 socket Intel Xeon E2670, for a total of 24 threads. For the tile-based code, the CPU time for the gridding function on this system was approximately 2310 ms (taken from [1] Table 11 and subtracting preprocessing functions in Table 7).

The reference GPU time was collected on an NVIDIA P100 and the GPU time was 438ms.

Version	Time for gridding function (ms)
v_tile_localTile	393401
v_tile_ivdep_localTile	84465
v_channels	84978
v_channels_reversedKernel	81852
v_channels_localKernel	75210
v_channels_unroll (unroll 4)	67530
24 core Xeon E2670 CPU	2310
P100 GPU	438

Table 1

6 Optimisations

The different optimisations tried can be found in branches of the https://github.com/OxfordSKA/FPGA_Gridding/tree/master repo, in the gridding_kernels_tile.cl file. Unless otherwise stated, optimisations are given here in order, ie previous optimisations are included in later versions. The general strategy was to optimise the hardware first for a single, serial pipeline (ensuring II=1 etc), then parallelise that pipeline within a single OpenCL kernel (by making parallel accesses to local memory banks, etc), then parallelising with multiple kernels. Where this work couldn't be finished, estimates were made for how effective such optimisaitons might be.

Table 1 shows a summary of all optimisations tried.

6.1 Single serial pipeline

Ensuring no data dependency to enable pipelining

For one visibility, each (u,v) coordinate in the convolution kernel centred on that visibility touches a unique cell in the global output grid. However, due to the nature of the global grid index lookup, the compiler cannot detect this automatically, and the loop is processed with no pipelining. The compiler can be told that it is safe to pipeline the loop using

```
#pragma ivdep
```

above both inner loops.

This optimisation reduces II from 173 to 1 (for a speedup of 173x) when grid updates are made directly to global memory (see version v_tile_ivdep as compared to version v_tile) for each visibility, and reduces II from 5 to 1 when grid updates are made to tiles stored in temporary local memory (see version v_tile_ivdep_tileLocal as compared to version v_tile_tileLocal).

Timings for the gridding step for the versions mentioned above is shown in Table 2

Version	Time for gridding function (ms)	II of convolution kernel inner loop
v_tile	Infeasible to measure – too long	173
v_tile_ivdep	114576	1
v_tile_localTile	393401	5
v_tile_ivdep_localTile	84465	1

Table 2

Reusing tile data in local memory

All visibilities in one tile access the same section of the global grid. Each tile was therefore preloaded into local memory and all visibilities in that tile were convolved onto the local memory before the tile being loaded back to global memory (compare v_tile_ivdep_tileLocal to v_tile_ivdep in Table 2).

The size of local memory required is:

$$\begin{aligned} \text{TILE_SIZE} \times \text{TILE_SIZE} \times \text{sizeof(complex float)} &= 64 \times 64 \times 8B \\ &= 32KB \end{aligned}$$

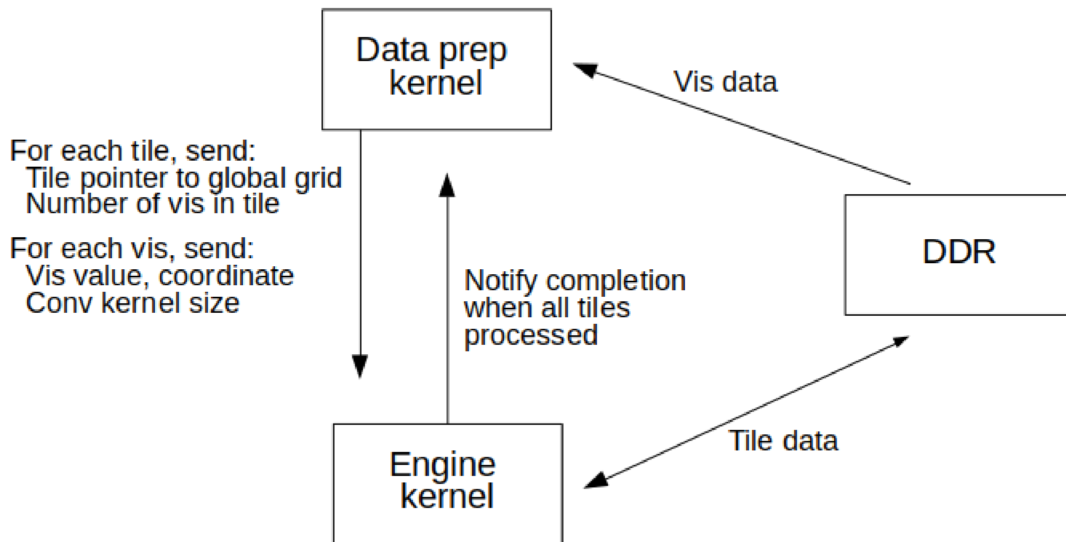
Channels

The loop over visibilities could not be pipelined in v_tile_ivdep_tileLocal due to the fact that the number of iterations in the inner loops over the two dimensions of the convolution kernel could vary with visibility (due to varying wsupport size or visibility overlap with the edges of the tile). This was a problem as the latency of the loop over visibilities is high (81 cycles) – several global memory accesses need to occur here to load visibility data.

As described in Section 3, the gridding function was split into two functions. One iterated over tiles and visibilities and prepared visibility data to pass to the main engine, and one received visibility data and processed it. This is described below:

```
Data preparation OpenCL kernel:
for all tiles in global grid:
    Send engine kernel tile location in global grid and number of
    visibilities in tile
    for all visibilities in tile:
         send engine kernel visibility coordinate, value and extent of
         kernel that overlaps with tile

Engine OpenCL kernel:
for all tiles in global grid:
    receive tile pointer from preparation kernel
    load tile to local memory
    for all visibilities in tile:
         receive visibility data from preparation kernel
         for v dimension in convolution kernel (that overlaps with tile)
              for u dimension in convolution kernel (that overlaps with tile)
                   convolve visibility value with convolution kernel
                   and add back to local tile
    save local tile back to global grid
```



The time for the new (v_channels) channel implementation and previous best implementation are shown in Table 3. There is no improvement seen at this stage, likely due to the more complicated layout of the channels implementation imposing a lower maximum clock frequency on the device, as seen in Table 3. The value of fmax can be found by performing a regular compile for the device. This will produce a file gridding_kernels_tile/acl_quartus_report.txt which includes the value of fmax as well as resource usage on the device.

Accessing kernels from global memory

In v_channels, kernels are accessed from global memory in bursts of only 1.8 where the peak would be 16, reducing the performance of the global memory access.

Due to the way in which symmetry is exploited when storing convolution kernel values, the inner loop in the large engine kernel in this version was accessing convolution kernel values from global memory in sequential order for visibilities with positive oversample offsets and reverse sequential order for visibilities with negative oversample offsets. As efficient memory accesses in burst can only be done if the memory is being accessed in sequential order, the code was modified in v_channels_reversedKernel to reverse the lookup for those visibilities that were in reverse order. This was done by reversing the loop over convolution kernel values for those cases.

Version	Time for gridding function (ms)	fmax (MHz)
v_tile_ivdep_localTile	84465	303.9
v_channels	84978	280.0

Table 3

Version	Time for gridding function (ms)	Bandwidth (MB/s)	Burst size (max 16)
v_channels	84978	1587	1.8
v_channels_reversedKernel	81852	reversed case: 758 non-reversed case: 1254	reversed case: 3.1 non-reversed case: 3.1

Table 4

Table 4 shows the time, bandwidth and burst size of the two versions. For the version with kernel reversing, the code branches to perform the inner loop over convolution kernels in reverse order when required, and bandwidth and burst size is shown for both cases. As can be seen, the burst size is increased by always accessing kernels from global memory in sequential order. However, this can't be increased to the maximum size of 16, presumably due to the fact that many kernels either have less than 16 elements along the u dimension, or are at the edge of the tile and therefore don't access the full convolution kernel.

The bandwidth and burst size was found by using a runtime profiler. The code was adjusted to change autorun kernels, which cannot be profiled, to explicitly launched kernels. The code was then compiled with

```
aoc -v gridding_kernels_tile.cl -profile
```

Running the code then generated the usual files, plus a source file and profile.mon files containing profiling data. The profiler could then be run using

```
aocl report gridding_kernels_tile.aocx profile.mon gridding_kernels_tile.source
```

Reusing convolution kernel data in local memory

In the original CPU tile based version, visibilities are sorted by u and v position into tiles, and then by w coordinate within tiles. In v_channels_localKernel, this last sorting is exploited by loading kernels into local memory. This needs to be done only when the w coordinate of the visibilities in a tile changes, as w coordinate determines which convolution kernel to use. Convolution kernels can then be accessed from local memory by all visibilities that require that convolution kernel within one tile.

There are two complications here. One is that different visibilities that have the same w coordinate can different oversample offsets depending on their u and v coordinates, requiring different parts of the convolution kernel. As these values can't be predicted ahead of time, the convolution kernel for all oversample offsets must be loaded. If there is only one visibility value with a particular w coordinate in a tile that visibility will actually access more data from global memory than in the previous version.

The other issue is that there are a few rare visibilities with very large w coordinate, and thus convolution kernel size. In practice it is infeasible to allocate enough local memory to handle these edge cases. Instead, a flag was added to the (CPU implemented) bucket sort for every tile. If any visibility in the tile had a wsupport size larger than MAX_W_SUPPORT_LOCAL the entire tile was marked as a large tile and processed with the old version's engine kernel. All other tiles were handled by a new small engine that preloads the convolution kernels into local memory. The engine kernel to use was selected by the data preparation kernel.

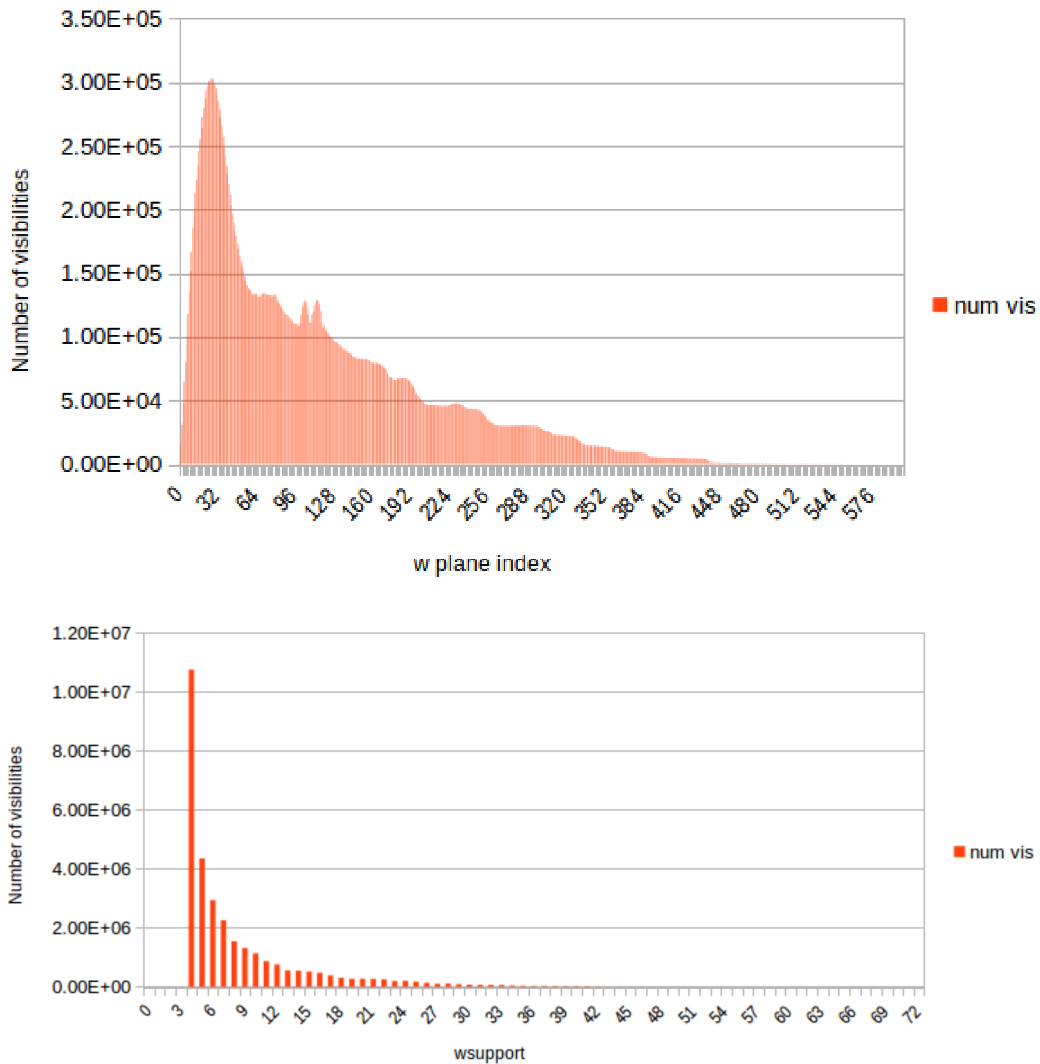


Figure 1

Fortunately, tiles which have few visibilities with the same w coordinate also tend to be tiles with large wsupport, meaning that the level of reuse in the small engine should be high. Figure 1 shows the distribution of w coordinates in the data set used.

The best size for MAX_W_SUPPORT_LOCAL was found to be 25 and the time for the gridding function in the new version was 75210 ms.

6.2 Single parallel pipeline

All versions previously described have read one piece of data or performed one calculation per pipeline step in the loop over convolution kernel dimensions in the engine kernel. However, the compiler can be instructed to assign more resources to each pipeline step, which allows multiple convolutions to be performed in parallel while still maintaining pipelined behaviour. Loop unrolling is one way to do this.

Parallel accesses to tiles in local memory

Both the small and large engine kernels update tiles stored in local memory in the inner loops over convolution kernels. Adding

```
#pragma unroll 4
```

to the loop over the u (fastest varying) dimension will results in four updates to the tile being performed at once. Naively, the compiler will achieve this by making four full copies of the array held in four banks, in order to give four memory load units access to the array at the same time. This version is in v_channel_unroll and with an unroll value of 4 takes 67530 ms.

In order to prevent memory duplication, the memory banking can be done explicitly, dividing the array into four parts in four banks. In addition, the local memory could be accessed in such a way that the same load unit always accessed the same memory bank at all times. This would minimise the number of load units that need to be assigned. A work in progress of this attempt is at v_channel_unroll_efficientTile.

7 Performance projections – multiple parallel pipelines

After creating a optimal versions of the small and large engine kernels, additional parallelism can be introduced by making multiple copies of these kernels. An upper bound on performance can be estimated by assuming perfect scaling with the number of kernels, up to the limit that will fit on the device. Table 5 shows the resource usage for the three kernels in v_channel_unroll.

All three kernels are limited by RAMs for this version. Assuming that the data preparation kernel is left the same and the small and large kernels are both replicated as many times as will fit, this leaves 71% of the total device free to replicate an area of 11 % – enough to replicate both kernels 6 times. In reality, as the large kernel involves frequent global memory accesses to retrieve convolution kernel values, it is likely that the optimal choice would be one large engine kernel and multiple small engine kernels. As the smaller kernel uses more RAMs, the replication factor would be less than 6.

With a speedup of 6, the FPGA version would take 11255 ms, still slower than the CPU version at 2310ms. There are several potential avenues for making the FPGA version competitive. They would likely need to be applied in combination.

- The value of 29 % RAM use for the data preparation engine is suspiciously large. Investigation showed that this was due to every piece of visibility data read from global memory being stored in a private 512 kilobit cache. This could not be disabled in the time available – neither removing const from those variables or adding volatile seemed to have an effect. If this could be disabled successfully, the RAM usage of this kernel should be reduced significantly. There should also be scope to reduce RAM usage by improving the access patterns for visibility data – for instance by loading all three visibility coordinates in one request. If the RAM usage could be reduced to a similar level as the other kernels, those kernels could then be replicated 8 times for an estimated run time of 8441 ms.

Kernel	ALUTs (%)	FFs (%)	RAMs (%)	DSPs (%)
small convolution kernel engine	1	1	7	3
large convolution kernel engine	2	2	4	3
data preparation kernel	9	5	29	5

Table 5

- The parallel versions of the small and large engines are not yet optimal – the layouts of tile and kernel data stored in local memory should be possible to improve to use fewer RAMs and fewer load units. Decreasing the time and resource usage of these kernels would naturally improve the performance of the final version with replicated kernels. Note that increasing the parellisation of the inner loop over convolution kernels would also have a knock on effect on the run time as it would in effect decrease the latency of the outer loops, which cannot be parallelised and are therefore sensitive to latency.
- Alternatively, it might be possible to parallelise the engine kernels in such a way that ALUT use is increased even if RAM use is not decreased, increasing performance while allowing for the same amount of duplication of the engine kernels.
- The large and small engine kernels are currently run serially – either one or the other executes. It should be possible for them to operate simultaneously to make better use of the requested resources.
- The board used was a development board, with half as many memory channels as in the production Arria 10 board – as the large engine is bound by global memory accesses, using a full board should improve the performance of this kernel.

8 Developer time

I began this project with no experience of FPGA programming or optimisation principles, and limited experience with OpenCL. In my subjective opinion, the process of learning to develop and optimise for FPGA using OpenCL felt similar to that of learning to optimise for GPU in CUDA. There was very little syntax to learn beyond the usual C – the work was in understanding the architecture and using it correctly, which is also necessary for GPU work.

The main challenge occurred when there was a bug which was only present when run on the actual device, not on the emulator. In this case I had to wait around 40 minutes for code to compile in fast mode for the device every time a change was made. This tended to occur for problems related to concurrency, which the emulator does not handle. In general, however, development and debugging could be done on the emulator, with compilation done in less than a minute. Gdb was used for debugging in emulator mode.

Finally, time for measuring execution time was longer still as the code could not be compiled for the device in fast mode and for me took more than an hour. However, much of the profiling work could be done with static analysis, which again took less than a minute. Finally, there is a run time profiler available which gives statistics such as memory bandwidth and stall rate, but again needed to be compiled for the device.

9 Conclusions

Several optimisation strategies were tried while creating a version of the w-projection gridding algorithm for FPGA using the Intel SDK for OpenCL. A version of the best single pipeline implementation created in this project scaled up to multiple pipelines could be estimated to run on the hardware available in at best 11255 ms, compared to a CPU time of 2310ms and a GPU time of 438 ms. However, a fully optimised single pipeline implementation could not be completed in the time available and it should be possible to improve performance before scaling to multiple pipelines, particularly through an improved parallel local memory access strategy.

10 Acknowledgements

Thanks to Suleyman Demirsoy (Intel) for detailed guidance in using the Intel SDK for OpenCL and in FPGA optimisation theory.

References

[1] Dingle, du Toit & Hopkins, *SDP Memo 36: Convolution Gridding on CPU, GPU and KNL*, The Numerical Algorithms Group, Manchester, 2017

[2] Cornwell, Golap & Bhatnagar (2008), IEEE Journal of Selected Topics in Signal Processing, Vol. 2, Issue 5, p.647-657